The OpenCL™ C++ 1.0 Specification

Khronos® OpenCL Working Group

Version v2.2-10, Tue, 05 Feb 2019 21:18:48 +0000

Table of Contents

Copyright 2008-2019 The Khronos Group.

This specification is protected by copyright laws and contains material proprietary to the Khronos Group, Inc. Except as described by these terms, it or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of Khronos Group.

Khronos Group grants a conditional copyright license to use and reproduce the unmodified specification for any purpose, without fee or royalty, EXCEPT no licenses to any patent, trademark or other intellectual property rights are granted under these terms. Parties desiring to implement the specification and make use of Khronos trademarks in relation to that implementation, and receive reciprocal patent license protection under the Khronos IP Policy must become Adopters and confirm the implementation as conformant under the process defined by Khronos for this specification; see https://www.khronos.org/adopters.

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

Vulkan and Khronos are registered trademarks, and OpenXR, SPIR, SPIR-V, SYCL, WebGL, WebCL, OpenVX, OpenVG, EGL, COLLADA, glTF, NNEF, OpenKODE, OpenKCAM, StreamInput, OpenWF, OpenSL ES, OpenMAX, OpenMAX AL, OpenMAX IL, OpenMAX DL, OpenML and DevU are trademarks of the Khronos Group Inc. ASTC is a trademark of ARM Holdings PLC, OpenCL is a trademark of Apple Inc. and OpenGL and OpenML are registered trademarks and the OpenGL ES and OpenGL SC logos are trademarks of Silicon Graphics International used under license by Khronos. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.

Acknowledgements

The OpenCL C++ 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:

* Eric Berdahl, Adobe
* Aaftab Munshi, Apple
* Brian Sumner, AMD
* Andrew Richards, Codeplay
* Maria Rovatsou, Codeplay
* Adam Stański, Intel
* Alexey Bader, Intel
* Allen Hux, Intel
* Bartosz Sochacki, Intel
* Ben Ashbaugh, Intel
* Kevin Stevens, Intel
* Łukasz Dudziak, Intel
* Łukasz Towarek, Intel
* Marcin Walkowiak, Intel
* Michael Kinsner, Intel
* Raun Krisch, Intel
* Tomasz Fiechowski, Intel
* Kedar Patil, NVIDIA
* Yuan Lin, NVIDIA
* Alex Bourd, Qualcomm
* Lee Howes, Qualcomm
* Anton Gorenko, StreamComputing
* Jakub Szuppe, StreamComputing
* James Price, University of Bristol
* Paul Preney, University of Windsor
* Ronan Keryell, Xilinx
* AJ Guillon, YetiWare Inc.

Chapter 1. Generic Type Name Notation

The generic type names are used when some entity has multiple overloads which differ only by argument(s). They can map to one or more built-in data types. The tables below describe these mappings in details.

Assuming that `gentype` maps to built-in types: `float`, `int` and `uint`, when coming across definition:

```
gentype function(gentype x);
```

reader should understand that such function has in fact three overloads:

```
float function(float x);
int function(int x);
uint function(uint x);
```

Note that if a function signature has multiple usages of `gentype` they all should map to the same type. Following this rule such overloads are then invalid:

```
float function(int x);
uint function(float x);
// etc.
```

If a function is meant to have such overloads, respective gentypes in its signature should be postfixed with numbers to indicate they represent different types. Declaration like this:

```
cl::common_type_t<gentype1, gentype2> greater(gentype1 x, gentype2 y);
```

would match following overloads:

```
cl::common_type_t<float, float> greater(float x, float y);
cl::common_type_t<float, int> greater(float x, int y);
cl::common_type_t<float, uint> greater(float x, uint y);
cl::common_type_t<int, float> greater(int x, float y);

// etc.
```

Table 1. generic types

generic type	corresponding built-in types

`typen`	scalar and all vector types of type
	
	Example:
	
	`floatn` matches: `float`, `float2`, `float3`, `float4`, `float8` and `float16`
	`floatn` doesn't match: `half`, `int2`

`gentype`	unspecified in global context, should be defined whenever used
`sgentype`	subset of scalar types from types matched by `gentype`
`ugentype`	subset of unsigned integer types from types matched by `gentype`
`gentypeh`	`half`, `half2`, `half3`, `half4`, `half8` or `half16`
`gentypef`	`float`, `float2`, `float3`, `float4`, `float8` or `float16`
`gentyped`	`double`, `double2`, `double3`, `double4`, `double8` or `double16`

Chapter 2. OpenCL C++ Programming Language

This section describes the OpenCL C++ programming language used to create kernels that are executed on OpenCL device(s). The OpenCL C++ programming language is based on the ISO/IEC JTC1 SC22 WG21 N 3690 language specification (a.k.a. C++14 specification) with specific restrictions (see the *OpenCL C++ restrictions* section). Please refer to this specification for a detailed description of the language grammar. This section describes restrictions to the C++14 specification supported in OpenCL C++.

2.1. Supported Built-in Data Types

The following data types are supported.

2.1.1. Built-in Scalar Data Types

Table 2. Device Built-in scalar data types

Type	Description
`bool`	A data type which is either `true` or `false`. (*See section 2.14.6 lex.bool and section 3.9.1 basic.fundamental of the C++14 Specification.*)
`char`, `signed char`	A signed two's complement 8-bit integer.
`unsigned char`	An unsigned 8-bit integer.
`short`	A signed two's complement 16-bit integer.
`unsigned short`	An unsigned 16-bit integer.
`int`	A signed two's complement 32-bit integer.
`unsigned int`	An unsigned 32-bit integer.
`long`	A signed two's complement 64-bit integer.
`unsigned long`	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.
`void`	The `void` type comprises an empty set of values; it is an incomplete type that cannot be completed.

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:

Table 3. Host Scalar Built-in Data Types

Type in OpenCL Language	API type for application
`bool`	n/a, i.e., there is no corresponding `cl_bool` type.
`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`
`void`	`void`

Built-in 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. All other operations are not allowed if the **cl_khr_fp16** extension is not supported.

A few valid examples are given below:

```
#include <opencl_def>
#include <opencl_memory>
#include <opencl_vector_load_store>

float bar(half *a) {
  return cl::vload_half< 1 >(0, a);
}

kernel void foo(cl::global_ptr<half> pg) { //ok: a global pointer
                                           // passed from the host
    int offset = 1;

    half *ptr = pg.get() + offset; //ok: half pointer arithmetic
    float b = bar(ptr);

    if(b < *ptr) { //not allowed: it is only supported if cl_khr_fp16
                   // extension is enabled
      //...
    }
}
```

The half scalar data type is required to be supported as a data storage format. Vector data load and store functions (described in the *Vector Data Load and Store Functions* section) must be supported.

cl_khr_fp16 extension

This extension adds support for half scalar and vector types as built-in types that can be used for arithmetic operations, conversions etc. An application that wants to use half and halfn types will need to specify the -cl-fp16-enable compiler option (see the *Double and half-precision floating-point options* section).

The OpenCL compiler accepts an h and H suffix on floating point literals, indicating the literal is typed as a half

A few valid examples:

```
#include <opencl_def>
#include <opencl_memory>

half bar(half a) {
    half b = a;
    b += 10.0h; //ok: cl_khr_fp16 extension is enabled. All arithmetic
                // operations on half built-in type are available

    return b;
}

kernel void foo(cl::global_ptr<half> pg) {
    int offset = 1;

    half *ptr = pg.get() + offset;
    half b = bar(*ptr);

    if(b < *ptr) { //ok: cl_khr_fp16 extension is enabled.
                   // All comparison operations are available
      //...
    }
}
```

Hexadecimal floating point literals

Hexadecimal floating point literals are supported in OpenCL C++.

```
float f = 0x1.fffffep127f
double d = 0x1.fffffffffffffp1023;
half h = 0x1.ffcp15h;
```

2.1.2. Built-in Vector Data Types

Supported Vector Data Types

The `bool`, `char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `half`, `float` and `double` vector data types are supported. The vector data type is defined with the type name i.e. `bool`, `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `half`, `float` or `double` 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.

Table 4. Device Built-in Vector Data Types

Type	Description
`booln`	A vector of *n* boolean values.
`charn`	A vector of *n* 8-bit signed two's complement integer values.

Type	Description
`ucharn`	A vector of *n* 8-bit unsigned integer values.
`shortn`	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.
`halfn`	A vector of *n* 16-bit floating-point values.
`floatn`	A vector of *n* 32-bit floating-point values.
`doublen`	A vector of *n* 64-bit floating-point values.

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:

Table 5. Host Built-in Vector Data Types

Type in OpenCL Language	API type for application
`booln`	n/a, i.e., there is no corresponding `cl_booln` type.
`charn`	`cl_charn`
`ucharn`	`cl_ucharn`
`shortn`	`cl_shortn`
`ushortn`	`cl_ushortn`
`intn`	`cl_intn`
`uintn`	`cl_uintn`
`longn`	`cl_longn`
`ulongn`	`cl_ulongn`
`halfn`	`cl_halfn`
`floatn`	`cl_floatn`
`doublen`	`cl_doublen`

The `halfn` vector data type is required to be supported as a data storage format. Vector data load and store functions (described in the *Vector Data Load and Store Functions* section) must be supported.

Support for the `doublen` vector data type is optional.

Vector Changes to C++14 standard

1. Vector types are classified as fundamental (*[ISO/IEC 14882:2014: basic.fundamental, ch. 3.9.1]*)

and literal types

> A vector type behave similarly to a trivially destructible class with all data members of literal type and all of its constructors defined as constexpr constructors

2. Abbreviating vector type as Tn, T is called the component type of a vector. The numerical value n specifies number of components in a vector. Device built-in vector data types table specifies supported vector types.

   A vector type which component type is *integral type* is called *integral vector type*. A vector type which component is *floating-point type* is called *floating-point vector type*.

   ```
   float8 a; // component type: float, number of components: 8
   uint16 b; // component type: uint, number of components: 16
   ```

3. An *integral vector type* can be used as type of value of non-type template-parameter. The change is introduced by following changes in C++ specification:

   - *[ISO/IEC 14882:2014: temp.param, ch. 14.1 (4, 4.1)]* Template parameters: A non-type template-parameter shall have one of the following (optionally cv-qualified) types:
     - integral, integral vector or enumeration type,
     - integral, integral vector or enumeration type,
     - […]
   - *[ISO/IEC 14882:2014: temp.param, ch. 14.1 (7)]* Template parameters: A non-type *template-parameter* shall not be declared to have floating point, floating-point vector, class, or void type.
   - *[ISO/IEC 14882:2014: temp.type, ch. 14.4 (1, 1.3)]* Type equivalence: Two *template-ids* refer to the same class, function, or variable if
     - […]
     - their corresponding non-type template arguments of integral, integral vector or enumeration type have identical values and
     - […]
   - *[ISO/IEC 14882:2014: temp.res, ch. 14.6 (8, 8.3, 8.3.1)]* Name resolution: […] If the interpretation of such a construct in the hypothetical instantiation is different from the interpretation of the corresponding construct
     - integral, integral vector or enumeration type, in any actual instantiation of the template, the program is ill-formed; no diagnostic is required. This can happen in situations including the following:
     - […]
     - constant expression evaluation (5.20) within the template instantiation uses
       - the value of a const object of integral, integral vector or unscoped enumeration type

or

   - […]

- […]

Vector Component Access

1. The components of vector type can be accessed using swizzle expression. The syntax of a swizzle expression is similar to syntax used in class member access expression *[ISO/IEC 14882:2014: expr.ref, ch. 5.2.5]*: The swizzle expression is a postfix expression formed with a postfix expression followed by a dot `.` or an arrow `->` and then followed by an *vector-swizzle-selector*. The postfix expression before the dot or arrow is evaluated. The result of that evaluation, together with the *vector-swizzle-selector*, determines the result of the entire postfix expression.

```
float4 v1 = float4(1.0f, 2.0f, 3.0f, 4.0f);
float4 *pv1 = &v1;

float2 v2 = v1.xz; // v1.xz is a swizzle expression
float3 v3 = pv1->s321; // pv1->s321 is a swizzle expression
                       // equivalent to (*pv1).s321
(*pv1).rgb = float3(0.0f, 0.5f, 1.0f); // (*pv1).rgb is a swizzle expression
pv1->lo.hi = 0.0f; // pv1->lo and pv1->lo.hi are swizzle
                   // expressions
```

2. For the first option (dot) the first expression shall have vector type or be a swizzle expression which results in vector-swizzle of vector type. For the second option (arrow) the first expression shall have pointer to vector type. The expression `E1->E2` is converted to the equivalent form `(*(E1)).E2`; the remainder of Vector Component Access will address only the first option (dot).

`(*(E1))` is lvalue. In either case, the *vector-swizzle-selector* shall name a vector component selection of a swizzle.

```
uint8 v1 = uint8(10, 11, 12, 13, 14, 15, 16, 17);

uint4 v2 = v1.s7301; // correct
uint3 v3 = (&v1)->s246; // correct
uint4 v4 = v1->s0123; // ill-formed: v1 is not a pointer to
                      //             vector type

uint8 *pv1 = &v1;

uint2 v5 = pv1->S13; // correct
uint2 v6 = (*pv1).s0745.even; // correct
uint4 v7 = pv1.odd; // ill-formed: pv1 is not vector or
                    // vector-swizzle
```

3. Abbreviating *postfix-expression.vector-swizzle-selector* as `E1.E2`, `E1` is called the vector expression. The type and value category of `E1.E2` are determined as follows. In the remainder of Vector Component Access, *cq* represents either `const` or the absence of `const` and *vq* represents either `volatile` or the absence of `volatile`. cv represents an arbitrary set of cv-qualifiers, as defined in *[ISO/IEC 14882:2014: basic.type.qualifier, ch. 3.9.3]* .

4. *vector-swizzle-selector* is subset of *identifier* with following syntax:

   vector-swizzle-selector:

   - *vector-swizzle-xyzw-selector*:
     - *vector-swizzle-xyzw-selector-value*
     - *vector-swizzle-xyzw-selector vector-swizzle-xyzw-selector-value*
   - *vector-swizzle-rgba-selector*:
     - *vector-swizzle-rgba-selector-value*
     - *vector-swizzle-rgba-selector vector-swizzle-rgba-selector-value*
   - *vector-swizzle-special-selector*:
     - `hi`
     - `lo`
     - `even`
     - `odd`
   - *vector-swizzle-num-selector*:
     - `s` *vector-swizzle-num-selector-values*
     - `S` *vector-swizzle-num-selector-values*

   vector-swizzle-num-selector-values:

   - *vector-swizzle-num-selector-value*
   - *vector-swizzle-num-selector-values vector-swizzle-num-selector-value*

   vector-swizzle-xyzw-selector-value: one of `x y z w`

   vector-swizzle-rgba-selector-value: one of `r g b a`

   vector-swizzle-num-selector-value: one of `0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`

   with following restrictions:

   - *vector-swizzle-selector* in a form of *vector-swizzle-special-selector* shall only be used with vector expression with at least 2 components.
   - *vector-swizzle-selector* shall not select components beyond those available in vector expression.

   Selector values and their corresponding components in swizzle table describes relation between selector value and components.

    - *vector-swizzle-selector* shall have swizzle size of 1, 2, 3, 4, 8 or 16.

> Result from the swizzle expression shall be either of scalar or of valid vector type.

If *vector-swizzle-selector* does not meet requirements, the swizzle expression is ill-formed.

```
int2 v2;
int3 v3;
int4 v4;
int8 v8;
int16 v16;

v4.xyz = int3(1, 2, 3); // correct: xyz selector
v4.baS01 = v8.lo; // ill-formed: baS01 is mix of rgba
                  // and numerical selectors
v3.rx = int2(20, 7); // ill-formed: mix of rgba and
                     // xyzw selectors

int v2c1 = v2.z; // correct: xyzw selector
int v3c1 = v3.b; // correct: rgba selector
int2 v4c1 = v4.ww; // correct: xyzw selector
int3 v8c1 = v8.xyz; // ill-formed: xyzw and rgba selectors
                    // are not allowed on vector expressions
                    // with more than 4 components
int2 v8c2 = v8.hi.xyz; // correct: xyzw selector on vector
                       // expression v8.hi (vector-swizzle
                       // of int4 type)

int2 v3c2 = v3.odd; // correct: special selector
int2 v3c2 = v3.x.even; // ill-formed: #1 vector expression
                       // is invalid (vector swizzle of
                       // scalar type)
                       // #2 special selector cannot be
                       // used with less than 2 components

v3.x = 1; // correct: xyzw selector
v3.w = 2; // ill-formed: there is no "w" component in int3
v2.gb = v4.hi; // ill-formed: there is no "b" component in int2
v8.S7890 = v4; // ill-formed: int8 allows numerical selector
               // in range 0-7

auto v16c1 = v16.s012; // correct: numerical selector
auto v16c2 = v16.s467899; // ill-formed: swizzle expression
                          // has not allowed size
                          // (there is no int6 type)

int16 vv1 = int16(v16.S98aabb01, v2, v2.gr, v3.xxxx); // correct
int16 vv2 = int16(v16.S98aabb0123, v2.gr, v3.xxxx);
                           // ill-formed:
                           // although it sums up to 16
                           // components the
                           // S98aabb0123 selector has invalid
                           // swizzle size (there is no int10)
```

5. *vector-swizzle-selector*, in a form of *vector-swizzle-xyzw-selector*, *vector-swizzle-rgba-selector* or *vector-swizzle-num-selector* can specify multiple values. Each value selects single component.

Values in a selector can be repeated and specified in any order. A number of values in a selector including repeated values is called the swizzle size.

Table 6. Selector values and their corresponding components in swizzle

Selector	Selector value	Selected component	Required number of components in vector expression
vector-swizzle-xyzw-selector	`x`	1st	2, 3 or 4
vector-swizzle-xyzw-selector	`y`	2nd	2, 3 or 4
vector-swizzle-xyzw-selector	`z`	3rd	3 or 4
vector-swizzle-xyzw-selector	`w`	4th	4
vector-swizzle-rgba-selector	`r`	1st	2, 3 or 4
vector-swizzle-rgba-selector	`g`	2nd	2, 3 or 4
vector-swizzle-rgba-selector	`b`	3rd	3 or 4
vector-swizzle-rgba-selector	`a`	4th	4
vector-swizzle-num-selector	`0`	1st	2, 3, 4, 8 or 16
vector-swizzle-num-selector	`1`	2nd	2, 3, 4, 8 or 16
vector-swizzle-num-selector	`2`	3rd	3, 4, 8 or 16
vector-swizzle-num-selector	`3`	4th	4, 8 or 16
vector-swizzle-num-selector	`4`	5th	8 or 16
vector-swizzle-num-selector	`5`	6th	8 or 16
vector-swizzle-num-selector	`6`	7th	8 or 16
vector-swizzle-num-selector	`7`	8th	8 or 16
vector-swizzle-num-selector	`8`	9th	16
vector-swizzle-num-selector	`9`	10th	16

Selector	**Selector value**	**Selected component**	**Required number of components in vector expression**
vector-swizzle-num-selector	`a` or `A`	11th	16
vector-swizzle-num-selector	`b` or `B`	12th	16
vector-swizzle-num-selector	`c` or `C`	13th	16
vector-swizzle-num-selector	`d` or `D`	14th	16
vector-swizzle-num-selector	`e` or `E`	15th	16
vector-swizzle-num-selector	`f` or `F`	16th	16

6. *vector-swizzle-selector* in a form of *vector-swizzle-special-selector* shall select:

   - if number of components in vector expression is 3, the same components as if number of components of the vector expression was 4 and the 4-th component was undefined.

     If 4-th component is read, the returned value is undefined; all writes to 4-th component shall be discarded.

   - otherwise, half of components of *vector expression* with
     - `hi` - highest numerical selector values in ascending order (higher half of the vector)
     - `lo` - lowest numerical selector values in ascending order (lower half of the vector)
     - `even` - even numerical selector values in ascending order
     - `odd` - odd numerical selector values in ascending order

The following Special selector values table describes special selector values and their numerical equivalents.

Table 7. Special selector values

Number of components in vector expression	**Selector value**	**Equivalent numerical selector**	**Number of components in result vector swizzle (swizzle size)**
2	`hi`	`s1`	1
3	`hi`	`s2?` [3]	2
4	`hi`	`s23`	2
8	`hi`	`s4567`	4
16	`hi`	`s89abcdef`	8
2	`lo`	`s0`	1

Number of components in vector expression	Selector value	Equivalent numerical selector	Number of components in result vector swizzle (swizzle size)
3	`lo`	`s01`	2
4	`lo`	`s01`	2
8	`lo`	`s0123`	4
16	`lo`	`s01234567`	8
2	`even`	`s0`	1
3	`even`	`s02`	2
4	`even`	`s02`	2
8	`even`	`s0246`	4
16	`even`	`s02468ace`	8
2	`odd`	`s1`	1
3	`odd`	`s1?` [3]	2
4	`odd`	`s13`	2
8	`odd`	`s1357`	4
16	`odd`	`s13579bdf`	8

```
float8 v = float8(1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f, 8.0f);

auto vv1 = v.hi;   // vv1 = float4(5, 6, 7, 8)
auto vv2 = v.lo;  // vv2 = float4(1, 2, 3, 4)
auto vv3 = v.even; // equivalent of v.s0246; vv3 = float4(1, 3, 5, 7)
auto vv4 = v.odd;  // equivalent of v.s1357; vv4 = float4(2, 4, 6, 8)

auto vv5 = v.odd.even; // vv5 = float2(2, 6)

int3 sv = int3(10, 20, 30);

// ? means undefined value
auto svv1 = sv.hi;  // svv1 = int2(30, ?)
auto svv2 = sv.odd; // svv2 = int2(20, ?)

sv.hi = int2(-123, 456); // write to 4-th channel in sv is discarded;
                         // equivalent of sv.s2 = int2(-123, 456).s0
```

7. The value of a swizzle expression `E1.E2` is *vector-swizzle*. The expression designates group of components of the object designated by expression `E1`. Selector `E2` specifies which components are designated, how many times and in which order.

   Assuming that in the type of a vector expression `E1` is `cv Tn` where `T` denotes type of components and `n` their number in vector type, the resulting *vector-swizzle* shall have:

- scalar type `cv T` if it is result of a swizzle expression with swizzle size of one or
- vector type `cv Tm` if it is result of a swizzle expression with swizzle size of two or more.

> `m` is a swizzle size.

If `E1` is an lvalue, then `E1.E2` is an lvalue; if `E1` is an xvalue, then `E1.E2` is an xvalue; otherwise, it is a prvalue.

```
long2 v;
const long2  pv = &v;

auto vc1 = pv->x; // pv->x is lvalue vector-swizzle of
                  // scalar type: const long
auto vc2 = pv->rg; // pv->rg is lvalue vector-swizzle of
                   // vector type: const long2

auto  vc3 = uchar4(1).xxy; // uchar4(1).xxy is prvalue
                           // vector-swizzle
                           // of vector type: uchar3

v.x = long2(1, 2); // ill-formed: cannot assign prvalue of long2
                   // to lvalue vector-swizzle of
                   // scalar type: long - types do not
                   // match
```

8. A *vector-swizzle* with vector type `T` shall have the same number of components as number of components of `T`. Each component of the vector-swizzle refers to component from `E1` designated by corresponding value specified in selector `E2`, assuming that `E1.E2` is swizzle expression used to create the *vector-swizzle*.

   > First component refers to component from `E1` selected by first value in selector `E2`, second - by second value and so on.

   A *vector-swizzle* with scalar type `T` shall behave as value of `T` and refer to component from `E1` designated by `E2`'s value, assuming `E1.E2` is swizzle expression used to create the *vector-swizzle*.

   > It is similar to reference bounded to value of selected component from `E1`.

9. A *vector-swizzle* shall have scalar or vector type. The address-of operator `&` shall not be applied to *vector-swizzle*, so there are no pointers to *vector-swizzles*. A non-const reference shall not be bound to *vector-swizzle*.

   > If the initializer for a reference of type `const T&` is lvalue that refers to vector-swizzle, the reference is bound to a temporary initialized to hold the value of the vector-swizzle; the reference is not bound to the vector-swizzle directly.

   There is no declarator for *vector-swizzle*.

Any variable, member or type declaration shall not involve vector-swizzle; vector-swizzle cannot be stored.

An *alignment-specifier* shall not be applied to *vector-swizzle*.

```
float4 v;

auto pv1 = &v; // correct: pv1 points to v
auto pv2 = &v.xy; // ill-formed: address-of operator & is not
                  // allowed on vector-swizzle

const auto &rv1 = v.xx; // correct: refers to temporary value of
                        // float2 type initialized with
                        // value of vector-swizzle
float2 &rv2 = v.xy; // ill-formed: binding to non-const reference
                    // is not allowed
```

10. A result *vector-swizzle* from swizzle expression `E1.E2` is modifiable if:

    - Vector expression `E1` is modifiable lvalue and
    - Each component selected by *vector-swizzle-selector* `E2` is selected at most once.

    Expression which modifies unmodifiable *vector-swizzle* is ill-formed.

    Changes applied to modifiable *vector-swizzle* are applied to components of `E1` referred by the *vector-swizzle* or by its components.

```
char4 v;
const char4  cv;

v.yx = char2(33, 45); // correct
v.zzwx = cv; // ill-formed: v.zzwx is not modifiable
             // (repeated components)
cv.zxy = char3(1); // ill-formed: cv.zxy is not modifiable
                   // (cv is const)
```

11. A prvalue for *vector-swizzle* of `T` type can be converted to a prvalue of `T` type.

    This conversion is called *swizzle-to-vector* conversion. *swizzle-to-vector* conversion shall be applied if necessary in all contexts where lvalue-to-rvalue conversions are allowed.

    swizzle-to-vector conversion shall be applied after lvalue-to-rvalue conversions and before any arithmetic conversions.

12. A glvalue *vector-swizzle* of scalar or vector type `T` can be used in all expressions where glvalue of type `T` can be used except those which do not meet requirements and restrictions for *vector-swizzle*.

For example the address-of operator `&` and binding to non-const reference are one of them.

13. A swizzle expression `E1.E2` where `E2` selects all components of vector expression `E1` in order of their numerical selector values is called identity swizzle.

    Components selected in `E2` are not repeated.

14. Additional changes to C++ specification:

    ◦ *[ISO/IEC 14882:2014: expr.static.cast, ch. 5.2.9 (3)]* static_cast: If value is not a bit-field or a *vector-swizzle*, [...]; if value is a *vector-swizzle*, the *lvalue-to-rvalue* conversion and *swizzle-to-vector* conversion are applied to the *vector-swizzle* and the resulting prvalue is used as the expression of the `static_cast` for the remainder of this section; otherwise, [...]

    ◦ *[ISO/IEC 14882:2014: expr.unary.op, ch. 5.3.1 (5)]* Unary operators: [...] The operand of `&` shall not be a bit-field or a *vector-swizzle*.

    ◦ *[ISO/IEC 14882:2014: expr.pre.incr, ch. 5.3.2 (1)]* Increment and decrement: The result is the updated operand; it is an lvalue, and it is a bit-field or a *vector-swizzle* if the operand is respectively a bit-field or a *vector-swizzle*.

    ◦ *[ISO/IEC 14882:2014: expr.sizeof, ch. 5.3.3 (2)]* Sizeof: [...] When applied to a *vector-swizzle* which has type `T`, the result is the same as result from `sizeof(T)`.

    ◦ *[ISO/IEC 14882:2014: expr.cond, ch. 5.16 (2.1)]* Conditional operator: - [...] The conditional-expression is a bit-field or a *vector-swizzle* if that operand is respectively a bit-field or a *vector-swizzle*.

    ◦ *[ISO/IEC 14882:2014: expr.cond, ch. 5.16 (4)]* Conditional operator: If the second and third operands are glvalues of the same value category and have the same type, the result is of that type and value category and it is a bit-field if the second or the third operand is a bit-field, or if both are bit-fields. The result is also a *vector-swizzle* if the second or the third operand is a *vector-swizzle*, or if both are *vector-swizzles*.

      An operand is converted to vector-swizzle if required by applying identity swizzle expression to it.

- *[ISO/IEC 14882:2014: expr.ass, ch. 5.18 (1)]* Assignment and compound assignment operators: The result in all cases is a bit-field or a *vector-swizzle* if the left operand is respectively a bit-field or a *vector-swizzle*.

- *[ISO/IEC 14882:2014: expr.comma, ch. 5.19 (1)]* Comma operator: The type and value of the result are the type and value of the right operand; the result is of the same value category as its right operand, and is a bit-field if its right operand is a glvalue and a bit-field, and is a *vector-swizzle* its right operand is a glvalue and a *vector-swizzle*.

- *[ISO/IEC 14882:2014: dcl.type.simple, ch. 7.1.6.2 (4, 4.1)]* Simple type specifiers: For an expression e, the type denoted by decltype(e) is defined as follows:

  ▪ if e is an unparenthesized id-expression or an unparenthesized class member access (5.2.5) or unparenthesized swizzle expression, `decltype(e)` is the type of the entity named by e. If there is no such entity, or if e names a set of overloaded functions, the program is ill-formed.

Vector Constructors

Vector constructors are defined to initialize a vector data type from a list of scalar or vectors. The forms of the constructors 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 )

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-to-vector conversion 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(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

int4 i = (int4)(1, 2, 3, 4); // warning, vector literals (from OpenCL C) are
                             // not part of OpenCL C++,
                             // this expression will be evaluated to (int4)4,
                             // and i will be (4, 4, 4, 4)
```

Vector Types and 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, per the *Implicit Type Conversions* section.

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.

2.1.3. 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 C++ 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 `vloadn`, `vload_halfn`, `vstoren`, and `vstore_halfn` functions defined in the *Vector Data Load and Store Functions* section. 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.

2.2. Keywords

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

- Names reserved as keywords by C++14.
- OpenCL C++ data types defined in Device built-in scalar data types and Device built-in vector data types tables.
- Function qualifiers: `__kernel` and `kernel`.
- Access qualifiers: `__read_only`, `read_only`, `\__write_only`, `write_only`, `__read_write` and `read_write`.

2.3. Implicit Type Conversions

Implicit conversions between scalar built-in types defined in Device built-in scalar data types table (except `void`) 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.

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. If conversion from a scalar type to the element type used by the vector result in truncation or precision loss, the program is ill-formed, with the exception that:

- if scalar value is prvalue of literal type and the value is representable as the element type, the conversion should take place without error (warnings may be generated in this case).

Implicit conversions between built-in vector data types are disallowed. Explicit conversions described in the *Conversions Library* section must be used instead.

Implicit conversions for pointer types follow the rules described in the C++14 specification.

2.4. Expressions

All expressions behave as described in *[ISO/IEC 14882:2014: expr, ch. 5]* with the the restrictions described in the *OpenCL C++ Restrictions* section and the following changes:

1. All built-in operators have their vector counterparts.
2. All built-in vector operations, apart from conditional operator, are performed component-wise.

Conditional operator logical-or-expression cannot be of vector type.

3. Built in operators taking two vectors require that vectors have the same number of components, otherwise expression is ill-formed.
4. Vector swizzle operations meet extra requirements and restrictions described in the *Vector Component Access* section.
5. Implicit and explicit casts between vector types are not legal. The conversion between vector types can be done only using `convert_cast` from the *Conversions Library* section.

   Examples:

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

   float4 f;
   int4   i = static_cast<int4>(f); // not allowed

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

6. Implicit and explicit casts from scalar to vector types are supported.
7. All built-in 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:
   a. The two operands are scalars. In this case, the operation is applied, resulting in a scalar.
   b. 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.
   c. The two operands are vectors of the same type. In this case, the operation is done component-wise resulting in the same size vector.
8. The built-in relational and equality operators equal (`==`), not equal (`!=`), greater than (`>`), greater than or equal (`>=`), less than (`<`), and less than or equal (`<=`) operate on scalar and vector types. All relational and equality operators result in a boolean (scalar or vector) type. After operand type conversion, the following cases are valid:
   a. The two operands are scalars. In this case, the operation is applied, resulting in a boolean scalar.
   b. 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 boolean vector.
   c. The two operands are vectors of the same type. In this case, the operation is done component-wise resulting in the same size boolean vector.

9. The built-in bitwise operators and (`&`), or (`|`), exclusive or (`^`), 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.

10. The built-in logical operators and (`&&`), or (`||`) operate on all scalar and vector built-in types. For scalar built-in types the logical operator and (`&&`) will only evaluate the right hand operand if the left hand operand compares unequal to `false`. For scalar built-in types the logical operator or (`||`) will only evaluate the right hand operand if the left hand operand compares equal to `false`. 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 result is a scalar or vector boolean.

11. The built-in 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 or vector boolean.

12. The built-in conditional operator (`?:`) described in *[ISO/IEC 14882:2014: expr, ch. 5.2]* operates on three expressions (`exp1 ? exp2 : exp3`). This operator evaluates the first expression `exp1`, which must be a scalar boolean result. If the result is *true* it selects to evaluate the second expression, otherwise it selects to evaluate the third expression. The second and third expressions can be any type, as long their types match, or there is a conversion in the *Implicit Type Conversions* section 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.

13. The built-in shift operators are supported for built-in vector 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 `log2(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. The vacated bits are filled with zeros. The result of `E1 >> E2` is `E1` right-shifted by `log2(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.

2.5. Address Spaces

The OpenCL C++ kernel language doesn't introduce any explicit named address spaces, but they are implemented as part of the standard library described in the *Address Spaces Library* section. There are 4 types of memory supported by all OpenCL devices: global, local, private and constant. The developers should be aware of them and know their limitations.

2.5.1. Implicit Storage Classes

The OpenCL C++ compiler can deduce an address space based on the scope where an object is declared:

- If a variable is declared in program scope, with `static` or `extern` specifier and the standard library storage class (see the *Address Spaces Library* section) is not used, the variable is allocated in the global memory of a device.
- If a variable is declared in function scope, without `static` specifier and the standard library storage class (see the *Address Spaces Library* section) is not used, the variable is allocated in the private memory of a device.

2.5.2. Memory Pools

Global

The variables are allocated from the global memory pool if they meet the criteria described in the *Implicit Storage Classes* section for the implicit global storage class or they are declared using explicit global storage class from the standard library (see the *global class* section).

The global memory objects can be:

- Passed by pointer or reference to a kernel from the host. In such case the host manages their visibility, lifetime and a type of allocation.
- Declared in the program source (`static`, `extern` and program scope global variables). In such case they are:
  - the coarse-grained SVM allocations that can be usable by multiple kernels on the same device safely
  - not shared across devices
  - not accessible from the host
  - their lifetime is the same as a program

The non-trivial constructors and destructors are supported with limitations described in the *Memory initialization* section.

The constructors of objects in global memory are executed before the first kernel execution in the program. The destructors executed at program release time.

The additional restrictions may apply if the explicit global storage class is used. Please refer to the *Restrictions* section for more details.

Local

The local variables can be only allocated in a program using the explicit local storage class from the standard library (see the *local class* section). This type of memory is allocated for each work-group executing the kernel and exist only for the lifetime of the work-group executing the kernel.

The non-trivial constructors and destructors are supported with limitations described in the *Memory initialization* section.

The constructors of objects in local memory are executed by one work-item before the kernel body execution. The destructors are executed by one work-item after the kernel body execution.

initialization of local variables can cause performance degradation.

The additional restrictions may apply if the explicit local storage class is used. Please refer to the *Restrictions* section for more details.

Private

The variables are allocated from the private memory pool if they meet the criteria described in Implicit Storage Classes for the implicit private storage class or they were declared using explicit private storage class from the standard library (see the *priv class* section).

The non-trivial constructors and destructors are supported.

The additional restrictions may apply if the explicit priv storage class is used. Please refer to the *Restrictions* section for more details.

Constant

The constant variables can be only allocated in a program using the explicit constant storage class from the standard library (see the *constant class* section). The variables declared using the `constant<T>` class refer to memory objects allocated from the global memory pool 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.

The constant objects must be constructible at compile time, they cannot have any user defined constructors, destructors, methods and operators. Otherwise behavior is undefined.

The additional restrictions may apply if the explicit constant storage class is used. Please refer to the *Restrictions* section for more details.

2.5.3. Pointers and references

All C++ pointers and references point to an object in the unnamed/generic address space if the explicit address space pointer classes are not used. The explicit address space pointer classes are implemented as a part of the standard library and they are described in the *Explicit address space* *pointer classes* section.

2.5.4. Memory initialization

Table 8. Supported memory initializers

Storage memory (address space)	**Scope type**	**Initialization type**		
		uninitialized (no constructor or trivial default constructor) **AND** trivial destructor	initialized by constant expression **AND** trivial destructor	custom initializer **OR** custom destructor
local	program	supported (not zero-pre-init)	not supported	not supported
	kernel	supported Variables are not zero-pre-initialized. Optional zero-pre-initialization possible using switch: -cl-zero -init-local-mem-vars	supported Variables are not zero-pre-initialized. Materialize temporary expressions are not supported. Optional zero-pre-initialization possible using switch: -cl-zero -init-local-mem-vars	supported Variables are not zero-pre-initialized. Materialize temporary expressions are not supported. Optional zero-pre-initialization possible using switch: -cl-zero -init-local-mem-vars
	local (non-kernel)	not supported	not supported	not supported
	class (static data member)	supported Variables are not zero-pre-initialized.	not supported	not supported

Storage memory (address space)	**Scope type**	**Initialization type**		
global	program	supported Variables are zero-pre-initialized.	supported Variables are zero or constexpr-pre-initialized.	supported
	kernel / local	supported Variables are zero-pre-initialized.	supported Variables are zero or constexpr-pre-initialized.	not supported
	class (static data member)	supported Variables are zero-pre-initialized.	supported Variables are zero or constexpr-pre-initialized.	not supported
constant	(any)	supported Variables are zero-pre-initialized.	supported Variables are zero or constexpr-pre-initialized.	not supported
private	(any)	supported	supported	supported

2.6. Kernel Functions

2.6.1. Function Qualifiers

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 enqueued by the host or on the device.

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

2.6.2. Restrictions

Kernel Function Restrictions

- Kernel functions are implicitly declared as `extern "C"`.
- Kernel functions cannot be overloaded.
- Kernel functions cannot be template functions.

- Kernel functions cannot be called by other kernel functions.
- Kernel functions cannot have parameters specified with default values.
- Kernel functions must have the return type `void`.
- Kernel functions cannot be called `main`.

Kernel Parameter Restrictions

The OpenCL host compiler and the OpenCL C++ kernel language device compiler can have different requirements for i.e. type sizes, data packing and alignment, etc., therefore the kernel parameters must meet the following requirements:

- Types passed by pointer or reference must be standard layout types.
- Types passed by value must be POD types.
- Types cannot be declared with the built-in `bool` scalar type, vector type or a class that contain `bool` scalar or vector type fields.
- Types cannot be structures or classes with bit field members.
- Marker types must be passed by value (see the *Marker Types* section).
- `global`, `constant`, and `local` storage classes can be passed only by reference or pointer. More details in the *Explicit address space storage classes* section.
- Pointers and references must point to one of the following address spaces: global, local or constant.

2.7. Preprocessor Directives and Macros

The preprocessing directives defined by the C++14 specification (*section 16*) are supported.

The `#pragma` directive is described as:

```
#pragma pp-tokensopt 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:

```
#pragma OPENCL FP_CONTRACT on-off-switch // on-off-switch: one of ON OFF DEFAULT

#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_CPP_VERSION__` substitutes an integer reflecting the OpenCL C++ version specified when compiling the OpenCL C++ program. The version of OpenCL C++ described in this document will have `__OPENCL_CPP_VERSION__` substitute the integer `100`.

The macro names defined by the C++14 specification in *section 16* but not currently supported by OpenCL are reserved for future use.

The predefined identifier `__func__` is available.

2.8. Attribute Qualifiers

The `[[ ]]` attribute qualifier syntax allows additional attributes to be attached to types, variables, kernel functions, kernel parameters, or loops.

Some attributes change the semantics of the program and are required for program correctness. Other attributes are optional hints that may be ignored without affecting program correctness. Nevertheless, frontend compilers that compile to an intermediate representation are required to faithfully pass optional attribute hints with an intermediate representation to device compilers for further processing.

2.8.1. Optional Type Attributes

`[[ ]]` attribute syntax can be used to specify special attributes of enum, class and union types when you define such types. Two attributes are currently defined for types: `aligned`, and `packed`.

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

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

cl::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]; } [[cl::aligned(8)]];
typedef int more_aligned_int [[cl::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 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 class 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 class 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 class 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]; } [[cl::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)` 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.

cl::packed

This attribute, attached to class 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 class 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 [[cl::packed]] my_packed_struct
{
  char c;
  int  i;
  struct my_unpacked_struct s;
};
```

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

2.8.2. Optional Variable Attributes

[[ ]] syntax allows you to specify special attributes of variables or structure fields. The following attribute qualifiers are currently defined:

cl::aligned

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

```
int x [[cl::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 double-word aligned int pair, you could write:

```
struct foo { int x[2] [[cl::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] [[cl::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 class, or class 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)` will still only provide you with 8 byte alignment. See your platform-specific documentation for further information.

cl::packed

The `packed` attribute specifies that a variable or class 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] [[cl::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 */
[[cl::aligned(128)]] struct A { int i; } a;

/* b has alignment of 16 */
[[cl::aligned(16)]] struct B { double d; } [[cl::aligned(32)]] b;

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

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

2.8.3. Optional Kernel Function Attributes

The kernel qualifier can be used with the `[[ ]]` attribute syntax to declare additional information about the kernel function. The kernel function attributes must appear immediately before the kernel function to be affected.

The following attributes are supported:

cl::work_group_size_hint

The optional `[[cl::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 `[[cl::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 specialization constants (see the *Specialization Constants* section) can be used as arguments of `cl::work_group_size_hint` attribute.

cl::required_work_group_size

The optional `[[cl::required_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 specialization constants (see the *Specialization Constants* section) can be used as arguments of `cl::required_work_group_size(X, Y, Z)` attribute.

cl::required_num_sub_groups

The optional `[[cl::required_num_sub_groups(X)]]` is the number of sub-groups that must be generated by a kernel launch. To ensure that this number is created the queries mapping number of sub-groups to local size may be used. This allows the compiler to optimize the kernel based on the sub-group count and in addition allows the API to enforce correctness of kernel use to the user when concurrency of sub-groups is a requirement.

The specialization constants (see the *Specialization Constants* section) can be used as argument of `cl::required_num_sub_groups` attribute.

cl::vec_type_hint

The optional `[[cl::vec_type_hint(<type>)]]` 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 `[[cl::vec_type_hint(<type>)]]` qualifier `<type>` is one of the built-in vector types listed in Device built-in vector data types table or the constituent scalar element types. If `cl::vec_type_hint(<type>)` is not specified, the kernel is assumed to have the `[[cl::vec_type_hint(int)]]` qualifier.

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
[[cl::vec_type_hint(float4)]] kernel
void foo(cl::global_ptr<float4> p) { ... }

// autovectorize assuming double as the
// basic computation width
[[cl::vec_type_hint(double)]] kernel
void foo(cl::global_ptr<float4> p) { ... }

// autovectorize assuming int (default)
// as the basic computation width
kernel void foo(cl::global_ptr<float4> p) { ... }
```

2.8.4. Optional Kernel Parameter Attributes

The kernel parameter can be used with the `[[ ]]` attribute syntax to declare additional information about an argument passed to the kernel. The kernel parameter attributes must appear immediately before or after the kernel parameter declaration to be affected.

The following attributes are supported:

cl::max_size

This attribute can be provided with a kernel argument of type `constant_ptr<T>`, `constant<T>*`, `constant<T>&`, `local_ptr<T>`, `local<T>*`, `local<T>&`. The value of the attribute specifies the maximum size in bytes of the corresponding memory object. This size cannot exceed the limits supported by the device:

- `CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE` for the kernel arguments in constant memory
- `CL_DEVICE_LOCAL_MEM_SIZE` for the kernel arguments in local memory

The specialization constants (see the *Specialization Constants* section) can be used as argument of `cl::max_size` attribute.

Examples:

```
#include <opencl_memory>

kernel void foo([[cl::max_size(65536)]] cl::constant_ptr<int> arg) {
  //...
}
```

2.8.5. Optional Loop Attributes

cl::unroll_hint

The and `[[cl::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 `[[cl::unroll_hint(n)]]` attribute qualifier must appear immediately before the loop to be affected.

Examples:

```
[[cl::unroll_hint(2)]]
while (*s != 0)
    *p++ = *s++;
```

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

```
[[cl::unroll_hint]]
for (int i=0; i<2; i++) {
   //...
}
```

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

```
[[cl::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 `[[cl::unroll_hint(n)]]`.

```
[[cl::unroll_hint(-1)]]
while (/* ... */) {
  //...
}
```

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

```
[[cl::unroll_hint]]
if(/* ... */) {
  //...
}
```

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

```
kernel void my_kernel(/* ... */) {
  int x;
  [[cl::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.

cl::ivdep

The `[[cl::ivdep]]` (ignore vector dependencies) attribute qualifier is a hint to the compiler and may appear in loops to indicate that the compiler may assume there are no memory dependencies across loop iterations in order to autovectorize consecutive iterations of the loop. This attribute qualifier may appear in one of the following forms:

```
[[cl::ivdep]]
[[cl::ivdep(len)]]
```

If the parameter `len` is specified, it is used to specify the maximum number of consecutive iterations without loop-carried dependencies. `len` is a lower bound on the distance of any loop-carried dependence, and it applies to arbitrary alignment. For example, any 4 consecutive iterations can be vectorized with `cl::ivdep(4)`. The `len` parameter must be a positive integer. The final decision whether to autovectorize the complete loop may be subject to other compiler heuristics as well as flags e.g., *-cl-fast-relaxed-math* to ignore non-associated operations.

Examples:

```
[[cl::ivdep]]
for (int i=0; i<N; i++) {
    C[i+offset] = A[i+offset] * B[i+offset];
}
```

In the example above, assuming that A and B are not restricted pointers, it is unknown if C aliases A or B. Placing the [[cl::ivdep]] attribute before the loop lets the compiler assume there are no memory dependencies across the loop iterations.

```
[[cl::ivdep(8)]]
for (int i=0; i<N; i++) {
    A[i+K] = A[i] * B[i];
}
```

In the example above, buffer A is read from and written to in the loop iterations. In each iteration, the read and write to A are to different indices. In this case it is not safe to vectorize the loop to a vector length greater than K, so the len parameter is specified with a value that is known to be not greater than any value that K may take during the execution of loop. In this example we are guaranteed (by len) that K will always be greater than or equal to 8.

Below is an example of invalid usage of [[cl::ivdep]].

```
[[cl::ivdep(-1)]]
for (int i=0; i<N; i++) {
    C[i+offset] = A[i+offset] * B[i+offset];
}
```

The above example is an invalid usage of the attribute qualifier as len is negative.

2.8.6. 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 2.2 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.

2.9. Restrictions

The following C++14 features are not supported by OpenCL C++:

* the `dynamic_cast` operator (*ISO C++ Section 5.2.7*)
* type identification (*ISO C++ Section 5.2.8*)
* recursive function calls (*ISO C++ Section 5.2.2, item 9*) unless they are a compile-time constant expression
* non-placement `new` and `delete` operators (*ISO C++ Sections 5.3.4 and 5.3.5*)
* `register` and `thread_local` storage qualifiers (*ISO C++ Section 7.1.1*)
* virtual function qualifier (*ISO C++ Section 7.1.2*)
* function pointers (*ISO C++ Sections 8.3.5 and 8.5.3*) unless they are a compile-time constant expression
* virtual functions and abstract classes (*ISO C++ Sections 10.3 and 10.4*)
* exception handling (*ISO C++ Section 15*)
* the C++ standard library (*ISO C++ Sections 17 … 30*)
* no implicit lambda to function pointer conversion (*ISO C++ Section 5.1.2, item 6*)
* variadic functions (*ISO C99 Section 7.15, Variable arguments <stdarg.h>*)
* and, like C++, OpenCL C++ does not support variable length arrays (*ISO C99, Section 6.7.5*)
* whether irreducible control flow is legal is implementation defined.

To avoid potential confusion with the above, please note the following features *are* supported in OpenCL C++:

* All variadic templates (*ISO C++ Section 14.5.3*) including variadic function templates are supported.
* Virtual inheritance (*ISO C++ Section 10.1, item 4*) is supported.

This page refers to *ISO C99* instead of *ISO C11* since the *ISO C++14* document refers to *ISO C99* in *ISO C++ Section 1.2 and Annex C*.

Chapter 3. OpenCL C++ Standard Library

OpenCL C++ does not support the C++14 standard library, but instead implements its own standard library. No OpenCL types and functions are auto-included.

3.1. OpenCL Definitions

Header <*opencl_def*> defines OpenCL scalar, vector types and macros. `cl_*` types are guaranteed to have exactly the same size as their host counterparts defined in *cl_platform.h* file.

3.1.1. Header <opencl_def> Synopsis

```
#define __OPENCL_CPP_VERSION__ 100

typedef __SIZE_TYPE__     size_t;
typedef __PTRDIFF_TYPE__  ptrdiff_t;
typedef decltype(nullptr) nullptr_t;
#define NULL              nullptr

typedef __INT8_TYPE__     int8_t     [[cl::aligned(1)]];
typedef __UINT8_TYPE__    uint8_t    [[cl::aligned(1)]];
typedef __INT16_TYPE__    int16_t    [[cl::aligned(2)]];
typedef __UINT16_TYPE__   uint16_t   [[cl::aligned(2)]];
typedef __INT32_TYPE__    int32_t    [[cl::aligned(4)]];
typedef __UINT32_TYPE__   uint32_t   [[cl::aligned(4)]];
typedef __INT64_TYPE__    int64_t    [[cl::aligned(8)]];
typedef __UINT64_TYPE__   uint64_t   [[cl::aligned(8)]];

#if   __INTPTR_WIDTH__ == 32
typedef int32_t           intptr_t;
typedef uint32_t          uintptr_t;
#elif __INTPTR_WIDTH__ == 64
typedef int64_t           intptr_t;
typedef uint64_t          uintptr_t;
#endif

namespace cl
{
using ::intptr_t;
using ::uintptr_t;
using ::ptrdiff_t;
using ::nullptr_t;
using ::size_t;
}

typedef int8_t            cl_char;
typedef uint8_t           cl_uchar;
typedef int16_t           cl_short
typedef uint16_t          cl_ushort;
typedef int32_t           cl_int;
typedef uint32_t          cl_uint;
typedef int64_t           cl_long;
typedef uint64_t          cl_ulong;

#ifdef cl_khr_fp16
typedef half              cl_half   [[aligned(2)]];
#endif
typedef float             cl_float  [[aligned(4)]];
#ifdef cl_khr_fp64
typedef double            cl_double [[aligned(8)]];
#endif

typedef implementation-defined bool2;
typedef implementation-defined bool3;
typedef implementation-defined bool4;
typedef implementation-defined bool8;
typedef implementation-defined bool16;
typedef implementation-defined char2;
typedef implementation-defined char3;
typedef implementation-defined char4;
typedef implementation-defined char8;
typedef implementation-defined char16;
typedef implementation-defined uchar2;
typedef implementation-defined uchar3;
typedef implementation-defined uchar4;
typedef implementation-defined uchar8;
typedef implementation-defined uchar16;
typedef implementation-defined short2;
typedef implementation-defined short3;
typedef implementation-defined short4;
typedef implementation-defined short8;
typedef implementation-defined short16;
typedef implementation-defined ushort2;
typedef implementation-defined ushort3;
typedef implementation-defined ushort4;
typedef implementation-defined ushort8;
typedef implementation-defined ushort16;
typedef implementation-defined int2;
typedef implementation-defined int3;
typedef implementation-defined int4;
typedef implementation-defined int8;
typedef implementation-defined int16;
typedef implementation-defined uint2;
typedef implementation-defined uint3;
typedef implementation-defined uint4;
typedef implementation-defined uint8;
typedef implementation-defined uint16;
typedef implementation-defined long2;
typedef implementation-defined long3;
typedef implementation-defined long4;
typedef implementation-defined long8;
typedef implementation-defined long16;
typedef implementation-defined ulong2;
typedef implementation-defined ulong3;
typedef implementation-defined ulong4;
typedef implementation-defined ulong8;
typedef implementation-defined ulong16;
typedef implementation-defined float2;
typedef implementation-defined float3;
typedef implementation-defined float4;
typedef implementation-defined float8;
typedef implementation-defined float16;
#ifdef cl_khr_fp16
typedef implementation-defined half2;
typedef implementation-defined half3;
typedef implementation-defined half4;
typedef implementation-defined half8;
typedef implementation-defined half16;
#endif
#ifdef cl_khr_fp64
typedef implementation-defined double2;
typedef implementation-defined double3;
typedef implementation-defined double4;
typedef implementation-defined double8;
typedef implementation-defined double16;
#endif

typedef bool2    cl_bool2;
typedef bool3    cl_bool3;
typedef bool4    cl_bool4;
typedef bool8    cl_bool8;
typedef bool16   cl_bool16;
typedef char2    cl_char2;
typedef char3    cl_char3;
typedef char4    cl_char4;
typedef char8    cl_char8;
typedef char16   cl_char16;
typedef uchar2   cl_uchar2;
typedef uchar3   cl_uchar3;
typedef uchar4   cl_uchar4;
typedef uchar8   cl_uchar8;
typedef uchar16  cl_uchar16;
typedef short2   cl_short2;
typedef short3   cl_short3;
typedef short4   cl_short4;
typedef short8   cl_short8;
typedef short16  cl_short16;
typedef ushort2  cl_ushort2;
typedef ushort3  cl_ushort3;
typedef ushort4  cl_ushort4;
typedef ushort8  cl_ushort8;
typedef ushort16 cl_ushort16;
typedef int2     cl_int2;
typedef int3     cl_int3;
typedef int4     cl_int4;
typedef int8     cl_int8;
typedef int16    cl_int16;
typedef uint2    cl_uint2;
typedef uint3    cl_uint3;
typedef uint4    cl_uint4;
typedef uint8    cl_uint8;
typedef uint16   cl_uint16;
typedef long2    cl_long2;
typedef long3    cl_long3;
typedef long4    cl_long4;
typedef long8    cl_long8;
typedef long16   cl_long16;
typedef ulong2   cl_ulong2;
typedef ulong3   cl_ulong3;
typedef ulong4   cl_ulong4;
typedef ulong8   cl_ulong8;
typedef ulong16  cl_ulong16;
typedef float2   cl_float2;
typedef float3   cl_float3;
typedef float4   cl_float4;
typedef float8   cl_float8;
typedef float16  cl_float16;
#ifdef cl_khr_fp16
typedef half2    cl_half2;
typedef half3    cl_half3;
typedef half4    cl_half4;
typedef half8    cl_half8;
typedef half16   cl_half16;
#endif
#ifdef cl_khr_fp64
typedef double2  cl_double2;
typedef double3  cl_double3;
typedef double4  cl_double4;
typedef double8  cl_double8;
typedef double16 cl_double16;
#endif
```

3.2. Conversions Library

This section describes the explicit conversion cast functions. These functions provide a full set of type conversions between supported scalar and vector data types (see the *Built-in Scalar Data Types* and the *Built-in Vector Data Types* sections) except for the following types: `size_t`, `ptrdiff_t`, `intptr_t`, `uintptr_t`, and `void`.

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 convert_cast type conversion operator that specifies a rounding mode and saturation is also provided.

3.2.1. Header <opencl_convert> Synopsis

```
namespace cl
{
enum class rounding_mode { rte, rtz, rtp, rtn };
enum class saturate { off, on };

template <class T, class U>
T convert_cast(U const& arg);
template <class T>
T convert_cast(T const& arg);

template <class T, rounding_mode rmode, class U>
T convert_cast(U const& arg);
template <class T, rounding_mode rmode>
T convert_cast(T const& arg);

template <class T, saturate smode, class U>
T convert_cast(U const& arg);
template <class T, saturate smode>
T convert_cast(T const& arg);

template <class T, rounding_mode rmode, saturate smode, class U>
T convert_cast(U const& arg);
template <class T, rounding_mode rmode, saturate smode>
T convert_cast(T const& arg);

}
```

3.2.2. Data Types

Conversions are available for the following scalar types: bool, char, uchar, short, ushort, int, uint, long, ulong, half [4], float, double, 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 the C++14 specification except for out-of-range behavior and saturated conversions which are described in the *Out-of-Range Behavior and Saturated Conversions* section below.

3.2.3. Rounding Modes

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

Table 9. Rounding Modes

Rounding Mode	Description
`rte`	Round to nearest even
`rtz`	Round toward zero
`rtp`	Round toward positive infinity
`rtn`	Round toward negative infinity

If a rounding mode is not specified, conversions to integer type use the `rtz` (round toward zero) rounding mode and conversions to floating-point type [5] uses the `rte` rounding mode.

3.2.4. 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 the C++14 specification in *chapter 4.9*. 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 saturation mode. 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 `convert_cast` operator with a saturate argument may not be used for conversions to floating-point formats.

3.2.5. Examples

Example 1

Examples of casting between two vector types with saturation.

```
#include <opencl_convert>
using namespace cl;

kernel void Foo() {
short4 s;
       // negative values clamped to 0
       ushort4 u = convert_cast<ushort4,saturate::on>(s);

// values > CHAR_MAX converted to CHAR_MAX
       // values < CHAR_MIN converted to CHAR_MIN
       char4 c = convert_cast<char4, saturate::on>(s);
}
```

Example 2

Examples of casting from float to integer vector type with saturation and rounding mode specified.

```
#include <opencl_convert>
using namespace cl;

kernel void Foo() {
       float4  f;

       // values implementation defined for
       // f > INT_MAX, f < INT_MIN or NaN
       int4    i1 = convert_cast<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_cast<int4,saturate::on>(f);

       // similar to convert_cast<int4>, except that floating-point
       // values are rounded to the nearest integer instead of
       // truncated
       int4    i3 = convert_cast<int4, rounding_mode::rte>(f);

       // similar to convert_cast<int4, saturate::on>, except that
       // floating-point values are rounded to the nearest integer
       // instead of truncated
       int4    i4 = convert_cast<int4, rounding_mode::rte,
       saturate::on>(f);
}
```

Example 3

Examples of casting from integer to float vector type.

```
#include <opencl_convert>
using namespace cl;

kernel void Foo() {
       int4    i;

       // convert ints to floats using the default rounding mode.
       float4  f1 = convert_cast<float4>(i);

       // convert ints to floats. integer values that cannot
       // be exactly represented as floats should round up to the
       // next representable float.
       float4  f2 = convert_cast<float4, rounding_mode::rtp>(i);
}
```

3.3. Reinterpreting Data Library

It is frequently necessary to reinterpret bits in a data type as another data type in OpenCL C++. 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.

3.3.1. Header <opencl_reinterpret> Synopsis

```
namespace cl
{
template <class T, class U>
T as_type(U const& arg);

}
```

3.3.2. Reinterpreting Types

All data types described in Device built-in scalar data types and Device built-in vector data types tables (except `bool` and `void`) may be also reinterpreted as another data type of the same size using the `as_type()` [6] function for scalar and 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.

For example, `as_type<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 the `as_type<T>` operator to reinterpret data to a type of a different number of bytes.

3.3.3. Examples

Example 1

Examples of reinterpreting data types using `as_type<>` function.

```
#include <opencl_reinterpret>
using namespace cl;

kernel void Foo() {
       float f = 1.0f;
       uint u = as_type<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_type<int4>(f);

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

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

       float4 f;
       // Error.  Result and operand have different sizes
       double4 g = as_type<double4>(f);

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

3.4. Address Spaces Library

Unlike OpenCL C, OpenCL C++ does not require the address space qualifiers to allocate storage from global, local and constant memory pool. The same functionality is provided using the storage and pointer classes. These new types are designed to avoid many programming issues and it is recommended to use them for the static and program scope variables even if it is not required.

3.4.1. Header <opencl_memory> Synopsis

```
namespace cl
{
enum class mem_fence
{
    local,
    global,
    image
};
```

```
inline mem_fence operator ~(mem_fence flags);
inline mem_fence operator &(mem_fence LHS, mem_fence RHS);
inline mem_fence operator |(mem_fence LHS, mem_fence RHS);
inline mem_fence operator ^(mem_fence LHS, mem_fence RHS);

// address space pointer classes
template<class T>
class global_ptr;

template<class T>
class local_ptr;

template<class T>
class private_ptr;

template<class T>
class constant_ptr;

template<class T>
using global = see 'global class' section;

template<class T>
using local = see 'local class' section;

template<class T>
using priv = see 'priv class' section;

template<class T>
using constant = see 'constant class' section;

// address space query functions
template<class T>
mem_fence get_mem_fence(T *ptr);

// address space cast functions
template<class T>
T dynamic_asptr_cast(T *ptr) noexcept;

template <class T, class U>
local_ptr<T> static_asptr_cast(local_ptr<U> const& ptr) noexcept;
template <class T, class U>
global_ptr<T> static_asptr_cast(global_ptr<U> const& ptr) noexcept;
template <class T, class U>
constant_ptr<T> static_asptr_cast(constant_ptr<U> const& ptr) noexcept;
template <class T, class U>
private_ptr<T> static_asptr_cast(private_ptr<U> const& ptr) noexcept;

template <class T, class U>
local_ptr<T> reinterpret_asptr_cast(local_ptr<U> const& ptr) noexcept;
template <class T, class U>
global_ptr<T> reinterpret_asptr_cast(global_ptr<U> const& ptr) noexcept;
```

```
template <class T, class U>
constant_ptr<T> reinterpret_asptr_cast(constant_ptr<U> const& ptr) noexcept;
template <class T, class U>
private_ptr<T> reinterpret_asptr_cast(private_ptr<U> const& ptr) noexcept;

template <class T>
T* addressof(T& t) noexcept;

}
```

3.4.2. Explicit address space storage classes

The explicit address space storage classes described in this section are designed to allocate memory in one of the named address spaces: global, local, constant or private.

global class

The variables declared using `global<T>` class refer to memory objects allocated from the global memory pool (see the *Global Memory Pool* section). The global storage class can only be used to declare variables at program, function and class scope. The variables at function and class scope must be declared with `static` specifier.

If `T` is a fundamental or an array type, the `global` class should meet the following requirements:

- no user provide default constructor
- default copy and move constructors
- default copy and move assignment operators
- address-of operators that return a generic `T` pointer (`T*`)
- conversion operators to a generic `T` lvalue reference type (`T&`)
- assignment `const T&` operator
- `ptr()` methods that return a `global_ptr<T>` pointer class

If `T` is a class type, the `global` class should provide the following interface:

- the same public interface as `T` type including constructors and assignment operators address-of operators that return a generic `T` pointer (`T*`)
- conversion operators to a generic `T` lvalue reference type (`T&`)
- `ptr()` methods that return a `global_ptr<T>` pointer class

local class

The variables declared using `local<T>` class refer to memory objects allocated from the local memory pool (see the *Local Memory Pool* section). The local storage class can only be used to declare variables at program, kernel and class scope. The variables at class scope must be declared with `static` specifier.

If `T` is a fundamental or an array type, the `local` class should meet the following requirements:

- no user provide default constructor
- default copy and move constructors
- default copy and move assignment operators
- address-of operators that return a generic `T` pointer (`T*`)
- conversion operators to a generic `T` lvalue reference type (`T&`)
- assignment `const T&` operator
- `ptr()` methods that return a `local_ptr<T>` pointer class

If `T` is a class type, the `local` class should provide the following interface:

- the same public interface as `T` type including constructors and assignment operators
- address-of operators that return a generic `T` pointer (`T*`)
- conversion operators to a generic `T` lvalue reference type (`T&`)
- `ptr()` methods that return a `local_ptr<T>` pointer class

priv class

The variables declared using the `priv<T>` class refer to memory objects allocated from the private memory pool.

The `priv` storage class cannot be used to declare variables in the program scope, with `static` specifier or `extern` specifier.

If `T` is a fundamental or an array type, the `priv` class should meet the following requirements:

- no user provide default constructor
- default copy and move constructors
- default copy and move assignment operators
- address-of operators that return a generic `T` pointer (`T*`)
- conversion operators to a generic `T` lvalue reference type (`T&`)
- assignment const `T&` operator
- `ptr()` methods that return a `private_ptr<T>` pointer class

If `T` is a class type, the `priv` class should provide the following interface:

- the same public interface as `T` type including constructors and assignment operators
- address-of operators that return a generic `T` pointer (`T*`)
- conversion operators to a generic `T` lvalue reference type (`T&`)
- `ptr()` methods that return a `private_ptr<T>` pointer class

constant class

The variables declared using the `constant<T>` class refer to memory objects allocated from the global memory pool and which are accessed inside a kernel(s) as read-only variables. The constant storage class can only be used to declare variables at program, kernel and class scope. The variables at class scope must be declared with `static` specifier.

The `T` type must meet the following requirements:

- `T` must be constructible at compile time
- `T` cannot have any user defined constructors, destructors, methods and operators

If `T` is a fundamental, array or class type, the `constant` class should meet the following requirements:

- no user provide default constructor
- default copy and move constructors
- copy and move assignment operators deleted
- address-of operators that return a `constant_ptr<T>` pointer class
- `ptr()` methods that return a `constant_ptr<T>` pointer class
- conversion operators to a constant `T` lvalue reference type (`add_constant_t<T>&`)

3.4.3. Explicit address space pointer classes

The explicit address space pointer classes are just like pointers: they can be converted to and from pointers with compatible address spaces, qualifiers and types. Assignment or casting between explicit pointer types of incompatible address spaces is illegal.

All named address spaces are incompatible with all other address spaces, but local, global and private pointers can be converted to standard C++ pointers.

global_ptr class

```
namespace cl
{
template <class T> class global_ptr
{
public:
    //types:
    typedef T element_type;
    typedef ptrdiff_t difference_type;
    typedef add_global_t<T>& reference;
    typedef const add_global_t<T>& const_reference;
    typedef add_global_t<T>* pointer;
    typedef const add_global_t<T>* const_pointer;

    //constructors:
    constexpr global_ptr() noexcept;
    explicit global_ptr(pointer p) noexcept;
    global_ptr(const global_ptr &r) noexcept;
    global_ptr(global_ptr &&r) noexcept;
    constexpr global_ptr(nullptr_t) noexcept;

    //assignment:
    global_ptr &operator=(const global_ptr &r) noexcept;
    global_ptr &operator=(global_ptr &&r) noexcept;
    global_ptr &operator=(pointer r) noexcept;
    global_ptr &operator=(nullptr_t) noexcept;

    //observers:
    add_lvalue_reference_t<add_global_t<T>> operator*() const noexcept;
    pointer operator->() const noexcept;
    pointer get() const noexcept;
    explicit operator bool() const noexcept;

    //modifiers:
    pointer release() noexcept;
    void reset(pointer p = pointer()) noexcept;
    void swap(global_ptr& r) noexcept;

    global_ptr &operator++() noexcept;
    global_ptr operator++(int) noexcept;
    global_ptr &operator--() noexcept;
    global_ptr operator--(int) noexcept;
    global_ptr &operator+=(difference_type r) noexcept;
    global_ptr &operator-=(difference_type r) noexcept;
    global_ptr operator+(difference_type r) noexcept;
    global_ptr operator-(difference_type r) noexcept;
};

template <class T> class global_ptr<T[]>
{
public:
    //types:
    typedef T element_type;
    typedef ptrdiff_t difference_type;
    typedef add_global_t<T>& reference;
    typedef const add_global_t<T>& const_reference;
    typedef add_global_t<T>* pointer;
    typedef const add_global_t<T>* const_pointer;

    //constructors:
    constexpr global_ptr() noexcept;
    explicit global_ptr(pointer p) noexcept;
    global_ptr(const global_ptr &r) noexcept;
    global_ptr(global_ptr &&r) noexcept;
    constexpr global_ptr(nullptr_t) noexcept;

    //assignment:
    global_ptr &operator=(const global_ptr &r) noexcept;
    global_ptr &operator=(global_ptr &&r) noexcept;
    global_ptr &operator=(pointer r) noexcept;
    global_ptr &operator=(nullptr_t) noexcept;

    //observers:
    reference operator[](size_t pos) const noexcept;
    pointer get() const noexcept;
    explicit operator bool() const noexcept;

    //modifiers:
    pointer release()noexcept;
    void reset(pointer p) noexcept;
    void reset(nullptr_t p = nullptr) noexcept;
    void swap(global_ptr& r) noexcept;

    global_ptr &operator++() noexcept;
    global_ptr operator++(int) noexcept;
    global_ptr &operator--() noexcept;
    global_ptr operator--(int) noexcept;
    global_ptr &operator+=(difference_type r) noexcept;
    global_ptr &operator-=(difference_type r) noexcept;
    global_ptr operator+(difference_type r) noexcept;
    global_ptr operator-(difference_type r) noexcept;
};

template<class T, class U>
bool operator==(const global_ptr<T> &a, const global_ptr<U> &b) noexcept;
template<class T, class U>
bool operator!=(const global_ptr<T> &a, const global_ptr<U> &b) noexcept;
template<class T, class U>
bool operator<(const global_ptr<T> &a, const global_ptr<U> &b) noexcept;
template<class T, class U>
bool operator>(const global_ptr<T> &a, const global_ptr<U> &b) noexcept;
template<class T, class U>
bool operator<=(const global_ptr<T> &a, const global_ptr<U> &b) noexcept;
template<class T, class U>
bool operator>=(const global_ptr<T> &a, const global_ptr<U> &b) noexcept;

template<class T>
bool operator==(const global_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator==(nullptr_t, const global_ptr<T> &x) noexcept;
template<class T>
bool operator!=(const global_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator!=(nullptr_t, const global_ptr<T> &x) noexcept;
template<class T>
bool operator<(const global_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator<(nullptr_t, const global_ptr<T> &x) noexcept;
template<class T>
bool operator>(const global_ptr<T> &x, nullptr_t) noexcept;
template<class T>
```

```
bool operator>(nullptr_t, const global_ptr<T> &x) noexcept;
template<class T>
bool operator<=(const global_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator<=(nullptr_t, const global_ptr<T> &x) noexcept;
template<class T>
bool operator>=(const global_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator>=(nullptr_t, const global_ptr<T> &x) noexcept;

template<class T>
void swap(global_ptr<T>& a, global_ptr<T>& b) noexcept;

}
```

local_ptr class

```
namespace cl
{
template <class T> class local_ptr
{
public:
    struct size_type
    {
        explicit constexpr size_type(size_t size);
        operator size_t();
    };

    //types:
    typedef T element_type;
    typedef ptrdiff_t difference_type;
    typedef add_local_t<T>& reference;
    typedef const add_local_t<T>& const_reference;
    typedef add_local_t<T>* pointer;
    typedef const add_local_t<T>* const_pointer;

    //constructors:
    constexpr local_ptr() noexcept;
    explicit local_ptr(pointer p) noexcept;
    local_ptr(const local_ptr &r) noexcept;
    local_ptr(local_ptr &&r) noexcept;
    constexpr local_ptr(nullptr_t) noexcept;

    //assignment:
    local_ptr &operator=(const local_ptr &r) noexcept;
    local_ptr &operator=(local_ptr &&r) noexcept;
    local_ptr &operator=(pointer r) noexcept;
    local_ptr &operator=(nullptr_t) noexcept;

    //observers:
```

```
    add_lvalue_reference_t<add_local_t<T>> operator*() const noexcept;
    pointer operator->() const noexcept;
    pointer get() const noexcept;
    explicit operator bool() const noexcept;

    //modifiers:
    pointer release() noexcept;
    void reset(pointer p = pointer()) noexcept;
    void swap(local_ptr& r) noexcept;

    local_ptr &operator++() noexcept;
    local_ptr operator++(int) noexcept;
    local_ptr &operator--() noexcept;
    local_ptr operator--(int) noexcept;
    local_ptr &operator+=(difference_type r) noexcept;
    local_ptr &operator-=(difference_type r) noexcept;
    local_ptr operator+(difference_type r) noexcept;
    local_ptr operator-(difference_type r) noexcept;
};

template <class T> class local_ptr<T[]>
{
public:
    //types:
    typedef T element_type;
    typedef ptrdiff_t difference_type;
    typedef add_local_t<T>& reference;
    typedef const add_local_t<T>& const_reference;
    typedef add_local_t<T>* pointer;
    typedef const add_local_t<T>* const_pointer;

    //constructors:
    constexpr local_ptr() noexcept;
    explicit local_ptr(pointer p) noexcept;
    local_ptr(const local_ptr &r) noexcept;
    local_ptr(local_ptr &&r) noexcept;
    constexpr local_ptr(nullptr_t) noexcept;

    //assignment:
    local_ptr &operator=(const local_ptr &r) noexcept;
    local_ptr &operator=(local_ptr &&r) noexcept;
    local_ptr &operator=(pointer r) noexcept;
    local_ptr &operator=(nullptr_t) noexcept;

    //observers:
    reference operator[](size_t pos) const noexcept;
    pointer get() const noexcept;
    explicit operator bool() const noexcept;

    //modifiers:
    pointer release()noexcept;
    void reset(pointer p) noexcept;
    void reset(nullptr_t p = nullptr) noexcept;
    void swap(local_ptr& r) noexcept;

    local_ptr &operator++() noexcept;
    local_ptr operator++(int) noexcept;
    local_ptr &operator--() noexcept;
    local_ptr operator--(int) noexcept;
    local_ptr &operator+=(difference_type r) noexcept;
    local_ptr &operator-=(difference_type r) noexcept;
    local_ptr operator+(difference_type r) noexcept;
    local_ptr operator-(difference_type r) noexcept;
};

template<class T, class U>
bool operator==(const local_ptr<T> &a, const local_ptr<U> &b) noexcept;
template<class T, class U>
bool operator!=(const local_ptr<T> &a, const local_ptr<U> &b) noexcept;
template<class T, class U>
bool operator<(const local_ptr<T> &a, const local_ptr<U> &b) noexcept;
template<class T, class U>
bool operator>(const local_ptr<T> &a, const local_ptr<U> &b) noexcept;
template<class T, class U>
bool operator<=(const local_ptr<T> &a, const local_ptr<U> &b) noexcept;
template<class T, class U>
bool operator>=(const local_ptr<T> &a, const local_ptr<U> &b) noexcept;

template<class T>
bool operator==(const local_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator==(nullptr_t, const local_ptr<T> &x) noexcept;
template<class T>
bool operator!=(const local_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator!=(nullptr_t, const local_ptr<T> &x) noexcept;
template<class T>
bool operator<(const local_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator<(nullptr_t, const local_ptr<T> &x) noexcept;
template<class T>
bool operator>(const local_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator>(nullptr_t, const local_ptr<T> &x) noexcept;
template<class T>
bool operator<=(const local_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator<=(nullptr_t, const local_ptr<T> &x) noexcept;
template<class T>
bool operator>=(const local_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator>=(nullptr_t, const local_ptr<T> &x) noexcept;
```

```
template<class T>
void swap(local_ptr<T>& a, local_ptr<T>& b) noexcept;

}
```

private_ptr class

```
namespace cl
{
template <class T> class private_ptr
{
public:
    //types:
    typedef T element_type;
    typedef ptrdiff_t difference_type;
    typedef add_private_t<T>& reference;
    typedef const add_private_t<T>& const_reference;
    typedef add_private_t<T>* pointer;
    typedef const add_private_t<T>* const_pointer;

    //constructors:
    constexpr private_ptr() noexcept;
    explicit private_ptr(pointer p) noexcept;
    private_ptr(const private_ptr &r) noexcept;
    private_ptr(private_ptr &&r) noexcept;
    constexpr private_ptr(nullptr_t) noexcept;

    //assignment:
    private_ptr &operator=(const private_ptr &r) noexcept;
    private_ptr &operator=(private_ptr &&r) noexcept;
    private_ptr &operator=(pointer r) noexcept;
    private_ptr &operator=(nullptr_t) noexcept;

    //observers:
    add_lvalue_reference_t<add_private_t<T>> operator*() const noexcept;
    pointer operator->() const noexcept;
    pointer get() const noexcept;
    explicit operator bool() const noexcept;

    //modifiers:
    pointer release() noexcept;
    void reset(pointer p = pointer()) noexcept;
    void swap(private_ptr& r) noexcept;

    private_ptr &operator++() noexcept;
    private_ptr operator++(int) noexcept;
    private_ptr &operator--() noexcept;
    private_ptr operator--(int) noexcept;
    private_ptr &operator+=(difference_type r) noexcept;
    private_ptr &operator-=(difference_type r) noexcept;
    private_ptr operator+(difference_type r) noexcept;
    private_ptr operator-(difference_type r) noexcept;
};

template <class T> class private_ptr<T[]> {
public:
    //types:
    typedef T element_type;
    typedef ptrdiff_t difference_type;
    typedef add_private_t<T>& reference;
    typedef const add_private_t<T>& const_reference;
    typedef add_private_t<T>* pointer;
    typedef const add_private_t<T>* const_pointer;

    //constructors:
    constexpr private_ptr() noexcept;
    explicit private_ptr(pointer p) noexcept;
    private_ptr(const private_ptr &r) noexcept;
    private_ptr(private_ptr &&r) noexcept;
    constexpr private_ptr(nullptr_t) noexcept;

    //assignment:
    private_ptr &operator=(const private_ptr &r) noexcept;
    private_ptr &operator=(private_ptr &&r) noexcept;
    private_ptr &operator=(pointer r) noexcept;
    private_ptr &operator=(nullptr_t) noexcept;

    //observers:
    reference operator[](size_t pos) const noexcept;
    pointer get() const noexcept;
    explicit operator bool() const noexcept;

    //modifiers:
    pointer release()noexcept;
    void reset(pointer p) noexcept;
    void reset(nullptr_t p = nullptr) noexcept;
    void swap(private_ptr& r) noexcept;

    private_ptr &operator++() noexcept;
    private_ptr operator++(int) noexcept;
    private_ptr &operator--() noexcept;
    private_ptr operator--(int) noexcept;
    private_ptr &operator+=(difference_type r) noexcept;
    private_ptr &operator-=(difference_type r) noexcept;
    private_ptr operator+(difference_type r) noexcept;
    private_ptr operator-(difference_type r) noexcept;
};

template<class T, class U>
bool operator==(const private_ptr<T> &a, const private_ptr<U> &b) noexcept;
```

```
template<class T, class U>
bool operator!=(const private_ptr<T> &a, const private_ptr<U> &b) noexcept;
template<class T, class U>
bool operator<(const private_ptr<T> &a, const private_ptr<U> &b) noexcept;
template<class T, class U>
bool operator>(const private_ptr<T> &a, const private_ptr<U> &b) noexcept;
template<class T, class U>
bool operator<=(const private_ptr<T> &a, const private_ptr<U> &b) noexcept;
template<class T, class U>
bool operator>=(const private_ptr<T> &a, const private_ptr<U> &b) noexcept;

template<class T>
bool operator==(const private_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator==(nullptr_t, const private_ptr<T> &x) noexcept;
template<class T>
bool operator!=(const private_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator!=(nullptr_t, const private_ptr<T> &x) noexcept;
template<class T>
bool operator<(const private_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator<(nullptr_t, const private_ptr<T> &x) noexcept;
template<class T>
bool operator>(const private_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator>(nullptr_t, const private_ptr<T> &x) noexcept;
template<class T>
bool operator<=(const private_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator<=(nullptr_t, const private_ptr<T> &x) noexcept;
template<class T>
bool operator>=(const private_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator>=(nullptr_t, const private_ptr<T> &x) noexcept;

template<class T>
void swap(private_ptr<T>& a, private_ptr<T>& b) noexcept;

}
```

constant_ptr class

```
namespace cl
{
template <class T> class constant_ptr
{
public:
    //types:
    typedef T element_type;
    typedef ptrdiff_t difference_type;
    typedef add_constant_t<T>& reference;
    typedef const add_constant_t<T>& const_reference;
    typedef add_constant_t<T>* pointer;
    typedef const add_constant_t<T>* const_pointer;

    //constructors:
    constexpr constant_ptr() noexcept;
    explicit constant_ptr(pointer p) noexcept;
    constant_ptr(const constant_ptr &r) noexcept;
    constant_ptr(constant_ptr &&r) noexcept;
    constexpr constant_ptr(nullptr_t) noexcept;

    //assignment:
    constant_ptr &operator=(const constant_ptr &r) noexcept;
    constant_ptr &operator=(constant_ptr &&r) noexcept;
    constant_ptr &operator=(pointer r) noexcept;
    constant_ptr &operator=(nullptr_t) noexcept;

    //observers:
    add_lvalue_reference_t<add_constant_t<T>> operator*() const noexcept;
    pointer operator->() const noexcept;
    pointer get() const noexcept;
    explicit operator bool() const noexcept;

    //modifiers:
    pointer release() noexcept;
    void reset(pointer p = pointer()) noexcept;
    void swap(constant_ptr& r) noexcept;

    constant_ptr &operator++() noexcept;
    constant_ptr operator++(int) noexcept;
    constant_ptr &operator--() noexcept;
    constant_ptr operator--(int) noexcept;
    constant_ptr &operator+=(difference_type r) noexcept;
    constant_ptr &operator-=(difference_type r) noexcept;
    constant_ptr operator+(difference_type r) noexcept;
    constant_ptr operator-(difference_type r) noexcept;
};

template <class T> class constant_ptr<T[]>
{
public:
    //types:
    typedef T element_type;
    typedef ptrdiff_t difference_type;
    typedef add_constant_t<T>& reference;
    typedef const add_constant_t<T>& const_reference;
    typedef add_constant_t<T>* pointer;
    typedef const add_constant_t<T>* const_pointer;
    //constructors:
    constexpr constant_ptr() noexcept;
    explicit constant_ptr(pointer p) noexcept;
    constant_ptr(const constant_ptr &r) noexcept;
    constant_ptr(constant_ptr &&r) noexcept;
    constexpr constant_ptr(nullptr_t) noexcept;

    //assignment:
    constant_ptr &operator=(const constant_ptr &r) noexcept;
    constant_ptr &operator=(constant_ptr &&r) noexcept;
    constant_ptr &operator=(pointer r) noexcept;
    constant_ptr &operator=(nullptr_t) noexcept;

    //observers:
    reference operator[](size_t pos) const noexcept;
    pointer get() const noexcept;
    explicit operator bool() const noexcept;

    //modifiers:
    pointer release()noexcept;
    void reset(pointer p) noexcept;
    void reset(nullptr_t p = nullptr) noexcept;
    void swap(constant_ptr& r) noexcept;

    constant_ptr &operator++() noexcept;
    constant_ptr operator++(int) noexcept;
    constant_ptr &operator--() noexcept;
    constant_ptr operator--(int) noexcept;
    constant_ptr &operator+=(difference_type r) noexcept;
    constant_ptr &operator-=(difference_type r) noexcept;
    constant_ptr operator+(difference_type r) noexcept;
    constant_ptr operator-(difference_type r) noexcept;
};

template<class T, class U>
bool operator==(const constant_ptr<T> &a, const constant_ptr<U> &b) noexcept;
template<class T, class U>
bool operator!=(const constant_ptr<T> &a, const constant_ptr<U> &b) noexcept;
template<class T, class U>
bool operator<(const constant_ptr<T> &a, const constant_ptr<U> &b) noexcept;
template<class T, class U>
bool operator>(const constant_ptr<T> &a, const constant_ptr<U> &b) noexcept;
template<class T, class U>
bool operator<=(const constant_ptr<T> &a, const constant_ptr<U> &b) noexcept;
template<class T, class U>
bool operator>=(const constant_ptr<T> &a, const constant_ptr<U> &b) noexcept;

template<class T>
bool operator==(const constant_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator==(nullptr_t, const constant_ptr<T> &x) noexcept;
```

```
template<class T>
bool operator!=(const constant_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator!=(nullptr_t, const constant_ptr<T> &x) noexcept;
template<class T>
bool operator<(const constant_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator<(nullptr_t, const constant_ptr<T> &x) noexcept;
template<class T>
bool operator>(const constant_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator>(nullptr_t, const constant_ptr<T> &x) noexcept;
template<class T>
bool operator<=(const constant_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator<=(nullptr_t, const constant_ptr<T> &x) noexcept;
template<class T>
bool operator>=(const constant_ptr<T> &x, nullptr_t) noexcept;
template<class T>
bool operator>=(nullptr_t, const constant_ptr<T> &x) noexcept;

template<class T>
void swap(constant_ptr<T>& a, constant_ptr<T>& b) noexcept;

}
```

Constructors

```
constexpr global_ptr() noexcept;
constexpr local_ptr() noexcept;
constexpr private_ptr() noexcept;
constexpr constant_ptr() noexcept;
```

Constructs an object which points to nothing.

```
explicit global_ptr(pointer p) noexcept;
explicit local_ptr(pointer p) noexcept;
explicit private_ptr(pointer p) noexcept;
explicit constant_ptr(pointer p) noexcept;
```

Constructs an object which points to `p`.

```
global_ptr(const global_ptr &) noexcept;
local_ptr(const local_ptr &) noexcept;
private_ptr(const private_ptr &) noexcept;
constant_ptr(const constant_ptr &) noexcept;
```

Copy constructor.

```
global_ptr(global_ptr &&r) noexcept;
local_ptr(local_ptr &&r) noexcept;
private_ptr(private_ptr &&r) noexcept;
constant_ptr(constant_ptr &&r) noexcept;
```

Move constructor.

```
constexpr global_ptr(nullptr_t) noexcept;
constexpr local_ptr(nullptr_t) noexcept;
constexpr private_ptr(nullptr_t) noexcept;
constexpr constant_ptr(nullptr_t) noexcept;
```

Constructs an object initialized with `nullptr`.

Assignment operators

```
global_ptr &operator=(const global_ptr &r) noexcept;
local_ptr &operator=(const local_ptr &r) noexcept;
private_ptr &operator=(const private_ptr &r) noexcept;
constant_ptr &operator=(const constant_ptr &r) noexcept;
```

Copy assignment operator

```
global_ptr &operator=(global_ptr &&r) noexcept;
local_ptr &operator=(local_ptr &&r) noexcept;
private_ptr &operator=(private_ptr &&r) noexcept;
constant_ptr &operator=(constant_ptr &&r) noexcept;
```

Move assignment operator

```
global_ptr &operator=(pointer r) noexcept;
local_ptr &operator=(pointer r) noexcept;
private_ptr &operator=(pointer r) noexcept;
constant_ptr &operator=(pointer r) noexcept;
```

Assigns `r` pointer to the stored pointer

```
global_ptr &operator=(nullptr_t) noexcept;
local_ptr &operator=(nullptr_t) noexcept;
private_ptr &operator=(nullptr_t) noexcept;
constant_ptr &operator=(nullptr_t) noexcept;
```

Assigns `nullptr` to the stored pointer

Observers

```
add_lvalue_reference_t<add_global_t<T>> operator*() const noexcept;
add_lvalue_reference_t<add_local_t<T>> operator*() const noexcept;
add_lvalue_reference_t<add_private_t<T>> operator*() const noexcept;
add_lvalue_reference_t<add_constant_t<T>> operator*() const noexcept;
```

Returns `*get()`. It is only defined in single object version of the explicit address space pointer class. The result of this operator is undefined if `get() == nullptr`.

```
pointer operator->() const noexcept;
```

Returns `get()`. It is only defined in single object version of the explicit address space pointer class. The result of this operator is undefined if `get() == nullptr`.

```
reference operator[](size_t pos) const noexcept;
```

Returns `get()[pos]`. The subscript operator is only defined in specialized `global_ptr<T[]>`, `local_ptr<T[]>`, `private_ptr<T[]>` and `constant_ptr<T[]>` version for array types. The result of this operator is undefined if `pos >=` the number of elements in the array to which the stored pointer points.

```
pointer get() const noexcept;
```

Returns the stored pointer.

```
explicit operator bool() const noexcept;
```

Returns `get() != nullptr`.

Modifiers

```
pointer release() noexcept;
```

Assigns `nullptr` to the stored pointer and returns the value `get()` had at the start of the call to release.

```
void reset(pointer p = pointer()) noexcept;
```

Assigned `p` to the stored pointer. It is only defined in single object version of the explicit address space pointer class

```
void reset(pointer p) noexcept;
```

Assigned `p` to the stored pointer. It is only defined in specialized `global_ptr<T[]>`, `local_ptr<T[]>`, `private_ptr<T[]>` and `constant_ptr<T[]>` version for array types.

```
void reset(nullptr_t p = nullptr) noexcept;
```

Equivalent to `reset(pointer())`. It is only defined in specialized `global_ptr<T[]>`, `local_ptr<T[]>`, `private_ptr<T[]>` and `constant_ptr<T[]>` version for array types.

```
void swap(global_ptr& r) noexcept;
void swap(local_ptr& r) noexcept;
void swap(private_ptr& r) noexcept;
void swap(constant_ptr& r) noexcept;
```

Invokes swap on the stored pointers.

```
global_ptr &operator++() noexcept;
local_ptr &operator++() noexcept;
private_ptr &operator++() noexcept;
constant_ptr &operator++() noexcept;
```

Prefix increment operator. Increments the stored pointer by one.

```
global_ptr operator++(int) noexcept;
local_ptr operator++(int) noexcept;
private_ptr operator++(int) noexcept;
constant_ptr operator++(int) noexcept;
```

Postfix increment operator. Increments the stored pointer by one.

```
global_ptr &operator--() noexcept;
local_ptr &operator--() noexcept;
private_ptr &operator--() noexcept;
constant_ptr &operator--() noexcept;
```

Prefix decrement operator. Decrements the stored pointer by one.

```
global_ptr operator--(int) noexcept;
local_ptr operator--(int) noexcept;
private_ptr operator--(int) noexcept;
constant_ptr operator--(int) noexcept;
```

Postfix decrement operator. Decrements the stored pointer by one.

```
global_ptr &operator+=(difference_type r) noexcept;
local_ptr &operator+=(difference_type r) noexcept;
private_ptr &operator+=(difference_type r) noexcept;
constant_ptr &operator+=(difference_type r) noexcept;
```

Adds `r` to the stored pointer and returns `*this`.

```
global_ptr &operator-=(difference_type r) noexcept;
local_ptr &operator-=(difference_type r) noexcept;
private_ptr &operator-=(difference_type r) noexcept;
constant_ptr &operator-=(difference_type r) noexcept;
```

Subtracts `r` to the stored pointer and returns `*this`.

```
global_ptr operator+(difference_type r) noexcept;
local_ptr operator+(difference_type r) noexcept;
private_ptr operator+(difference_type r) noexcept;
constant_ptr operator+(difference_type r) noexcept;
```

Adds `r` to the stored pointer and returns the value `*this` has at the start of `operator+`.

```
global_ptr operator-(difference_type r) noexcept;
local_ptr operator-(difference_type r) noexcept;
private_ptr operator-(difference_type r) noexcept;
constant_ptr operator-(difference_type r) noexcept;
```

Subtracts `r` to the stored pointer and returns the value `*this` has at the start of `operator-`.

Non-member functions

```
template<class T, class U>
bool operator==(const global_ptr<T> &a, const global_ptr<U> &b) noexcept;
template<class T, class U>
bool operator==(const local_ptr<T> &a, const local_ptr<U> &b) noexcept;
template<class T, class U>
bool operator==(const private_ptr<T> &a, const private_ptr<U> &b) noexcept;
template<class T, class U>
bool operator==(const constant_ptr<T> &a, const constant_ptr<U> &b) noexcept;
```

Comparison `operator==` for the explicit address space pointer classes.

```
template<class T>
bool operator==(nullptr_t, const global_ptr<T> &x) noexcept;
template<class T>
bool operator==(const global_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator==(nullptr_t, const local_ptr<T> &x) noexcept;
template<class T>
bool operator==(const local_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator==(nullptr_t, const private_ptr<T> &x) noexcept;
template<class T>
bool operator==(const private_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator==(nullptr_t, const constant_ptr<T> &x) noexcept;
template<class T>
bool operator==(const constant_ptr<T> &x, nullptr_t) noexcept;
```

Comparison `operator==` for the explicit address space pointer classes with a `nullptr_t`.

```
template<class T, class U>
bool operator!=(const global_ptr<T> &a, const global_ptr<U> &b) noexcept;
template<class T, class U>
bool operator!=(const local_ptr<T> &a, const local_ptr<U> &b) noexcept;
template<class T, class U>
bool operator!=(const private_ptr<T> &a, const private_ptr<U> &b) noexcept;
template<class T, class U>
bool operator!=(const constant_ptr<T> &a, const constant_ptr<U> &b) noexcept;
```

Comparison `operator!=` for the explicit address space pointer classes.

```
template<class T>
bool operator!=(nullptr_t, const global_ptr<T> &x) noexcept;
template<class T>
bool operator!=(const global_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator!=(nullptr_t, const local_ptr<T> &x) noexcept;
template<class T>
bool operator!=(const local_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator!=(nullptr_t, const private_ptr<T> &x) noexcept;
template<class T>
bool operator!=(const private_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator!=(nullptr_t, const constant_ptr<T> &x) noexcept;
template<class T>
bool operator!=(const constant_ptr<T> &x, nullptr_t) noexcept;
```

Comparison `operator!=` for the explicit address space pointer classes with a `nullptr_t`.

```
template<class T, class U>
bool operator<(const global_ptr<T> &a, const global_ptr<U> &b) noexcept;
template<class T, class U>
bool operator<(const local_ptr<T> &a, const local_ptr<U> &b) noexcept;
template<class T, class U>
bool operator<(const private_ptr<T> &a, const private_ptr<U> &b) noexcept;
template<class T, class U>
bool operator<(const constant_ptr<T> &a, const constant_ptr<U> &b) noexcept;
```

Comparison `operator<` for the explicit address space pointer classes.

```
template<class T>
bool operator<(nullptr_t, const global_ptr<T> &x) noexcept;
template<class T>
bool operator<(const global_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator<(nullptr_t, const local_ptr<T> &x) noexcept;
template<class T>
bool operator<(const local_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator<(nullptr_t, const private_ptr<T> &x) noexcept;
template<class T>
bool operator<(const private_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator<(nullptr_t, const constant_ptr<T> &x) noexcept;
template<class T>
bool operator<(const constant_ptr<T> &x, nullptr_t) noexcept;
```

Comparison `operator<` for the explicit address space pointer classes with a `nullptr_t`.

```
template<class T, class U>
bool operator>(const global_ptr<T> &a, const global_ptr<U> &b) noexcept;
template<class T, class U>
bool operator>(const local_ptr<T> &a, const local_ptr<U> &b) noexcept;
template<class T, class U>
bool operator>(const private_ptr<T> &a, const private_ptr<U> &b) noexcept;
template<class T, class U>
bool operator>(const constant_ptr<T> &a, const constant_ptr<U> &b) noexcept;
```

Comparison `operator>` for the explicit address space pointer classes.

```
template<class T>
bool operator>(nullptr_t, const global_ptr<T> &x) noexcept;
template<class T>
bool operator>(const global_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator>(nullptr_t, const local_ptr<T> &x) noexcept;
template<class T>
bool operator>(const local_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator>(nullptr_t, const private_ptr<T> &x) noexcept;
template<class T>
bool operator>(const private_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator>(nullptr_t, const constant_ptr<T> &x) noexcept;
template<class T>
bool operator>(const constant_ptr<T> &x, nullptr_t) noexcept;
```

Comparison `operator>` for the explicit address space pointer classes with a `nullptr_t`.

```
template<class T, class U>
bool operator<=(const global_ptr<T> &a, const global_ptr<U> &b) noexcept;
template<class T, class U>
bool operator<=(const local_ptr<T> &a, const local_ptr<U> &b) noexcept;
template<class T, class U>
bool operator<=(const private_ptr<T> &a, const private_ptr<U> &b) noexcept;
template<class T, class U>
bool operator<=(const constant_ptr<T> &a, const constant_ptr<U> &b) noexcept;
```

Comparison `operator<=` for the explicit address space pointer classes.

```
template<class T>
bool operator<=(nullptr_t, const global_ptr<T> &x) noexcept;
template<class T>
bool operator<=(const global_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator<=(nullptr_t, const local_ptr<T> &x) noexcept;
template<class T>
bool operator<=(const local_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator<=(nullptr_t, const private_ptr<T> &x) noexcept;
template<class T>
bool operator<=(const private_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator<=(nullptr_t, const constant_ptr<T> &x) noexcept;
template<class T>
bool operator<=(const constant_ptr<T> &x, nullptr_t) noexcept;
```

Comparison `operator<=` for the explicit address space pointer classes with a `nullptr_t`.

```
template<class T, class U>
bool operator>=(const global_ptr<T> &a, const global_ptr<U> &b) noexcept;
template<class T, class U>
bool operator>=(const local_ptr<T> &a, const local_ptr<U> &b) noexcept;
template<class T, class U>
bool operator>=(const private_ptr<T> &a, const private_ptr<U> &b) noexcept;
template<class T, class U>
bool operator>=(const constant_ptr<T> &a, const constant_ptr<U> &b) noexcept;
```

Comparison `operator>=` for the explicit address space pointer classes.

```
template<class T>
bool operator>=(nullptr_t, const global_ptr<T> &x) noexcept;
template<class T>
bool operator>=(const global_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator>=(nullptr_t, const local_ptr<T> &x) noexcept;
template<class T>
bool operator>=(const local_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator>=(nullptr_t, const private_ptr<T> &x) noexcept;
template<class T>
bool operator>=(const private_ptr<T> &x, nullptr_t) noexcept;

template<class T>
bool operator>=(nullptr_t, const constant_ptr<T> &x) noexcept;
template<class T>
bool operator>=(const constant_ptr<T> &x, nullptr_t) noexcept;
```

Comparison `operator>=` for the explicit address space pointer classes with a `nullptr_t`.

```
template<class T>
void swap(global_ptr<T>& a, global_ptr<T>& b) noexcept;
template<class T>
void swap(local_ptr<T>& a, local_ptr<T>& b) noexcept;
template<class T>
void swap(private_ptr<T>& a, private_ptr<T>& b) noexcept;
template<class T>
void swap(constant_ptr<T>& a, constant_ptr<T>& b) noexcept;
```

Calls `a.swap(b)`

3.4.4. Other functions

The OpenCL C++ address space library implements the address space query and cast functions. The cast function that allows to explicitly convert from a pointer in the generic address space to a pointer in the global, local and private address space.

get_mem_fence

```
template <class T>
mem_fence get_mem_fence (T *ptr);
```

Returns the `mem_fence` value for `ptr`. `ptr` must be the generic pointer and it cannot be the explicit address space pointer (`global_ptr<>`, `local_ptr<>`, `private_ptr<>` and `constant_ptr<>`) or pointer to address space storage class (`global<>*`, `local<>*`, `priv<>*` and `constant<>*`).

dynamic_asptr_cast

```
template<class T, class U>
T dynamic_asptr_cast(U *ptr);
```

Returns a pointer that points to a region in the address space pointer class specified in `T` if `dynamic_asptr_cast` can cast `ptr` to the specified address space. Otherwise it returns `nullptr`. Only `global_ptr<U>`, `local_ptr<U>` and `private_ptr<U>` are valid `T` template arguments. `ptr` must be the generic pointer and it cannot be the explicit address space pointer (`global_ptr<>`, `local_ptr<>`, `private_ptr<>` and `constant_ptr<>`) or pointer to address space storage class (`global<>*`, `local<>*`, `priv<>*` and `constant<>*`).

static_asptr_cast

```
template <class T, class U>
local_ptr<T> static_asptr_cast(local_ptr<U> const& ptr) noexcept;
template <class T, class U>
global_ptr<T> static_asptr_cast(global_ptr<U> const& ptr) noexcept;
template <class T, class U>
constant_ptr<T> static_asptr_cast(constant_ptr<U> const& ptr) noexcept;
template <class T, class U>
private_ptr<T> static_asptr_cast(private_ptr<U> const& ptr) noexcept;
```

The expression `static_cast(r.get())` shall be well formed.

reinterpret_asptr_cast

```
template <class T, class U>
local_ptr<T> reinterpret_asptr_cast(local_ptr<U> const& ptr) noexcept;
template <class T, class U>
global_ptr<T> reinterpret_asptr_cast(global_ptr<U> const& ptr) noexcept;
template <class T, class U>
constant_ptr<T> reinterpret_asptr_cast(constant_ptr<U> const& ptr) noexcept;
template <class T, class U>
private_ptr<T> reinterpret_asptr_cast(private_ptr<U> const& ptr) noexcept;
```

The expression `reinterpret_cast(r.get())` shall be well formed.

3.4.5. Restrictions

1. The objects allocated using `global`, `local` and `constant` storage classes can be passed to a function only by reference or pointer

```
#include <opencl_memory>
#include <opencl_array>
using namespace cl;

kernel void foo(global<array<int, 5>> val) {
    // Error: variable in the global
    //        address space passed by value
    //...
}

kernel void bar(global<array<int, 5>> &val) { // OK
    //...
}

kernel void foobar(global_ptr<int> val) { // OK
    //...
}

kernel void barfoo(global_ptr<int[]> val) { // OK
    //...
}
```

2. The `global`, `local`, `priv` and `constant` storage classes cannot be used as a return type of function

```
#include <opencl_memory>
#include <opencl_array>
using namespace cl;

global<array<int, 5>> programVar;

global<array<int, 5>> foo() { // error: variable in the global
                              // address space returned by value
    return programVar;
}

global<array<int, 5>> &bar() { // OK
    return programVar;
}
```

3. The `global`, `local` and `constant` storage classes cannot be used to declare class members unless `static` keyword is used

```
#include <opencl_memory>
#include <opencl_array>
using namespace cl;

struct Foo {
    global<int> a; // error: class members cannot be qualified
                   // with address space
    local<array<int, 5>> b; // error: class members cannot be
                            // qualified with address space

    static global<int> c; // OK
    static local<array<int, 5>> d; // OK
};
```

4. The `global` storage class cannot be used to declare variables at function scope unless `static` keyword is used

```
#include <opencl_memory>
using namespace cl;

kernel void foo() {
    global<int> b; // error
    static global<int> b; // OK
}
```

5. The `local` variables can be declared only at kernel function scope, program scope and with `static` keyword

```
#include <opencl_memory>
#include <opencl_array>
using namespace cl;

// An array of 5 ints allocated in
// local address space.
local<array<int, 5>> a = { 10 }; // OK: program scope local
                                 // variable

kernel void foo() {
    // A single int allocated in
    // local address space
    local<int> b{1}; // OK
    static local<int> d{1}; // OK

    if(get_local_id(0) == 0) {
        // example of variable in local address space
        // but not declared at __kernel function scope.
        local<int> c{2}; // not allowed
  }
}
```

6. The objects allocated using `global` storage class must be initialized with the constant expression arguments

```
#include <opencl_memory>
#include <opencl_work_item>
using namespace cl;

kernel void foo() {
    int a = get_local_id(0);
    static global<int> b{a}; // undefined behavior
    static global<int> c{0}; // OK
}
```

7. The constructors of objects allocated using `constant` storage class must be constant expression

```
#include <opencl_memory>
#include <opencl_work_item>
using namespace cl;

constant<int> b{0}; // OK

kernel void foo() {
    int a = get_local_id(0);
    static constant<int> b{a}; // undefined behavior
}
```

8. Constant variables must be initialized

```
#include <opencl_memory>
using namespace cl;

constant<int> a{0}; // OK
constant<int> b; // error: constant variable must be initialized

kernel void foo() {
    static constant<int> c{0}; // OK
    static constant<int> d; // error: constant variable must be initialized
}
```

9. The `priv` storage class cannot be used to declare variables in the program scope or with `static` specifier.

```
#include <opencl_memory>
using namespace cl;

priv<int> a{0}; // error: priv variable in program scope

kernel void foo() {
    static priv<int> c{0}; // error: priv variable with static specifier
    priv<int> d; // OK
}
```

10. `T` type used in `constant` storage class cannot have any user defined constructors, destructors, operators and methods

```
#include <opencl_memory>
using namespace cl;

struct bar {
    int get() { return 10; }
};

kernel void foo() {
    constant<bar> a;
    int b = a.get() // undefined behavior
}
```

11. `T` type used in `global`, `local`, `priv` and `constant` storage class cannot be sealed class

```
#include <opencl_memory>
using namespace cl;

struct bar final { };

kernel void foo() {
    local<bar> a; // error: bar is marked as final
}
```

12. Using work-group barriers or relying on a specific work-item to be executed in constructors and destructors of global and local objects can result in undefined behavior

```
#include <opencl_memory>
#include <opencl_synchronization>
using namespace cl;

struct Foo {
    Foo() {
        work_group_barrier(mem_fence::local); // not allowed
    }

    ~Foo() {
        if(get_local_id(0) != 5) { // not allowed
            while(1) {}
        }
    }
};

kernel void bar() {
    local<Foo> a;
}
```

13. All local (address-space) variable declarations in kernel-scope shall be declared before any explicit return statement. Declaring local variable after return statement may cause undefined behavior. Implementation is encouraged to generate at least a warning in such cases.

3.4.6. Examples

Example 1

Example of passing an explicit address space storage object to a kernel.

```
#include <opencl_memory>
using namespace cl;

kernel void foo(global<int> *arg) {
    //...
}
```

Example 2

Example of passing an explicit address space pointer object to a kernel.

```
#include <opencl_memory>
using namespace cl;

kernel void foo(global_ptr<int> arg) {
    //...
}
```

Example 3

Example of casting a generic pointer to an explicit address space pointer object. This is the runtime operation and the `dynamic_asptr_cast` can fail.

```
#include <opencl_memory>
using namespace cl;

kernel void foo(global_ptr<int> arg) {
    int *ptr = arg;
    auto globalPtr = dynamic_asptr_cast<global_ptr<int>>(ptr);
    if(globalPtr)
    {
        //...
    }
}
```

Example 4

Example of using an array with an explicit address space storage class.

```
#include <opencl_memory>
#include <opencl_array>
#include <opencl_work_item>
using namespace cl;

kernel void foo() {
    local<array<int, 2>> localArray;
    if(get_local_id(0) == 0) {
        for(auto it = localArray.begin(); it != localArray.end(); ++it)
            *it = 0;
    }
    work_group_barrier(mem_fence::local);
    localArray[0] += 1;
}
```

Example 5

Example of using a fundamental type with an explicit address space storage class.

```
#include <opencl_memory>
#include <opencl_work_item>
using namespace cl;

kernel void foo() {
    local<int> a;
    if(get_local_id(0) == 0)
        a = 1;

    work_group_barrier(mem_fence::local);
    if(get_local_id(0) == 1)
        a += 1;
}
```

3.5. Specialization Constants Library

The specialization constants are objects that will not have known constant values until after initial generation of a module in an intermediate representation format (e.g. SPIR-V). Such objects are called specialization constants. Application might provide values for the specialization constants that will be used when program is built from an intermediate format.

3.5.1. Header <opencl_spec_constant> Synopsis

```
namespace cl
{
template<class T, unsigned int ID>
struct spec_constant
{
    spec_constant() = delete;
    spec_constant(const spec_constant &) = default;
    spec_constant(spec_constant&&) = default;

    constexpr spec_constant(const T& value);

    spec_constant& operator=(const spec_constant&) = delete;
    spec_constant& operator=(spec_constant&&) = delete;

    const T& get() const noexcept;

    operator const T&() const noexcept;
};

template<class T, unsigned int ID>
const T& get(const spec_constant<T, ID> &r) noexcept;

}
```

3.5.2. spec_constant class methods and get function

spec_constant::spec_constant

```
constexpr spec_constant(const T& value);
```

Constructor of spec_constant class. The value parameter is a default value of the specialization constant that will be used if a value is not set by the host API. It must be a literal value.

get

```
const T& get() const noexcept;

operator const T&() const noexcept;

template<class T, unsigned int ID>
const T& get(const spec_constant<T, ID> &r) noexcept;
```

Return a value of specialization constant. If an object is not specialized from the host, the default value will be returned.

3.5.3. Requirements

Specialization constant variables cannot be defined `constexpr`.

Data

Template parameter `T` in spec_constant class template denotes the data type of specialization constant. The type `T` must be integral or floating point type.

ID

Template parameter `ID` in spec_constant class template denotes an unique ID of the specialization constant that can be used to set a value from the host API. The value of `ID` must be unique within this compilation unit and across any other modules that it is linked with.

3.5.4. Examples

Example 1

Example of using the specialization constant in the kernel.

```
#include <opencl_spec_constant>
cl::spec_constant<int, 1> spec1{ 255 };
constexpr cl::spec_constant<int, 2> spec2{ 255 }; // error, constexpr specialization
                                                  // constant variables are not
allowed

kernel void myKernel()
{
  if(cl::get(spec1) == 255)
  {
       // do something if a default value is used
  }
  else
  {
    // do something if the spec constant was specialized by the host
  }
}
```

Example 2

Example of specializing one of the dimensions in `cl::required_work_group_size` attribute.

```
#include <opencl_spec_constant>
cl::spec_constant<int, 1> spec1{ 512 };

[[cl::required_work_group_size(spec1, 1, 1)]]
kernel void myKernel()
{
   //...
}
```

3.6. Half Wrapper Library

The OpenCL C++ programming language implements a wrapper class for the built-in half data type (see the *Built-in Half Data Type* section). The class methods perform implicit `vload_half` and `vstore_half` operations from the *Vector Data Load and Store Functions* section.

3.6.1. Header <opencl_half> Synopsis

```
namespace cl {
struct fp16
{
    fp16() = default;
    fp16(const fp16 &) = default;
    fp16(fp16 &&) = default;
    fp16 &operator=(const fp16 &) = default;
    fp16 &operator=(fp16 &&) = default;

    explicit operator bool() const noexcept;

#ifdef cl_khr_fp16
    fp16(half r) noexcept;
    fp16 &operator=(half r) noexcept;
    operator half() const noexcept;
#endif

    fp16(float r) noexcept;
    fp16 &operator=(float r) noexcept;
    operator float() const noexcept;

#ifdef cl_khr_fp64
    fp16(double r) noexcept;
    fp16 &operator=(double r) noexcept;
    operator double() const noexcept;
#endif

    fp16 &operator++() noexcept;
    fp16 operator++(int) noexcept;
    fp16 &operator--() noexcept;
    fp16 operator--(int) noexcept;
    fp16 &operator+=(const fp16 &r) noexcept;
    fp16 &operator-=(const fp16 &r) noexcept;
    fp16 &operator*=(const fp16 &r) noexcept;
    fp16 &operator/=(const fp16 &r) noexcept;
};

bool operator==(const fp16& lhs, const fp16& rhs) noexcept;
bool operator!=(const fp16& lhs, const fp16& rhs) noexcept;
bool operator< (const fp16& lhs, const fp16& rhs) noexcept;
bool operator> (const fp16& lhs, const fp16& rhs) noexcept;
bool operator<=(const fp16& lhs, const fp16& rhs) noexcept;
bool operator>=(const fp16& lhs, const fp16& rhs) noexcept;
fp16 operator+(const fp16& lhs, const fp16& rhs) noexcept;
fp16 operator-(const fp16& lhs, const fp16& rhs) noexcept;
fp16 operator*(const fp16& lhs, const fp16& rhs) noexcept;
fp16 operator/(const fp16& lhs, const fp16& rhs) noexcept;

}
```

3.6.2. Constructors

```
fp16(const half &r) noexcept;
```

Constructs an object with a half built-in type.

```
fp16(const float &r) noexcept;
```

Constructs an object with a float built-in type. If the **cl_khr_fp16** extension is not supported, `vstore_half` built-in function is called with the default rounding mode.

```
fp16(const double &r) noexcept;
```

Constructs an object with a double built-in type. If the **cl_khr_fp16** extension is not supported, `vstore_half` built-in function is called with the default rounding mode. The constructor is only present if the double precision support is enabled.

3.6.3. Assignment operators

```
fp16 &operator=(const half &r) noexcept;
```

Assigns r to the stored half type.

```
fp16 &operator=(const float &r) noexcept;
```

Assigns r to the stored half type. If the **cl_khr_fp16** extension is not supported, `vstore_half` built-in function is called with the default rounding mode.

```
fp16 &operator=(const double &r) noexcept;
```

Assigns r to the stored half type. If the **cl_khr_fp16** extension is not supported, `vstore_half` built-in function is called with the default rounding mode. The operator is only present if the double precision support is enabled.

3.6.4. Conversion operators

```
explicit operator bool() const noexcept;
```

Returns `m != 0.0h`. If the **cl_khr_fp16** extension is not supported, `vload_half` built-in function is called.

```
operator half() const noexcept;
```

Conversion operator to the built-in half type.

```
operator float() const noexcept;
```

Conversion operator. If the **cl_khr_fp16** extension is not supported, `vload_half` built-in function is called.

```
operator double() const noexcept;
```

Conversion operator. If the **cl_khr_fp16** extension is not supported, `vload_half` built-in function is called. The operator is only present if the double precision support is enabled.

3.6.5. Arithmetic operations

```
fp16 &operator++() noexcept;
```

Pre-increment operator. If the **cl_khr_fp16** extension is not supported, `vload_half` and `vstore_half` built-in functions are called.

```
fp16 operator++(int) noexcept;
```

Post-increment operator. If the **cl_khr_fp16** extension is not supported, `vload_half` and `vstore_half` built-in functions are called.

```
fp16 &operator--() noexcept;
```

Pre-decrement operator. If the **cl_khr_fp16** extension is not supported, `vload_half` and `vstore_half` built-in functions are called.

```
fp16 operator--(int) noexcept;
```

Pre-decrement operator. If the **cl_khr_fp16** extension is not supported, `vload_half` and `vstore_half` built-in functions are called.

```
fp16 &operator+=(const fp16 &r) noexcept;
```

Addition operator. If the **cl_khr_fp16** extension is not supported, `vload_half` and `vstore_half` built-in functions are called.

```
fp16 &operator-=(const fp16 &r) noexcept;
```

Subtract operator. If the **cl_khr_fp16** extension is not supported, `vload_half` and `vstore_half` built-in functions are called.

```
fp16 &operator*=(const fp16 &r) noexcept;
```

Multiplication operator. If the **cl_khr_fp16** extension is not supported, `vload_half` and `vstore_half` built-in functions are called.

```
fp16 &operator/=(const fp16 &r) noexcept;
```

Division operator. If the **cl_khr_fp16** extension is not supported, `vload_half` and `vstore_half` built-in functions are called.

3.6.6. Non-member functions

```
bool operator==(const fp16& lhs, const fp16& rhs) noexcept;
```

Comparison operator ==. If the **cl_khr_fp16** extension is not supported, `vload_half` built-in function is called.

```
bool operator!=(const fp16& lhs, const fp16& rhs) noexcept;
```

Comparison operator !=. If the **cl_khr_fp16** extension is not supported, `vload_half` built-in function is called.

```
bool operator< (const fp16& lhs, const fp16& rhs) noexcept;
```

Comparison operator <. If the **cl_khr_fp16** extension is not supported, `vload_half` built-in function is called.

```
bool operator> (const fp16& lhs, const fp16& rhs) noexcept;
```

Comparison operator >. If the **cl_khr_fp16** extension is not supported, `vload_half` built-in function is called.

```
bool operator<=(const fp16& lhs, const fp16& rhs) noexcept;
```

Comparison operator ⇐. If the **cl_khr_fp16** extension is not supported, `vload_half` built-in function is called.

```
bool operator>=(const fp16& lhs, const fp16& rhs) noexcept;
```

Comparison operator >=. If the **cl_khr_fp16** extension is not supported, `vload_half` built-in function is called.

```
fp16 operator+(const fp16& lhs, const fp16& rhs) noexcept;
```

Addition operator. If the **cl_khr_fp16** extension is not supported, `vload_half` and `vstore_half` built-in functions are called.

```
fp16 operator-(const fp16& lhs, const fp16& rhs) noexcept;
```

Subtract operator. If the **cl_khr_fp16** extension is not supported, `vload_half` and `vstore_half` built-in functions are called.

```
fp16 operator*(const fp16& lhs, const fp16& rhs) noexcept;
```

Multiplication operator. If the **cl_khr_fp16** extension is not supported, `vload_half` and `vstore_half` built-in functions are called.

```
fp16 operator/(const fp16& lhs, const fp16& rhs) noexcept;
```

Division operator. If the **cl_khr_fp16** extension is not supported, `vload_half` and `vstore_half` built-in functions are called.

3.7. Vector Wrapper Library

The OpenCL C++ programming language implements a vector wrapper type that works efficiently on the OpenCL devices. The vector class supports methods that allow construction of a new vector from a swizzled set of component elements or from a built-in vector type. The vector class can be converted to a corresponding built-in vector type.

The `Size` parameter can be one of: 2, 3, 4, 8 or 16. Any other value should produce a compilation failure. The element type parameter `T`, must be one of the basic scalar types defined in Device built-in scalar data types table except void type.

3.7.1. Header <opencl_vec> Synopsis

```
namespace cl {
static constexpr size_t undef_channel = static_cast<size_t>(-1);
enum class channel : size_t { r = 0, g = 1, b = 2, a = 3, x = 0, y = 1, z = 2, w = 3,
undef = undef_channel };

template<class T, size_t Size>
struct vec
{
    using element_type = T;
    using vector_type = make_vector_t<T, Size>;
    static constexpr size_t size = Size;

    vec( ) = default;
    vec(const vec &) = default;
    vec(vec &&) = default;

    vec(const vector_type &r) noexcept;
    vec(vector_type &&r) noexcept;

    template <class... Params>
    vec(Params... params) noexcept;

    vec& operator=(const vec &) = default;
    vec& operator=(vec &&) = default;

    vec& operator=(const vector_type &r) noexcept;
    vec& operator=(vector_type &&r) noexcept;

    operator vector_type() const noexcept;

    vec& operator++() noexcept;
    vec& operator++(int) noexcept;
    vec& operator--() noexcept;
    vec& operator--(int) noexcept;
    vec& operator+=(const vec &r) noexcept;
    vec& operator+=(const element_type &r) noexcept;
    vec& operator-=(const vec &r) noexcept;
    vec& operator-=(const element_type &r) noexcept;
    vec& operator*=(const vec &r) noexcept;
    vec& operator*=(const element_type &r) noexcept;
    vec& operator/=(const vec &r) noexcept;
    vec& operator/=(const element_type &r) noexcept;
    vec& operator%=(const vec &r) noexcept;
    vec& operator%=(const element_type &r) noexcept;

    template <size_t... Sizes>
    auto swizzle() noexcept;

    template <size_t... Sizes>
    auto swizzle() const noexcept;

#ifdef SIMPLE_SWIZZLES
    auto x() noexcept;
...
```

```
    auto xyzw() noexcept;
...
    auto zzzz() noexcept;
#endif
};

template <size_t... Swizzle, class Vec>
auto swizzle(Vec& v);
template <channel... Swizzle, class Vec>
auto swizzle(Vec& v);

template<class T, size_t Size>
make_vector_t<bool, Size> operator==(const vec<T, Size> &lhs,
                                     const vec<T, Size> &rhs) noexcept;
template<class T, size_t Size>
make_vector_t<bool, Size> operator!=(const vec<T, Size> &lhs,
                                     const vec<T, Size> &rhs) noexcept;
template<class T, size_t Size>
make_vector_t<bool, Size> operator<(const vec<T, Size> &lhs,
                                    const vec<T, Size> &rhs) noexcept;
template<class T, size_t Size>
make_vector_t<bool, Size> operator>(const vec<T, Size> &lhs,
                                    const vec<T, Size> &rhs) noexcept;
template<class T, size_t Size>
make_vector_t<bool, Size> operator<=(const vec<T, Size> &lhs,
                                     const vec<T, Size> &rhs) noexcept;
template<class T, size_t Size>
make_vector_t<bool, Size> operator>=(const vec<T, Size> &lhs,
                                     const vec<T, Size> &rhs) noexcept;
template<class T, size_t Size>
vec<T, Size> operator+(const vec<T, Size> &lhs,
                       const vec<T, Size> &rhs) noexcept;
template<class T, size_t Size>
vec<T, Size> operator+(const vec<T, Size> &lhs, const T &rhs) noexcept;
template<class T, size_t Size>
vec<T, Size> operator+(const T &lhs, const vec<T, Size> &rhs) noexcept;
template<class T, size_t Size>
vec<T, Size> operator-(const vec<T, Size> &lhs,
                       const vec<T, Size> &rhs) noexcept;
template<class T, size_t Size>
vec<T, Size> operator-(const vec<T, Size> &lhs, const T &rhs) noexcept;
template<class T, size_t Size>
vec<T, Size> operator-(const T &lhs, const vec<T, Size> &rhs) noexcept;
template<class T, size_t Size>
vec<T, Size> operator*(const vec<T, Size> &lhs,
                       const vec<T, Size> &rhs) noexcept;
template<class T, size_t Size>
vec<T, Size> operator*(const vec<T, Size> &lhs, const T &rhs) noexcept;
template<class T, size_t Size>
vec<T, Size> operator*(const T &lhs, const vec<T, Size> &rhs) noexcept;
template<class T, size_t Size>
```

```
vec<T, Size> operator/(const vec<T, Size> &lhs,
                       const vec<T, Size> &rhs) noexcept;
template<class T, size_t Size>
vec<T, Size> operator/(const vec<T, Size> &lhs, const T &rhs) noexcept;
template<class T, size_t Size>
vec<T, Size> operator/(const T &lhs, const vec<T, Size> &rhs) noexcept;

}
```

3.7.2. Constructors

```
vec(const vector_type &r) noexcept;
```

Copy constructor. Constructs an object with the corresponding built-in vector type.

```
vec(vector_type &&r) noexcept;
```

Move constructor. Constructs an object with the corresponding built-in vector type.

```
template <class... Params>
vec(Params... params) noexcept;
```

Constructs a vector object from a swizzled set of component elements.

3.7.3. Assignment operators

```
vec& operator=(const vector_type &r) noexcept;
```

Copy assignment operator. The operator assigns a corresponding built-in vector type.

```
vec& operator=(vector_type &&r) noexcept;
```

Move assignment operator. The operator assigns a corresponding built-in vector type.

3.7.4. Conversion operators

```
operator vector_type() const noexcept;
```

Conversion operator. The operator converts from the vector wrapper class to a corresponding built-in vector type.

3.7.5. Arithmetic operations

```
vec& operator++() noexcept;
```

Pre-increment operator.

```
vec& operator++(int) noexcept;
```

Post-increment operator.

```
vec& operator--() noexcept;
```

Pre-decrement operator.

```
vec& operator--(int) noexcept;
```

Post-decrement operator.

```
vec& operator+=(const vec &r) noexcept;
vec& operator+=(const element_type &r) noexcept;
```

Add each element of `r` to the respective element of the current vector in-place.

```
vec& operator-=(const vec &r) noexcept;
vec& operator-=(const element_type &r) noexcept;
```

Subtract each element of `r` from the respective element of the current vector in-place.

```
vec& operator*=(const vec &r) noexcept;
vec& operator*=(const element_type &r) noexcept;
```

Multiply each element of `r` by the respective element of the current vector in-place.

```
vec& operator/=(const vec &r) noexcept;
vec& operator/=(const element_type &r) noexcept;
```

Divide each element of the current vector in-place by the respective element of `r`.

```
vec& operator%=(const vec &r) noexcept;
vec& operator%=(const element_type &r) noexcept;
```

Remainder of each element of the current vector in-place by the respective element of `r`.

3.7.6. Swizzle methods

All swizzle methods return a temporary object representing a swizzled set of the original vector's member elements. The swizzled vector may be used as a source (rvalue) and destination (lvalue). In order to enable the r-value and lvalue swizzling to work, this returns an intermediate swizzled-vector class, which can be implicitly converted to a vector (rvalue evaluation) or assigned to.

```
template <size_t... Sizes>
auto swizzle() noexcept;

template <size_t... Sizes>
auto swizzle() const noexcept;
```

Returns a vector swizzle. The number of template parameters specified in `Sizes` must be from `1` to `Size`. `Sizes` parameters must be channel values: `channel::r`, `channel::b`, `…`. Swizzle letters may be repeated or re-ordered.

```
auto x() noexcept;
...
auto xyzw() noexcept;
...
auto zzzz() noexcept;
```

Returns a swizzle. These swizzle methods are only generated if the user defined the `SIMPLE_SWIZZLES` macro before including *opencl_vec* header.

3.7.7. Non-member functions

```
template <size_t... Swizzle, class Vec>
auto swizzle(Vec& v);

template <channel... Swizzle, class Vec>
auto swizzle(Vec& v);
```

Returns a vector swizzle. The number of template parameters specified in `Sizes` must be from `1` to `Size`. `Sizes` parameters must be channel values: `channel::r`, `channel::b`, `…` . Swizzle letters may be repeated or re-ordered.

```
template<class T, size_t Size>
make_vector_t<bool, Size> operator==(const vec<T, Size> &lhs,
                                     const vec<T, Size> &rhs) noexcept;
```

Return `true` if all elements of `rhs` compare equal to the respective element of `lhs`.

```
template<class T, size_t Size>
make_vector_t<bool, Size> operator!=(const vec<T, Size> &lhs,
                                     const vec<T, Size> &rhs) noexcept;
```

Return `true` if any one element of `rhs` does not compare equal to the respective element of `lhs`.

```
template<class T, size_t Size>
make_vector_t<bool, Size> operator<(const vec<T, Size> &lhs,
                                    const vec<T, Size> &rhs) noexcept;
```

Return `true` if all elements of `lhs` are less than the respective element of `rhs`.

```
template<class T, size_t Size>
make_vector_t<bool, Size> operator>(const vec<T, Size> &lhs,
                                    const vec<T, Size> &rhs) noexcept;
```

Return `true` if all elements of `lhs` are greater than the respective element of `rhs`.

```
template<class T, size_t Size>
make_vector_t<bool, Size> operator<=(const vec<T, Size> &lhs,
                                     const vec<T, Size> &rhs) noexcept;
```

Return `true` if all elements of `lhs` are less than or equal to the respective element of `rhs`.

```
template<class T, size_t Size>
make_vector_t<bool, Size> operator>=(const vec<T, Size> &lhs,
                                     const vec<T, Size> &rhs) noexcept;
```

Return `true` if all elements of `lhs` are greater than or equal to the respective element of `rhs`.

```
template<class T, size_t Size>
vec<T, Size> operator+(const vec<T, Size> &lhs,
                       const vec<T, Size> &rhs) noexcept;

template<class T, size_t Size>
vec<T, Size> operator+(const vec<T, Size> &lhs, const T &rhs) noexcept;

template<class T, size_t Size>
vec<T, Size> operator+(const T &lhs, const vec<T, Size> &rhs) noexcept;
```

Add each element of `rhs` to the respective element of `lhs`.

```
template<class T, size_t Size>
vec<T, Size> operator-(const vec<T, Size> &lhs,
                       const vec<T, Size> &rhs) noexcept;
template<class T, size_t Size>
vec<T, Size> operator-(const vec<T, Size> &lhs, const T &rhs) noexcept;
template<class T, size_t Size>
vec<T, Size> operator-(const T &lhs, const vec<T, Size> &rhs) noexcept;
```

Subtract each element of `rhs` from the respective element of `lhs`.

```
template<class T, size_t Size>
vec<T, Size> operator*(const vec<T, Size> &lhs,
                       const vec<T, Size> &rhs) noexcept;
template<class T, size_t Size>
vec<T, Size> operator*(const vec<T, Size> &lhs, const T &rhs) noexcept;
template<class T, size_t Size>
vec<T, Size> operator*(const T &lhs, const vec<T, Size> &rhs) noexcept;
```

Multiply each element of `rhs` by the respective element of `lhs`.

```
template<class T, size_t Size>
vec<T, Size> operator/(const vec<T, Size> &lhs,
                       const vec<T, Size> &rhs) noexcept;
template<class T, size_t Size>
vec<T, Size> operator/(const vec<T, Size> &lhs, const T &rhs) noexcept;
template<class T, size_t Size>
vec<T, Size> operator/(const T &lhs, const vec<T, Size> &rhs) noexcept;
```

Divide each element of `lhs` by the respective element of the `rhs`.

3.8. Range Library

OpenCL C++ implements small library that contains useful utilities to manipulate iterator ranges.

3.8.1. Header <opencl_range> Synopsis

```
namespace cl
{
template <class It>
struct range_type
{
    constexpr range_type(It& it) noexcept;
    constexpr range_type(It& it, difference_type end) noexcept;
    constexpr range_type(It& it, difference_type begin,
                         difference_type end) noexcept;

    constexpr auto begin( ) noexcept;
    constexpr auto end( ) noexcept;
};

template <class It>
constexpr auto range(It& it) noexcept;

template <class It>
constexpr auto range(It& it, difference_type end) noexcept;

template <class It>
constexpr auto range(It& it, difference_type begin,
                     difference_type end) noexcept;

// difference_type is It::difference_type if present ptrdiff_t otherwise.

}
```

3.8.2. Range type

Range type represents a given range over iterable type. Depending on constructor used:

```
constexpr range_type(It& it) noexcept;
```

Represents range from `begin(it)` to `end(it)`.

```
constexpr range_type(It& it, difference_type end) noexcept;
```

Represents range from `begin(it)` to `begin(it)+end`.

```
constexpr range_type(It& it, difference_type begin,
                     difference_type end) noexcept;
```

Represents range from `begin(it)+begin` to `begin(it)+end`.

3.8.3. Range function

range function is present in three overloads matching `range_type` constructors. It is a factory function building `range_type`.

> This function main purpose is enabling the use of range based for loops on built-in vectors.

3.9. Vector Utilities Library

OpenCL C++ implements vector utilities library that contains multiple helper classes to help working with built-in vectors.

3.9.1. Header <opencl_vector_utility> Synopsis

```
namespace cl
{
template <size_t Channel, class Vec>
constexpr remove_attrs_t<vector_element_t<Vec>> get(Vec & vector) noexcept;

template <size_t Channel, class Vec>
constexpr void set(Vec & vector,
                   remove_attrs_t<vector_element_t<Vec>> value) noexcept;

template <class Vec>
struct channel_ref
{
    using type = remove_attrs_t<vector_element_t<Vec>>;

    constexpr operator type( ) noexcept;
    constexpr channel_ref& operator=(type value) noexcept;
    constexpr channel_ref& operator +=(type value) noexcept;
    constexpr friend type operator +(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator -=(type value) noexcept;
    constexpr friend type operator -(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator *=(type value) noexcept;
    constexpr friend type operator *(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator /=(type value) noexcept;
    constexpr friend type operator /(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator %=(type value) noexcept;
    constexpr friend type operator %(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator ^=(type value) noexcept;
    constexpr friend type operator ^(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator &=(type value) noexcept;
    constexpr friend type operator &(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator |=(type value) noexcept;
    constexpr friend type operator |(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator++( ) noexcept;
    constexpr channel_ref operator++(int) noexcept;
    constexpr channel_ref& operator--( ) noexcept;
    constexpr channel_ref operator--(int) noexcept;
};

template <>
struct channel_ref<floating_point_vector>
{
    using type = remove_attrs_t<vector_element_t<Vec>>;

    constexpr operator type( ) noexcept;
    constexpr channel_ref& operator=(type value) noexcept;
    constexpr channel_ref& operator +=(type value) noexcept;
    constexpr friend type operator +(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator -=(type value) noexcept;
    constexpr friend type operator -(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator *=(type value) noexcept;
    constexpr friend type operator *(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator /=(type value) noexcept;
    constexpr friend type operator /(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator++( ) noexcept;
    constexpr channel_ref& operator++(int) noexcept;
    constexpr channel_ref& operator--( ) noexcept;
    constexpr channel_ref& operator--(int) noexcept;
};

template <>
struct channel_ref<boolean_vector>
{
    using type = remove_attrs_t<vector_element_t<Vec>>;

    constexpr operator type( ) noexcept;
    constexpr channel_ref& operator=(type value) noexcept;
    constexpr channel_ref& operator +=(type value) noexcept;
    constexpr friend type operator +(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator -=(type value) noexcept;
    constexpr friend type operator -(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator *=(type value) noexcept;
    constexpr friend type operator *(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator /=(type value) noexcept;
    constexpr friend type operator /(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator %=(type value) noexcept;
    constexpr friend type operator %(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator ^=(type value) noexcept;
    constexpr friend type operator ^(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator &=(type value) noexcept;
    constexpr friend type operator &(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator |=(type value) noexcept;
    constexpr friend type operator |(channel_ref lhs, type rhs) noexcept;
    constexpr channel_ref& operator++( ) noexcept;
    constexpr channel_ref& operator++(int) noexcept;
};
```

```
template <class Vec>
struct channel_ptr
{
    constexpr channel_ptr( ) noexcept;
    constexpr channel_ptr(const channel_ref<Vec>& ref) noexcept;
    constexpr channel_ptr(const channel_ptr&) noexcept = default;
    constexpr channel_ptr(channel_ptr&&) noexcept = default;
    constexpr channel_ptr& operator=(const channel_ptr&) noexcept = default;
    constexpr channel_ptr& operator=(channel_ptr&&) noexcept = default;

    using type = remove_attrs_t<vector_element_t<Vec>>;

    constexpr channel_ref<Vec>& operator*( ) noexcept;
};

template <class Vec>
struct vector_iterator : iterator<random_access_iterator_tag,
                          remove_attrs_t<vector_element_t<remove_attrs_t<Vec>>>,
                          ptrdiff_t,
                          channel_ptr<remove_attrs_t<Vec>>,
                          channel_ref<remove_attrs_t<Vec>>>
{
    using type = remove_attrs_t<Vec>;

    constexpr vector_iterator(type & vector, size_t offset) noexcept;
    constexpr vector_iterator( ) noexcept = default;
    constexpr vector_iterator(const vector_iterator&) noexcept = default;
    constexpr vector_iterator(vector_iterator&&) noexcept = default;
    constexpr vector_iterator& operator=(
                                     const vector_iterator&) noexcept = default;
    constexpr vector_iterator& operator=(vector_iterator&&) noexcept = default;

    constexpr vector_iterator& operator+=(difference_type value) noexcept;
    constexpr friend vector_iterator operator+(const vector_iterator& lhs,
                                               difference_type rhs) noexcept;
    constexpr friend vector_iterator operator+(difference_type lhs,
                                           const vector_iterator& rhs) noexcept;
    constexpr vector_iterator& operator-=(difference_type value) noexcept;
    constexpr friend vector_iterator operator-(const vector_iterator& lhs,
                                                  difference_type rhs) noexcept;

    constexpr vector_iterator operator++(int) noexcept;
    constexpr vector_iterator& operator++( ) noexcept;
    constexpr vector_iterator operator--(int) noexcept;
    constexpr vector_iterator& operator--( ) noexcept;

    friend constexpr bool operator ==(const vector_iterator& lhs,
                                      const vector_iterator& rhs) noexcept;
    friend constexpr bool operator !=(const vector_iterator& lhs,
                                      const vector_iterator& rhs) noexcept;
    friend constexpr bool operator <(const vector_iterator& lhs,
                                     const vector_iterator& rhs) noexcept;
    friend constexpr bool operator <=(const vector_iterator& lhs,
                                      const vector_iterator& rhs) noexcept;
    friend constexpr bool operator >(const vector_iterator& lhs,
                                     const vector_iterator& rhs) noexcept;
    friend constexpr bool operator >=(const vector_iterator& lhs,
                                      const vector_iterator& rhs) noexcept;

    constexpr reference operator[ ](difference_type value) noexcept;
    constexpr reference operator*( ) noexcept;

    constexpr pointer operator->( ) noexcept;
};

template <class Vec, class = enable_if_t<is_vector_type<Vec>::value, void>>
constexpr channel_ref<Vec> index(Vec& vector, size_t channel) noexcept;

template <class Vec, class = enable_if_t<is_vector_type<Vec>::value, void>>
constexpr vector_iterator<Vec> begin(Vec & vector) noexcept;

template <class Vec, class = enable_if_t<is_vector_type<Vec>::value, void>>
constexpr vector_iterator<Vec> end(Vec & vector) noexcept;

}
```

3.9.2. Vector iterator

Vector iterator is a random access iterator that allows runtime iteration over vector channels. Meets all the requirements for random access iterator. Iterating outside of vector bounds is an undefined behavior.

The library also exposes non member begin and end functions for vectors.

Due to the usage of argument-dependent lookup in range based for loops this functions are not available, and the new range adapter has to be used

There is also an `index` function present in the library that allows runtime numerical indexing of channels. It returns a channel reference to a given channel number. Indexing out of vector bounds results in undefined behavior.

The following example will present simple template function computing sum of channels of a given vector:

```
template<class V>
auto sum(const V& v) {
    vector_element_t<V> temp = 0;
    for(auto e : range(v)) {
        temp += e;
    }
    return temp;
}
```

3.9.3. Channel reference and channel pointer

`channel_ref` and `channel_ptr` classes provide lightweight reference and pointer wrappers for vector channels. This is required due to the fact that vector channels can be moved across memory during execution so direct physical addressing is impossible. Reference wrapper provides a set of binary operators (depending on vector channel type).

The following example will present a simple usage of channel reference to set first channel of given vector to 0:

```
template <class V>
void fun(V& v) {
    channel_ref<V> r = *begin(v);
    r = 0;
}
```

3.9.4. Get and set functions

Get and set functions allow compile time numerical indexing of channels to substitute for normal swizzling. Indexing out of vector range generates a compile error. Get function returns a copy of channel value.

The following example will present how get and set functions can be used to duplicate the value of the first channel of given vector:

```
template <class V>
void fun(V& v) {
    auto c = get< 0 >(v);
    set< 0 >(v, 2*c);
}
```

3.9.5. Examples

Example 1

Example of using built-in vector iterators.

```
#include <opencl_vector_utility>
#include <opencl_range>
using namespace cl;

kernel void foo() {
    int8 v_i8;
    auto iter_begin = begin(v_i8);
    auto iter_end = end(v_i8);

    iter_begin = iter_end;

    int a = 0;
    a = *iter_begin;
    a = iter_begin[0];

    iter_begin++;
    iter_begin+=1;
    iter_begin = iter_begin + 1;

    iter_begin--;
    iter_begin-=1;
    iter_begin = iter_begin - 1;
}
```

Example 2

Example of iterating though built-in vector channels and using range library.

```
#include <opencl_vector_utility>
#include <opencl_range>

kernel void foo() {
    int16 a;

    for (auto it = cl::begin(a); it != cl::end(a); it++) {
        int b = *it;
        *it = 2;
    }

    for (auto c : cl::range(a,3,6)) {
        int b = c;
        c = 2;
    }
}
```

3.10. Marker Types

Some types in OpenCL C++ are considered marker types. These types are special in the manner that their usages need to be tracked by the compiler. This results in the following set of restrictions that marker types have to follow:

- Marker types have the default constructor deleted.
- Marker types have all default copy and move assignment operators deleted.
- Marker types have address-of operator deleted.
- Marker types cannot be used in divergent control flow. It can result in undefined behavior.
- Size of marker types is undefined.

3.10.1. Header <opencl_marker> Synopsis

```
namespace cl
{
struct marker_type;

template<class T>
struct is_marker_type;

}
```

3.10.2. marker_type class

All special OpenCL C++ types must use the `marker_type` class as a base class.

3.10.3. is_marker_type type trait

`is_marker_type` type trait provides compile-time check if the base of a class is `marker_type`.

```
namespace cl
{
template<class T>
struct is_marker_type : integral_constant <bool, is_base_of<marker_type, T>::value> {
};

}
```

3.10.4. Examples

Example 1

The examples of invalid use of `marker types`.

```
#include <opencl_image>
#include <opencl_work_item>
using namespace cl;

float4 bar(image1d<float4> img) {
    return img.read({get_global_id(0), get_global_id(1)});
}

kernel void foo(image1d<float4> img1, image1d<float4> img2) {
    image1d<float4> img3; //error: marker type cannot be declared
                          //       in the kernel
    img1 = img2; //error: marker type cannot be assigned
    image1d<float4> *imgPtr = &img1; //error: taking address of
                                     //       marker type

    size_t s = sizeof(img1); //undefined behavior: size of marker
                             //                type is not defined

    float4 val = bar(get_global_id(0) ? img1: img2);
                          //undefined behavior: divergent control flow
}
```

Example 2

The examples of how to use `is_marker_type` trait.

```
#include <opencl_image>
using namespace cl;

kernel void foo(image1d<float4> img) {
  static_assert(is_marker_type<decltype(img)>(), "");
}
```

3.11. Images and Samplers Library

This section describes the image and sampler types and functions that can be used to read from and/or write to an image. `image1d`, `image1d_buffer`, `image1d_array`, `image2d`, `image2d_array`, `image3d`, `image2d_depth`, `image2d_array_depth`, `image2d_ms`, `image2d_array_ms`, `image2d_depth_ms`, `image2d_array_depth_ms` [13] and `sampler` follow the rules for marker types (see the *Marker Types* section). The image and sampler types can only be used if the device support images i.e. `CL_DEVICE_IMAGE_SUPPORT` as described in table 4.3 in OpenCL 2.2 specification is `CL_TRUE`.

3.11.1. Image and Sampler Host Types

The below table describes the OpenCL image and sampler data types and the corresponding data type available to the application:

Table 10. Host image and sampler types

Type in OpenCL C++	API type for application
`cl::image1d`, `cl::image1d_buffer`, `cl::image1d_array`, `cl::image2d`, `cl::image2d_array`, `cl::image3d`, `cl::image2d_depth`, `cl::image2d_array_depth`, `cl::image2d_ms`, `cl::image2d_array_ms`, `cl::image2d_depth_ms`, `cl::image2d_array_depth_ms`	`cl_image`
`cl::sampler`	`cl_sampler`

3.11.2. Header <opencl_image> Synopsis

```
namespace cl
{
enum class image_access;
enum class image_channel_type;
enum class image_channel_order;
enum class addressing_mode;
enum class normalized_coordinates;
enum class filtering_mode;

struct sampler;

template <addressing_mode A, normalized_coordinates C, filtering_mode F>
constexpr sampler make_sampler();

template <class T, image_access A, image_dim Dim, bool Depth, bool Array,
          bool MS>
struct image;

template <class T, image_access A = image_access::read>
using image1d = image<T, A, image_dim::image_1d, false, false, false>;

template <class T, image_access A = image_access::read>
using image1d_buffer = image<T, A, image_dim::image_buffer, false, false,
                             false>;

template <class T, image_access A = image_access::read>
using image1d_array = image<T, A, image_dim::image_1d, false, true, false>;

template <class T, image_access A = image_access::read>
using image2d = image<T, A, image_dim::image_2d, false, false, false>;
template <class T, image_access A = image_access::read>
using image2d_depth = image<T, A, image_dim::image_2d, true, false, false>;

template <class T, image_access A = image_access::read>
using image2d_array = image<T, A, image_dim::image_2d, false, true, false>;

template <class T, image_access A = image_access::read>
using image3d = image<T, A, image_dim::image_3d, false, false, false>;

template <class T, image_access A = image_access::read>
using image2d_array_depth = image<T, A, image_dim:: image_2d, true, true,
                                  false>;

#if defined(cl_khr_gl_msaa_sharing) && defined(cl_khr_gl_depth_images)
template <class T, image_access A = image_access::read>
using image2d_ms = image<T, A, image_dim::image_2d, false, false, true>;

template <class T, image_access A = image_access::read>
using image2d_array_ms = image<T, A, image_dim::image_2d, false, true, true>;

template <class T, image_access A = image_access::read>
using image2d_depth_ms = image<T, A, image_dim::image_2d, true, false, true>;

template <class T, image_access A = image_access::read>
using image2d_array_depth_ms = image<T, A, image_dim::image_2d, true, true,
                                     true>;
#endif

}
```

Where T is the type of value returned when reading or sampling from given image or the type of color used to write to image.

3.11.3. image class

Every image type has the following set of publicly available members and typedefs:

```
template <class T, image_access A, image_dim Dim, bool Depth, bool Array,
          bool MS>
struct image: marker_type
{
    static constexpr image_dim dimension = Dim;
    static constexpr size_t dimension_num = image_dim_num<Dim>::value;
    static constexpr size_t image_size = dimension_num + (Array? 1: 0);
    static constexpr image_access access = A;
    static constexpr bool is_array = Array;
    static constexpr bool is_depth = Depth;
#if defined(cl_khr_gl_msaa_sharing) && defined(cl_khr_gl_depth_images)
    static constexpr bool is_ms = MS;
#else
    static constexpr bool is_ms = false;
#endif
    typedef element_type T;
    typedef integer_coord make_vector_t<int, image_size>;
    typedef float_coord make_vector_t<float, image_size>;

#ifdef cl_khr_mipmap_image
    typedef gradient_coord make_vector_t<float, dimension_num>;
#endif

    struct pixel;

    image() = delete;
    image(const image&) = default;
    image(image&&) = default;

    image& operator=(const image&) = delete;
    image& operator=(image&&) = delete;
    image* operator&() = delete;
};
```

3.11.4. Image element types

We can classify images into four categories: depth images which have the `Depth` template parameter set to `true`, multi-sample depth images which have the `Depth` and `MS` template parameters set to `true`, multi-sample which have the `MS` template parameter set to `true`, and the normal images which have the `Depth` and `MS` template parameters set to `false`.

- For non-multisample depth images the only valid element types are: `float` and `half` [4]
- For normal images the only valid element types are: `float4`, `half4` [4], `int4` and `uint4`
- For multi-sample 2D and multi-sample 2D array images the only valid element types are: `float4`, `int4` and `uint4`
- For multi-sample 2D depth and multi-sample 2D array depth images the only valid element type is: `float`

Image type with invalid pixel type is ill formed.

3.11.5. Image dimension

```
namespace cl
{
enum class image_dim
{
    image_1d,
    image_2d,
    image_3d,
    image_buffer
};

template <image_dim Dim>
struct image_dim_num;

}
```

Image types present different set of methods depending on their dimensionality and arrayness.

- Images of dimension 1 (`image_dim::image_1d` and `image_dim::buffer`) have method:

  ```
  int width() const noexcept;
  ```

- Images of dimension 2 (`image_dim::image_2d`) have all methods of 1 dimensional images and

  ```
  int height() const noexcept;
  ```

- Images of dimension 3 (`image_dim::image_3d`) have all methods of 2 dimensional images and

  ```
  int depth() const noexcept;
  ```

- Arrayed images have additional method

  ```
  int array_size() const noexcept;
  ```

If the **cl_khr_mipmap_image** or **cl_khr_mipmap_image_writes** extension is enabled, then the following methods are also present:

- Images of dimension 1 (`image_dim::image_1d` and `image_dim::buffer`) have method:

  ```
  int width(int lod) const noexcept;
  ```

- Images of dimension 2 (`image_dim::image_2d`) have all methods of 1 dimensional images and

  ```
  int height(int lod) const noexcept;
  ```

- Images of dimension 3 (`image_dim::image_3d`) have all methods of 2 dimensional images and

  ```
  int depth(int lod) const noexcept;
  ```

- Arrayed images have additional method

  ```
  int array_size(int lod) const noexcept;
  ```

If the **cl_khr_gl_msaa_sharing** and **cl_khr_gl_depth_images** extensions are enabled, then the following methods are also present:

- Images of dimension 2D (`image_dim::image_2d`) have method:

  ```
  int num_samples() const noexcept;
  ```

The following table describes the `image_dim_num` trait that return a number of dimensions based on `image_dim` parameter.

Table 11. Image_dim_num trait

Template	**Value**
`template <image_dim Dim> struct image_dim_num;`	If `Dim` is `image_dim::image_1d` or `image_dim::image_buffer`, image dimension is 1. If `Dim` is `image_dim::image_2d`, image dimension is 2. If `Dim` is `image_dim::image_3d`, image dimension is 3.

3.11.6. Image access

```
namespace cl
{
enum class image_access
{
    sample,
    read,
    write,
    read_write
};

}
```

The non-multisample image template class specializations present different set of methods based on their access parameter.

- Images specified with `image_access::read` provide additional methods:

  ```
  element_type image::read(integer_coord coord) const noexcept;

  pixel image::operator[](integer_coord coord) const noexcept;

  element_type image::pixel::operator element_type() const noexcept;
  ```

- Images specified with `image_access::write` provide additional method:

  ```
  void image::write(integer_coord coord, element_type color) noexcept;

  image::pixel image::operator[](integer_coord coord) noexcept;

  image::pixel & image::pixel::operator=(element_type color) noexcept;
  ```

- Images specified with `image_access::read_write` provide additional methods:

  ```
  element_type image::read(integer_coord coord) const noexcept;

  void image::write(integer_coord coord, element_type color) noexcept;

  image::pixel image::operator[](integer_coord coord) noexcept;

  element_type image::pixel::operator element_type() const noexcept;

  image::pixel & image::pixel::operator=(element_type color) noexcept;
  ```

- Images specified with `image_access::sample` provide additional methods:

```
element_type image::read(integer_coord coord) const noexcept;

element_type image::sample(const sampler &s,
                           integer_coord coord) const noexcept;

element_type image::sample(const sampler &s, float_coord coord) const noexcept;

image::pixel image::operator[](integer_coord coord) const noexcept;

element_type image::pixel::operator element_type() const noexcept;
```

If the **cl_khr_mipmap_image** extension is enabled the following methods are added to the non-multisample image types:

- Images specified with `image_access::sample` provide additional methods:

```
element_type image::sample(const sampler &s, float_coord coord,
                           float lod) const noexcept;

element_type image::sample(const sampler &s, float_coord coord,
                           gradient_coord gradient_x,
                           gradient_coord gradient_y) const noexcept;
```

If the **cl_khr_mipmap_image_writes** extension is enabled the following methods are added to the non-multisample image types:

- Images specified with `image_access::write` provide additional method:

```
void image::write(integer_coord coord, element_type color, int lod) noexcept;
```

If the **cl_khr_gl_msaa_sharing** and **cl_khr_gl_depth_images** extensions are enabled and the multisample image type is used, the following method is available:

- The multisample images specified with `image_access::read` provide method:

```
element_type image::read(integer_coord coord, int sample) noexcept;
```

3.11.7. Common image methods

Each image type implements a set of common methods:

```
image_channel_type image::data_type() const noexcept;
image_channel_order image::order() const noexcept;
```

If the **cl_khr_mipmap_image** or **cl_khr_mipmap_image_writes** extension is enabled, then the following method is also present in the non-multisample image types:

```
int image::miplevels() const noexcept;
```

where `image_channel_type` and `image_channel_order` are defined as follows:

```
namespace cl
{
enum class image_channel_type
{
    snorm_int8,
    snorm_int16,
    unorm_int8,
    unorm_int16,
    unorm_int24,
    unorm_short_565,
    unorm_short_555,
    unorm_int_101010,
    unorm_int_101010_2,
    signed_int8,
    signed_int16,
    signed_int32,
    unsigned_int8,
    unsigned_int16,
    unsigned_int32,
    fp16,
    fp32
};

enum class image_channel_order
{
    a,
    r,
    rx,
    rg,
    rgx,
    ra,
    rgb,
    rgbx,
    rgba,
    argb,
    bgra,
    intensity,
    luminance,
    abgr,
    depth,
    depth_stencil,
    srgb,
    srgbx,
    srgba,
    sbgra
};

}
```

3.11.8. Other image methods

image::sample

```
element_type image::sample(const sampler &s, float_coord coord) const noexcept;
```

Reads a color value from the non-multisample image using sampler and floating point coordinates.

```
element_type image::sample(const sampler &s, integer_coord coord) const noexcept;
```

Reads a color value from non-multisample image using sampler and integer coordinates.

A sampler must use filter mode set to `filtering_mode::nearest`, normalized coordinates and addressing mode set to `addressing_mode::clamp_to_edge`, `addressing_mode::clamp`, `addressing_mode::none`, otherwise the values returned are undefined.

```
element_type image::sample(const sampler &s, float_coord coord, float lod) const
noexcept;
```

Reads a color value from non-multisample image using sampler and floating point coordinates in the mip-level specified by `lod`.

Method is present for non-multisample images if the **cl_khr_mipmap_image** extension is enabled.

```
element_type image::sample(const sampler &s, float_coord coord,
                           gradient_coord gradient_x,
                           gradient_coord gradient_y) const noexcept;
```

Use the gradients to compute the lod and coordinate `coord` to do an element lookup in the mip-level specified by the computed lod.

Method is present if the **cl_khr_mipmap_image** extension is enabled.

Based on the parameters with which image was created on host side the function will return different ranges of values

- returns floating-point values in the range [0.0, 1.0] for image objects created with `image_channel_type` set to one of the pre-defined packed formats or `image_channel_type::unorm_int8` or `image_channel_type::unorm_int16`.
- returns floating-point values in the range [-1.0, 1.0] for image objects created with `image_channel_type::snorm_int8` or `image_channel_type::snorm_int16`.
- returns floating-point values for image objects created with `image_channel_type::float16` or `image_channel_type::float32`.

Values returned by `image::sample` where `T` is a floating-point type for image objects with image_channel_type values not specified in the description above are undefined.

The image::sample functions that take an image object where T is a signed integer type can only be used with image objects created with:

- image_channel_type::sint8,
- image_channel_type::sint16 and
- image_channel_type::sint32.

If the image_channel_type is not one of the above values, the values returned by image::sample are undefined.

The image::sample functions that take an image object where T is an unsigned integer type can only be used with image objects created with:

- image_channel_type::uint8,
- image_channel_type::uint16 and
- image_channel_type::uint32.

If the image_channel_type is not one of the above values, the values returned by image::sample are undefined.

image::read

```
element_type image::read(integer_coord coord) const noexcept;
```

Reads a color value from non-multisample image without sampler and integral coordinates. If the **cl_khr_mipmap_image** extension is present, may also perform reads from mipmap layer 0.

The image read function behaves exactly as the corresponding image sample function using a sampler that has filter mode set to filtering_mode::nearest, normalized coordinates set to normalized_coordinates::unnormalized and addressing mode set to addressing_mode::none. There is one execption for cases where the image sample type is image_channel_type::fp32. In this exceptional case, when channel data values are denormalized, the image read function may return the denormalized data, while the sample function may flush denormalized channel data values to zero. The coordinates must be between 0 and image size in that dimension non inclusive.

Based on the parameters with which image was created on host side the function will return different ranges of values

- returns floating-point values in the range [0.0, 1.0] for image objects created with image_channel_type set to one of the pre-defined packed formats or image_channel_type::unorm_int8 or image_channel_type::unorm_int16.
- returns floating-point values in the range [-1.0, 1.0] for image objects created with image_channel_type::snorm_int8 or image_channel_type::snorm_int16.
- returns floating-point values for image objects created with image_channel_type::float16 or image_channel_type::float32.

Values returned by image::read where T is a floating-point type for image objects with image_channel_type values not specified in the description above are undefined.

The image::read functions that take an image object where T is a signed integer type can only be used with image objects created with:

- image_channel_type::sint8,
- image_channel_type::sint16 and
- image_channel_type::sint32.

If the image_channel_type is not one of the above values, the values returned by image::read are undefined.

The image::read functions that take an image object where T is an unsigned integer type can only be used with image objects created with image_channel_type set to one of the following values:

- image_channel_type::uint8,
- image_channel_type::uint16 and
- image_channel_type::uint32.

If the image_channel_type is not one of the above values, the values returned by image::read are undefined.

```
element_type image::read(integer_coord coord, int sample) noexcept;
```

Use the coordinate and sample to do an element lookup in the image object. Method is only available in the MSAA image types, and if the **cl_khr_gl_msaa_sharing** and **cl_khr_gl_depth_images** extensions are enabled.

When a multisample image is accessed in a kernel, the access takes one vector of integers describing which pixel to fetch and an integer corresponding to the sample numbers describing which sample within the pixel to fetch. sample identifies the sample position in the multi-sample image.

For best performance, we recommend that sample be a literal value so it is known at compile time and the OpenCL compiler can perform appropriate optimizations for multisample reads on the device.

No standard sampling instructions are allowed on the multisample image. Accessing a coordinate outside the image and/or a sample that is outside the number of samples associated with each pixel in the image is undefined.

image::write

```
void image::write(integer_coord coord, element_type color) noexcept;
```

Writes a color value to location specified by coordinates from non-multisample image. If the

cl_khr_mipmap_image_writes extension is present, may also perform writes to mipmap layer 0. The coordinates must be between 0 and image size in that dimension non inclusive.

Based on the parameters with which image was created on host side the function will perform appropriate data format conversions before writing a color value.

```
void image::write(integer_coord coord, element_type color, int lod) noexcept;
```

Writes a color value to location specified by coordinates and lod from mipmap image. The coordinates must be between 0 and image size in that dimension non inclusive.

Method is present if the **cl_khr_mipmap_image** extension is enabled.

Based on the parameters with which image was created on host side the function will perform appropriate data format conversions before writing a color value.

The `image::write` functions that take an image object where `T` is a floating-point type can only be used with image objects created with `image_channel_type` set to one of the pre-defined packed formats or set to:

- `image_channel_type::snorm_int8`
- `image_channel_type::unorm_int8`
- `image_channel_type::snorm_int16`
- `image_channel_type::unorm_int16`
- `image_channel_type::float16`
- `image_channel_type::float32`

The `image::write` functions that take an image object where `T` is a signed integer type can only be used with image objects created with:

- `image_channel_type::sint8`
- `image_channel_type::sint16`
- `image_channel_type::sint32`

The `image::write` functions that take an image object where `T` is an unsigned integer type can only be used with image objects created with:

- `image_channel_type::uint8`
- `image_channel_type::uint16`
- `image_channel_type::uint32`

The behavior of `image::write` for image objects created with `image_channel_type` values not specified in the description above is undefined.

\image::operator[]

```
pixel operator[](integer_coord coord) noexcept;

pixel operator[](integer_coord coord) const noexcept;
```

Creates a pixel which can be used to read or/and write operation(s). It depends on image_access specified in the image.

> The pixel stores a reference to image and coordinates. This operation can consume more private memory than `image::read` and `image::write` methods. It can also negatively impact performance.

image::pixel::operator element_type

```
element_type pixel::operator element_type() const noexcept;
```

Reads a color value from non-multisample image without sampler and integral coordinates specified in pixel. If the **cl_khr_mipmap_image** extension is present, may also perform reads from mipmap layer 0.

This function is similar to `image::read` method. Please refer to description of this method for more details.

image::pixel::operator=

```
pixel & pixel::operator=(element_type color) noexcept;
```

Writes a color value to location specified by coordinates in pixel from non-multisample image. If the **cl_khr_mipmap_image_writes** extension is present, may also perform writes to mipmap layer 0. The coordinates specified in pixel must be between 0 and image size in that dimension non inclusive.

Based on the parameters with which image was created on host side the function will perform appropriate data format conversions before writing a color value.

This function is similar to `image::write` method. Please refer to description of this method for more details.

image::width

```
int width() const noexcept;
```

Returns width of the image.

```
int width(int lod) const noexcept;
```

Returns width of the mip-level specified by `lod`.

Method is present in the non-multisample image types if the **cl_khr_mipmap_image** extension is enabled.

image::height

```
int height() const noexcept;
```

Returns height of the image.

```
int height(int lod) const noexcept;
```

Returns height of the mip-level specified by `lod`.

Method is present in the non-multisample image types if the **cl_khr_mipmap_image** extension is enabled.

image::depth

```
int depth() const noexcept;
```

Returns depth of the image.

```
int depth(int lod) const noexcept;
```

Returns depth of the mip-level specified by `lod`.

Method is present in the non-multisample image types if the **cl_khr_mipmap_image** extension is enabled.

image::array_size

```
int array_size() const noexcept;
```

Returns size of the image array.

```
int array_size(int lod) const noexcept;
```

Returns size of the image array specified by `lod`.

Method is present in the non-multisample image types if the **cl_khr_mipmap_image** extension is enabled.

image::size

```
integer_coord size() const noexcept;
```

Returns appropriately sized vector, or scalar for 1 dimensional images, containing all image dimensions followed by array size.

image::data_type

```
image_channel_type image::data_type() const noexcept;
```

Returns format of the image as specified upon its creation on host side.

image::order

```
image_channel_order image::order() const noexcept;
```

Returns channel order of the image as specified upon its creation on host side.

image::miplevels

```
int miplevels() const noexcept;
```

Returns number of mipmaps of image. Method is present if the **cl_khr_mipmap_image** or **cl_khr_mipmap_image_writes** extension is enabled.

image::num_samples

```
int num_samples() const noexcept;
```

3.11.9. Sampler

```
namespace cl
{
struct sampler: marker_type
{
    sampler() = delete;
    sampler(const sampler&) = default;
    sampler(sampler&&) = default;

    sampler& operator=(const sampler&) = delete;
    sampler& operator=(sampler&&) = delete;
    sampler* operator&() = delete;

};

template <addressing_mode A, normalized_coordinates C, filtering_mode F>
constexpr sampler make_sampler();

}
```

There are only two ways of acquiring a sampler inside of a kernel. One is to pass it as a kernel parameter from host using `clSetKernelArg`, the other one is to create one using make_sampler function in the kernel code. `make_sampler` function has three template parameters specifying behavior of sampler. Once acquired sampler can only be passed by reference as all other marker types. The sampler objects at non-program scope must be declared with static specifier.

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

3.11.10. Sampler Modes

```
namespace cl
{
enum class addressing_mode
{
    mirrored_repeat,
    repeat,
    clamp_to_edge,
    clamp,
    none
};

enum class normalized_coordinates
{
    normalized,
    unnormalized
};

enum class filtering_mode
{
    nearest,
    linear
};

}
```

The following tables describe the inline sampler parameters and their behavior.

Table 12. Addressing modes

Addressing mode	Description
`mirrored_repeat`	Out of range coordinates will be flipped 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.
`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.
`clamp_to_edge`	Out of range image coordinates are clamped to the extent.
`clamp`	Out of range image coordinates will return a border color.

Addressing mode	Description
`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 clamp_to_edge.

Table 13. Normalized coordinates

Normalized Coordinate Values	Description
`normalized`	Specifies whether the *x*, *y* and *z* coordinates are passed in as normalized values.
`unnormalized`	Specifies whether the *x*, *y* and *z* coordinates are passed in as unnormalized values.

Sampling from an image with samplers that differ in specification of coordinates normalization result in undefined behavior.

Table 14. Coordinate filtering modes

Filtering mode	Description
`nearest`	Chooses a color of nearest pixel.
`linear`	Performs a linear sampling of adjacent pixels.

Refer to *section 4.2* in the OpenCL API specification for a description of these filter modes.

3.11.11. Determining the border color or value

If `<addressing mode>` in sampler is 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 `image_channel_order::a`, `image_channel_order::intensity`, `image_channel_order::rx`, `image_channel_order::ra`, `image_channel_order::rgx`, `image_channel_order::rgbx`, `image_channel_order::srgbx`, `image_channel_order::argb`, `image_channel_order::bgra`, `image_channel_order::abgr`, `image_channel_order::rgba`, `image_channel_order::srgba` or `image_channel_order::sbgra`, the border color is (0.0f, 0.0f, 0.0f, 0.0f).
- If the image channel order is `image_channel_order::r`, `image_channel_order::rg`, `image_channel_order::rgb`, or `image_channel_order::luminance`, the border color is (0.0f, 0.0f, 0.0f, 1.0f).
- If the image channel order is `image_channel_order::depth`, the border value is 0.0f.

3.11.12. sRGB Images

The built-in image read functions perform sRGB to linear RGB conversions if the image is an sRGB image. Writes 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 `image::write`. `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 the **cl_khr_srgb_image_writes** extension is supported and has been enabled, the built-in image write functions 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.

3.11.13. Reading and writing to the same image in a kernel

To read and write to the same image in a kernel, the image must be declared with the `image_access::read_write`. Only sampler-less reads and write functions can be called on an image declared with the `image_access::read_write` access qualifier. Calling the `image::sample` functions on an image declared with the `image_access::read_write` will result in a compilation error.

The `atomic_fence` function from the *Atomic Fences* section can be used to make sure that writes are visible to later reads by the same work-item. Without use of the `atomic_fence` function, write-read coherence on image objects is not guaranteed: if a work-item reads from an image to which it has previously written without an intervening atomic_fence, it is not guaranteed that those previous writes are visible to the work-item. Only a scope of `memory_order_acq_rel` is valid for `atomic_fence` when passed the `mem_fence::image` flag. If multiple work-items are writing to and reading from multiple locations in an image, the `work_group_barrier` from the *Synchronization Functions* section should be used.

Consider the following example:

```
#include <opencl_work_item>
#include <opencl_atomic>
#include <opencl_image>
using namespace cl;

kernel void foo(image2d<float4, image_access::read_write> img, ... ) {
    int2 coord;
    coord.x = (int)get_global_id(0);
    coord.y = (int)get_global_id(1);

    float4 clr = img.read(coord);
    //...
    img.write(coord, clr);

    // required to ensure that following read from image at
    // location coord returns the latest color value.
    atomic_fence(mem_fence::image,
     memory_order_acq_rel,
     memory_scope_work_item);

    float4 clr_new = img.read(coord);
    //...
}
```

3.11.14. Mapping image channels to color values returned by image::sample, image::read and color values passed to image::write 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 `image::sample`, `image::read` or supplied to `image::write`. 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.

Table 15. Image channel mappings

Image Channel Order	**float4, int4 or uint4 components of channel data**
r, rx	(r, 0.0, 0.0, 1.0)
a	(0.0, 0.0, 0.0, a)
rg, rgx	(r, g, 0.0, 1.0)
ra	(r, 0.0, 0.0, a)
rgb, rgbx, srgb, srgbx	(r, g, b, 1.0)
rgba, bgra, argb, abgr, srgba, sbgra	(r, g, b, a)
intensity	(I, I, I, I)
luminance	(L, L, L, 1.0)

For `image_channel_order::depth` images, a scalar value is returned by `image::sample`, `image::read` or supplied to `image::write`.

A kernel that uses a sampler with the 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 `image::sample`, 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. The 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.

3.11.15. Restrictions

- The image and sampler types cannot be used with variables declared inside a class or union field, a pointer type, an array, global variables declared at program scope or the return type of a function.
- The image and sampler types cannot be used with the `global`, `local`, `priv` and `constant` address space storage classes (see the *Explicit address space storage classes* section).
- The values returned by applying the `sizeof` operator to the image and sampler types are implementation-defined.

3.11.16. Examples

Example 1

The example how to use an image object with sampler-less reads.

```
#include <opencl_image>
#include <opencl_work_item>
using namespace cl;

kernel void foo(image2d<float4, image_access::read> img) {
    int2 coord;
    coord.x = get_global_id(0);
    coord.y = get_global_id(1);

    float4 val = img.read(coord);
}
```

Example 2

The example how to use an image object with `image_access::read_write` access and `atomic_fence` function.

```
#include <opencl_image>
#include <opencl_atomic>
#include <opencl_work_item>
using namespace cl;

kernel void foo(image2d<float4, image_access::read_write> img) {
    int2 coord;
    coord.x = get_global_id(0);
    coord.y = get_global_id(1);

    float4 val1{0.5f};
    img[coord] = val1;

    atomic_fence(mem_fence::image, memory_order_acq_rel,
                 memory_scope_work_item);

    float4 val2 = img[coord];
}
```

Example 3

The example how to use an image object with sampler passed by a kernel argument.

```
#include <opencl_image>
#include <opencl_work_item>
using namespace cl;

kernel void foo(image2d<float4, image_access::sample> img, sampler s) {
    int2 coord;
    coord.x = get_global_id(0);
    coord.y = get_global_id(1);

    float4 val = img.sample(s, coord);
}
```

Example 4

The example how to use an image object with sampler declared at program scope.

```
#include <opencl_image>
#include <opencl_work_item>
using namespace cl;

sampler s = make_sampler<addressing_mode::clamp,
                         normalized_coordinates::unnormalized,
                         filtering_mode::nearest>();

kernel void foo(image2d<float4, image_access::sample> img) {
    int2 coord;
    coord.x = get_global_id(0);
    coord.y = get_global_id(1);

    float4 val = img.sample(s, coord);
}
```

Example 5

The example how to use an image object with sampler declared at non-program scope.

```
#include <opencl_image>
#include <opencl_work_item>
using namespace cl;

kernel void foo(image2d<float4, image_access::sample> img) {
    int2 coord;
    coord.x = get_global_id(0);
    coord.y = get_global_id(1);

    static sampler s = make_sampler<addressing_mode::clamp,
                                    normalized_coordinates::unnormalized,
                                    filtering_mode::nearest>();

    float4 val = img.sample(s, coord);
}
```

3.12. Pipes Library

Header <*opencl_pipe*> defines `pipe` and `pipe_storage` template classes. `pipe` and `pipe_storage` can be used as a communication channel between kernels. `pipe`, `reservation` and `pipe_storage` template classes follow all the rules for marker types as specified in the *Marker Types* section.

3.12.1. Pipe Host Type

The below describes the OpenCL pipe data type and the corresponding data type available to the application:

Table 16. Host pipe type

Type in OpenCL C++	API type for application
`cl::pipe`	`cl_pipe`

3.12.2. Header <opencl_pipe> Synopsis

```
namespace cl
{
enum class pipe_access { read, write };

template <class T, pipe_access Access = pipe_access::read>
struct pipe;

template <class T, size_t N>
struct pipe_storage;

template<pipe_access Access = pipe_access::read, class T, size_t N>
pipe<T, Access> make_pipe(const pipe_storage<T, N>& ps);

}
```

3.12.3. pipe class specializations

`pipe` class has two distinct specializations depending on `pipe_access` parameter defined as follows:

```
namespace cl
{
template <class T, pipe_access Access = pipe_access::read>
struct pipe: marker_type
{
    typedef T element_type;
    static constexpr pipe_access access = Access;

    template<memory_scope S>
    struct reservation: marker_type
    {
        reservation() = delete;
        reservation(const reservation&) = default;
        reservation(reservation&&) = default;

        reservation& operator=(const reservation&) = delete;
        reservation& operator=(reservation&&) = delete;
        reservation* operator&() = delete;

        bool is_valid() const noexcept;
        bool read(uint index, T& ref) const noexcept;
        void commit() noexcept;
        explicit operator bool() const noexcept;
    };

    pipe() = delete;
    pipe(const pipe&) = default;
    pipe(pipe&&) = default;

    pipe& operator=(const pipe&) = delete;
    pipe& operator=(pipe&&) = delete;
    pipe* operator&() = delete;

    bool read(T& ref) const noexcept;
    reservation<memory_scope_work_item> reserve(
                                           uint num_packets) const noexcept;
    reservation<memory_scope_work_group> work_group_reserve(
                                           uint num_packets) const noexcept;
    reservation<memory_scope_sub_group> sub_group_reserve(
                                           uint num_packets) const noexcept;

    uint num_packets() const noexcept;
    uint max_packets() const noexcept;
};

template <class T>
struct pipe<T, pipe_access::write>: marker_type
{
    typedef T element_type;
    static constexpr pipe_access access = pipe_access::write;

    template<memory_scope S>
    struct reservation: marker_type
    {
        reservation() = delete;
        reservation(const reservation &) = default;
        reservation(reservation &&) = default;

        reservation& operator=(const reservation &) noexcept = delete;
        reservation& operator=(reservation &&) noexcept = delete;
        reservation* operator&() = delete;

        bool is_valid() const noexcept;
        bool write(uint index, const T& ref) noexcept;
        void commit() noexcept;

        explicit operator bool() const noexcept;
    };

    pipe() = delete;
    pipe(const pipe&) = default;
    pipe(pipe&&) = default;
    pipe& operator=(const pipe&) = delete;
    pipe& operator=(pipe&&) = delete;
    pipe* operator&() = delete;

    bool write(const T& ref) noexcept;
    reservation<memory_scope_work_item> reserve(uint num_packets) noexcept;
    reservation<memory_scope_work_group> work_group_reserve(
                                                  uint num_packets) noexcept;
    reservation<memory_scope_sub_group> sub_group_reserve(
                                                  uint num_packets) noexcept;

    uint num_packets() const noexcept;
    uint max_packets() const noexcept;
};

}
```

3.12.4. pipe class methods

pipe::read

```
bool read(T& ref) const noexcept;
```

Read packet from pipe into `ref`.

Returns `true` if read is successful and `false` if the pipe is empty.

pipe::write

```
bool write(const T& ref) noexcept;
```

Write packet specified by `ref` to pipe. Returns `true` if write is successful and `false` if the pipe is full.

pipe::reserve

```
reservation reserve(uint num_packets) const noexcept;

reservation reserve(uint num_packets) noexcept;
```

Reserve `num_packets` entries for reading/writing from/to pipe. Returns a valid reservation if the reservation is successful.

The reserved pipe entries are referred to by indices that go from `0 … num_packets - 1`.

pipe::work_group_reserve

```
reservation work_group_reserve(uint num_packets) const noexcept;

reservation work_group_reserve(uint num_packets) noexcept;
```

Reserve `num_packets` entries for reading/writing from/to pipe. Returns a valid reservation if the reservation is successful.

The reserved pipe entries are referred to by indices that go from `0 … num_packets - 1`.

pipe::sub_group_reserve

```
reservation sub_group_reserve(uint num_packets) const noexcept;

reservation sub_group_reserve(uint num_packets) noexcept;
```

Reserve `num_packets` entries for reading/writing from/to pipe. Returns a valid reservation if the reservation is successful.

The reserved pipe entries are referred to by indices that go from `0 … num_packets - 1`.

pipe::num_packets

```
uint num_packets() const noexcept;
```

Returns the current number of packets that have been written to the pipe, but have not yet been read from the pipe. The number of available entries in a pipe is a dynamic value. The value returned should be considered immediately stale.

pipe::max_packets

```
uint max_packets() const noexcept;
```

Returns the maximum number of packets specified when pipe was created.

pipe::reservation::read

```
bool pipe::reservation::read(uint index, T& ref) const noexcept;
```

Read packet from the reserved area of the pipe referred to by `index` into `ref`.

The reserved pipe entries are referred to by indices that go from `0 … num_packets - 1`.

Returns `true` if read is successful and `false` otherwise.

pipe::reservation::write

```
bool pipe::reservation::write(uint index, const T& ref) noexcept;
```

Write packet specified by `ref` to the reserved area of the pipe referred to by `index`.

The reserved pipe entries are referred to by indices that go from `0 … num_packets - 1`.

Returns `true` if write is successful and `false` otherwise.

pipe::reservation::commit

```
void pipe::reservation::commit() const noexcept;

void pipe::reservation::commit() noexcept;
```

Indicates that all reads/writes to `num_packets` associated with reservation are completed.

pipe::reservation::is_valid

```
bool pipe::reservation::is_valid();
```

Return `true` if reservation is a valid reservation and `false` otherwise.

pipe::reservation::operator bool

```
explicit pipe::reservation::operator bool() const noexcept;
```

Return `true` if reservation is a valid reservation and `false` otherwise.

3.12.5. pipe_storage class

The lifetime of `pipe_storage` objects is the same as a program where they were declared. The variables of such type are not shared across devices.

`N` in the `pipe_storage` template class specifies the maximum number of packets which can be held by an object.

```
namespace cl
{
template<class T, size_t N>
struct pipe_storage: marker_type
{
    pipe_storage();
    pipe_storage(const pipe_storage&) = default;
    pipe_storage(pipe_storage&&) = default;

    pipe_storage& operator=(const pipe_storage&) = delete;
    pipe_storage& operator=(pipe_storage&&) = delete;
    pipe_storage* operator&() = delete;

    template<pipe_access Access = pipe_access::read>
    pipe<T, Access> get() const noexcept
};

template<pipe_access Access = pipe_access::read, class T, size_t N>
pipe<T, Access> make_pipe(const pipe_storage<T, N>& ps);

}
```

3.12.6. pipe_storage class methods and make_pipe function

pipe_storage::get

```
template<pipe_access Access = pipe_access::read>
pipe<T, Access> get() noexcept;
```

Constructs a read only or write only pipe from `pipe_storage` object. One kernel can have only one pipe accessor associated with one `pipe_storage` object.

make_pipe

```
template<pipe_access Access = pipe_access::read, class T, size_t N>
pipe<T, Access> make_pipe(const pipe_storage<T, N>& ps);
```

Constructs a read only or write only pipe from `pipe_storage` object. One kernel can have only one pipe accessor associated with one `pipe_storage` object.

3.12.7. Operations ordering using reservations

The `reservation::read` and `reservation::write` pipe functions can be used to read from or write to a packet index. These functions 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 `reservation::write` method, the contents of that packet in the pipe are undefined. `reservation::commit` remove the entries reserved for reading from the pipe. `reservation::commit` 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 `CL_DEVICE_PIPE_MAX_ACTIVE_RESERVATIONS` (refer to *Table 4.3*) reservations active (i.e. reservations 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.

3.12.8. Requirements

Data

Template parameter `T` in `pipe` and `pipe_storage` class template denotes the data type stored in pipe. The type `T` must be a POD type i.e. satisfy `is_pod<T>::value == true`.

Work-group operations

All work-group specific 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.

Sub-group operations

All sub-group specific functions must be encountered by all work items in a sub-group executing the kernel with the same argument values, otherwise the behavior is undefined.

3.12.9. Restrictions

pipe

- The `pipe` type cannot be used with variables declared inside a class or union field, a pointer type, an array, global variables declared at program scope or the return type of a function.
- A kernel cannot read from and write to the same pipe object.
- The `pipe` type cannot be used with the `global`, `local`, `priv` and `constant` address space storage classes (see the *Explicit address space storage classes* section).
- The value returned by applying the `sizeof` operator to the `pipe` type is implementation-defined.

reservation

- The `reservation` type cannot be used with variables declared inside a class or union field, a pointer type, an array, global variables declared at program scope or the return type of a function.

- The reservation type cannot be used with the global, local, priv and constant address space storage classes (see the *Explicit address space storage classes* section).
- The value returned by applying the sizeof operator to the reservation type is implementation-defined.

The following behavior is undefined:

- A kernel calls reservation::read or reservation::write with a valid reservation but with an index that is not a value from 0 … num_packets - 1 specified to the corresponding call to pipe::reserve, pipe::work_group_reserve or pipe::sub_group_reserve.
- A kernel calls reservation::read or reservation::write with a reservation that has already been committed (i.e. a reservation::commit with this reservation has already been called).
- The contents of the reserved data packets in the pipe are undefined if the kernel does not call reservation::write for all entries that were reserved by the corresponding call to pipe::reserve, pipe::work_group_reserve or pipe::sub_group_reserve.
- Calls to reservation::read and reservation::commit or reservation::write and reservation::commit for a given reservation must be called by the same kernel that made the reservation using pipe::reserve, pipe::work_group_reserve or pipe::sub_group_reserve. The reservation cannot be passed to another kernel including child kernels.

pipe_storage

- Variables of type pipe_storage can only be declared at program scope or with the static specifier.
- The pipe_storage type cannot be used as a class or union field, a pointer type, an array or the return type of a function.
- The pipe_storage type cannot be used with the global, local, priv and constant address space storage classes (see the *Explicit address space storage classes* section).
- The value returned by applying the sizeof operator to the pipe_storage type is implementation-defined.
- Variables of type pipe created from pipe_storage can only be declared inside a kernel function at kernel scope.

The following behavior is undefined:

- A kernel cannot contain more than one pipe accessor made from one pipe_storage object. Otherwise behavior is undefined.

3.12.10. Examples

Example 1

Example of reading from a pipe object.

```
#include <opencl_pipe>
using namespace cl;

kernel void reader(pipe<int> p) {
    int val;
    if(p.read(val)) {
        //...
    }
}
```

Example 2

Example of writing to a pipe object.

```
#include <opencl_pipe>
using namespace cl;

kernel void writer(pipe<int, pipe_access::write> p) {
    //...
    int val;
    if(p.write(val)) {
        //...
    }
}
```

Example 3

Example of reading from a pipe object using reservations.

```
#include <opencl_pipe>
using namespace cl;

kernel void reader(pipe<int, pipe_access::read> p) {
    int val;
    auto rid = p.reserve(1);
    if(rid.read(0, val)) {
        //...
    }
    rid.commit();
}
```

Example 4

Example of using a pipe_storage object and how to create the pipe objects/accessors from it.

```
#include <opencl_pipe>

cl::pipe_storage <int, 100> myProgramPipe0;

kernel void producer() {
    cl::pipe<int, cl::pipe_access::write> p =
   myProgramPipe0.get<cl::pipe_access::write>();
    //...
    p.write(...);
}

kernel void consumer() {
    cl::pipe<int, cl::pipe_access::read> p =
  myProgramPipe0.get<cl::pipe_access::read>();
    if(p.read(...)) {
        //...
    }
}
```

Example 5

Example of using more than one pipe_storage object.

```
#include <opencl_pipe>
using namespace cl;

pipe_storage<int2, 20> myProgramPipe2;
pipe_storage<float, 40> myProgramPipe3;

kernel void input() {
    auto p = make_pipe<pipe_access::write>(myProgramPipe2);
    //...
    p.write(...);
}

kernel void processor() {
    auto p_in = make_pipe<pipe_access::read>(myProgramPipe2);
    auto p_out = make_pipe<pipe_access::write>(myProgramPipe3);
    ...
    if(p_in.read(...)) {
        //...
    }
    p_out.write(...);
}

kernel void output() {
    auto p = make_pipe<pipe_access::read>(myProgramPipe3);
    if(p.read(...)) {
        //...
    }
}
```

3.13. Device Enqueue Library

OpenCL C++ device enqueue functionality allows a kernel to enqueue the same device, without host interaction. A kernel may enqueue code represented by lambda syntax, and control execution order with event dependencies including user events and markers. `device_queue` follows all the rules for marker types as specified in the *Marker Types* section.

3.13.1. Queue Host Type

The below table describes the OpenCL queue data type and the corresponding data type available to the application:

Table 17. Host queue type

Type in OpenCL C++	API type for application
`cl::device_queue`	`cl_queue`

3.13.2. Header <opencl_device_queue> Synopsis

```
namespace cl
{
enum class enqueue_status;
enum class enqueue_policy;
enum class event_status;
enum class event_profiling_info;

struct event
{
    event();
    event(const event&) = default;
    event(event&) = default;

    event& operator=(const event&) = default;
    event& operator=(event&&) = default;

    bool is_valid() const noexcept;
    void retain() noexcept;
    void release() noexcept;

    explicit operator bool() const noexcept;

    void set_status(event_status status) noexcept;
    void profiling_info(event_profiling_info name,
                        global_ptr<long> value) noexcept;
};

event make_user_event();

struct ndrange
{
    explicit ndrange(size_t global_work_size) noexcept;
    ndrange(size_t global_work_size,
            size_t local_work_size) noexcept;
    ndrange(size_t global_work_offset,
            size_t global_work_size,
            size_t local_work_size) noexcept;

    template <size_t N>
    ndrange(const size_t (&global_work_size)[N]) noexcept;
    template <size_t N>
    ndrange(const size_t (&global_work_size)[N],
            const size_t (&local_work_size)[N]) noexcept;
    template <size_t N>
    ndrange(const size_t (&global_work_offset)[N],
            const size_t (&global_work_size)[N],
            const size_t (&local_work_size)[N]) noexcept;
};
```

```
struct device_queue: marker_type
{
    device_queue() noexcept = delete;
    device_queue(const device_queue&) = default;
    device_queue(device_queue&&) = default;

    device_queue& operator=(const device_queue&) = delete;
    device_queue& operator=(device_queue&&) = delete;
    device_queue* operator&() = delete;

    template <class Fun, class... Args>
    enqueue_status enqueue_kernel(enqueue_policy flag,
                                  const ndrange &ndrange,
                                  Fun fun,
                                  Args... args) noexcept;

    template <class Fun, class... Args>
    enqueue_status enqueue_kernel(enqueue_policy flag,
                                  uint num_events_in_wait_list,
                                  const event *event_wait_list,
                                  event *event_ret,
                                  const ndrange &ndrange,
                                  Fun fun,
                                  Args... args) noexcept;

    enqueue_status enqueue_marker(uint num_events_in_wait_list,
                                  const event *event_wait_list,
                                  event *event_ret) noexcept;
};

device_queue get_default_device_queue();

template <class Fun, class... Args>
uint get_kernel_work_group_size(Fun fun, Args... args);
template <class Fun, class... Args>
uint get_kernel_preferred_work_group_size_multiple(Fun fun,
                                                   Args... args);
template <class Fun, class... Args>
uint get_kernel_sub_group_count_for_ndrange(const ndrange &ndrange,
                                            Fun fun,
                                            Args... args);
template <class Fun, class... Args>
uint get_kernel_max_sub_group_size_for_ndrange(const ndrange &ndrange,
                                               Fun fun,
                                               Args... args);
template <class Fun, class... Args>
uint get_kernel_local_size_for_sub_group_count(uint num_sub_groups,
                                               Fun fun,
                                               Args... args);
template <class Fun, class... Args>
```

```
uint get_kernel_max_num_sub_groups(Fun fun, Args... args);

}
```

3.13.3. device_queue class methods

`device_queue` object represents work queue of the device. Device queue meets all requirements of the marker types as in the *Marker Types* section.

device_queue::enqueue_kernel

```
template <class Fun, class... Args>
enqueue_status enqueue_kernel(enqueue_policy policy,
                              const ndrange &ndrange,
                              Fun fun,
                              Args... args) noexcept;
```

This method allows to enqueue functor or lambda `fun` on the device with specified `policy` over the specified `ndrange`.

`args` are the arguments that will be passed to `fun` when kernel will be enqueued with the exception for `local_ptr` parameters. For local pointers user must supply the size of local memory that will be allocated using `local_ptr<T>::size_type\{num elements}`. Please see examples how to use `enqueue_kernel` are in the *Examples* section.

```
template <class Fun, class... Args>
enqueue_status enqueue_kernel(enqueue_policy policy,
                              uint num_events_in_wait_list,
                              const event *event_wait_list,
                              event *event_ret,
                              const ndrange &ndrange,
                              Fun fun,
                              Args... args) noexcept;
```

This method enqueues functor or lambda `fun` in the same way as the overload above with the exception for the passed event list. If an event is returned, `enqueue_kernel` performs an implicit retain on the returned event.

device_queue::enqueue_marker

```
enqueue_status enqueue_marker(uint num_events_in_wait_list,
                              const event *event_wait_list,
                              event *event_ret) noexcept;
```

This method enqueues a marker to device 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 `nullptr` as otherwise this is a no-op.

If an event is returned, `enqueue_marker` performs an implicit retain on the returned event.

3.13.4. event class methods

event::is_valid

```
bool is_valid() const noexcept;
```

Returns `true` if event object is a valid event. Otherwise returns `false`.

event::operator bool

```
explicit operator bool() const noexcept;
```

Returns `true` if event object is a valid event. Otherwise returns `false`.

event::retain

```
void retain() noexcept;
```

Increments the event reference count. Event must be an event returned by `enqueue_kernel` or `enqueue_marker` or a user event.

event::release

```
void release() noexcept;
```

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.

event::set_status

```
void set_status(event_status status) noexcept;
```

Sets the execution status of a user event. Event must be a user event. `status` can be either `event_status::complete` or `event_status::error` value indicating an error.

event::profiling_info

```
void profiling_info(event_profiling_info name,
                    global_ptr<long> value) noexcept;
```

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:

- `event_profiling_info::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_PROFILING_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_COMMAND_START` for the command identified by event in nanoseconds.

`profiling_info` when called multiple times for the same event is undefined.

3.13.5. Other operations

get_default_device_queue

```
device_queue get_default_device_queue();
```

Returns the default device queue. If a default device queue has not been created, `device_queue::is_valid()` will return `false`.

make_user_event

```
event make_user_event();
```

Creates a user event. Returns the user event. The execution status of the user event created is set to `event_status::submitted`.

get_kernel_work_group_size

```
template <class Fun, class... Args>
uint get_kernel_work_group_size(Fun fun, Args... args);
```

This provides a mechanism to query the maximum work-group size that can be used to execute a functor on the current device.

fun specifies the functor representing the kernel code that would be enqueued.

args are the arguments that will be passed to fun when kernel will be enqueued with the exception for local_ptr parameters. For local pointers user must supply the size of local memory that will be allocated.

get_kernel_preferred_work_group_size_multiple

```
template <class Fun, class... Args>
uint get_kernel_preferred_work_group_size_multiple(Fun fun,
                                                   Args... args);
```

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 will not fail to enqueue the functor for execution unless the work-group size specified is larger than the device maximum.

fun specifies the functor representing the kernel code that would be enqueued.

args are the arguments that will be passed to fun when kernel will be enqueued with the exception for local_ptr parameters. For local pointers user must supply the size of local memory that will be allocated.

get_kernel_sub_group_count_for_ndrange

```
template <class Fun, class... Args>
uint get_kernel_sub_group_count_for_ndrange(const ndrange &ndrange,
                                            Fun fun,
                                            Args... args);
```

Returns the number of sub-groups in each work-group of the dispatch (except for the last in cases where the global size does not divide cleanly into work-groups) given the combination of the passed ndrange and functor.

fun specifies the functor representing the kernel code that would be enqueued.

args are the arguments that will be passed to fun when kernel will be enqueued with the exception for local_ptr parameters. For local pointers user must supply the size of local memory that will be allocated.

get_kernel_max_sub_group_size_for_ndrange

```
template <class Fun, class... Args>
uint get_kernel_max_sub_group_size_for_ndrange(const ndrange &ndrange,
                                               Fun fun,
                                               Args... args);
```

Returns the maximum sub-group size for a functor.

`fun` specifies the functor representing the kernel code that would be enqueued.

`args` are the arguments that will be passed to `fun` when kernel will be enqueued with the exception for `local_ptr` parameters. For local pointers user must supply the size of local memory that will be allocated.

get_kernel_local_size_for_sub_group_count

```
template <class Fun, class... Args>
uint get_kernel_local_size_for_sub_group_count(uint num_sub_groups,
                                               Fun fun,
                                               Args... args);
```

Returns a valid local size that would produce the requested number of sub-groups such that each sub-group is complete with no partial sub-groups.

`fun` specifies the functor representing the kernel code that would be enqueued.

`args` are the arguments that will be passed to `fun` when kernel will be enqueued with the exception for `local_ptr` parameters. For local pointers user must supply the size of local memory that will be allocated.

get_kernel_max_num_sub_groups

```
template <class Fun, class... Args>
uint get_kernel_max_num_sub_groups(Fun fun, Args... args);
```

Provides a mechanism to query the maximum number of sub-groups that can be used to execute the passed functor on the current device.

`fun` specifies the functor representing the kernel code that would be enqueued.

`args` are the arguments that will be passed to `fun` when kernel will be enqueued with the exception for `local_ptr` parameters. For local pointers user must supply the size of local memory that will be allocated.

3.13.6. ndrange

`ndrange` type is used to represent the size of the enqueued workload. The dimension of the workload ranges from 1 to 3. User can specify global work size, local work size and global work offset. Unspecified parameters are defaulted to 0.

3.13.7. Enqueue policy

```
enum class enqueue_policy
{
    no_wait,
    wait_kernel,
    wait_work_group
};
```

Enqueue policy enumerable is used to specify launch policy of enqueued kernel. It is defined as follows:

Table 18. Enqueue policy

Policy	Description
`no_wait`	Indicates that the enqueued kernels do not need to wait for the parent kernel to finish execution before they begin execution.
`wait_kernel`	Indicates that all work-items of the parent kernel must finish executing and all immediate [14] side effects committed before the enqueued child kernel may begin execution.
`wait_work_group` [15]	Indicates that the enqueued kernels wait only for the work-group that enqueued the kernels to finish before they begin execution.

3.13.8. Enqueue status

```
enum class enqueue_status
{
    success,
    failure,
    invalid_queue,
    invalid_ndrange,
    invalid_event_wait_list,
    queue_full,
    invalid_arg_size,
    event_allocation_failure,
    out_of_resources
};
```

The `enqueue_kernel` and `enqueue_marker` methods return `enqueue_status::success` if the command is enqueued successfully and return `enqueue_status::failure` otherwise. If the *-g* compile option is specified in compiler options passed to `clBuildProgram`, the other errors may be returned instead of `enqueue_status::failure` to indicate why `enqueue_kernel` or `enqueue_marker` failed. Enqueue status is defined as follows:

Table 19. Enqueue status

Status	Description
success	
failure	
invalid_queue	*queue* is not a valid device queue.
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.
invalid_event_wait_list	If event_wait_list is nullptr and num_events_in_wait_list > 0, or if event_wait_list is not nullptr and num_events_in_wait_list is 0, or if event objects in event_wait_list are not valid events.
queue_full	If *queue* is full.
invalid_arg_size	If size of local memory arguments is 0.
event_allocation_failure	If event_ret is not nullptr and an event could not be allocated.
out_of_resources	If there is a failure to queue the kernel in *queue* because of insufficient resources needed to execute the kernel.

3.13.9. Event status

```
enum class event_status
{
    submitted,
    complete,
    error,
};
```

Event status enumerable is used to set a user event status. It is defined as follows:

Table 20. Event status

Status	Description
submitted	Initial status of a user event
complete	
error	Status indicating an error

3.13.10. 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 event_status::complete if this kernel and all its child kernels finish execution successfully. The execution status of the kernel will be `event_status::error` 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 `event_status::complete` only after `A` and the kernels `A` enqueued (and any kernels these enqueued kernels enqueue and so on) have finished execution.

3.13.11. Event profiling info

```
enum class event_profiling_info
{
    exec_time,
};
```

Event profiling info enumerable is used to determine the outcome of `event::profiling_info` function. It is defined as follows:

Table 21. Event profiling info

Status	Description
`exec_time`	Identifies profiling information to be queried. If specified, the two 64-bit values are returned The first 64-bit value describes the elapsed time: `CL_PROFILING_COMMAND_END - CL_PROFILING_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_COMMAND_START` for the command identified by event in nanoseconds.

3.13.12. Requirements

Functor and lambda objects passed to `enqueue_kernel` method of device queue has to follow specific restrictions:

- It has to be trivially copyable.
- It has to be trivially copy constructible.
- It has to be trivially destructible.

Code enqueuing function objects that do not meet this criteria is ill-formed.

Pointers, references and marker types

Functors that are enqueued cannot have any reference and pointer fields, nor can have fields of marker types. If object containing such fields is enqueued the behavior is undefined.

The same restrictions apply to *capture-list* variables in lambda.

> This requirements are caused by the fact that kernel may be enqueued after parent kernel terminated all pointers and references will be invalidated by then.

Events

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 `event_wait_list` argument to various `clEnqueue`* APIs or runtime APIs that take events as arguments such as `clRetainEvent`, `clReleaseEvent`, `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 `make_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 `event::set_status` method.

3.13.13. Restrictions

- The `device_queue` type cannot be used with variables declared inside a class or union field, a pointer type, an array, global variables declared at program scope or the return type of a function.
- The `event` type cannot be used with variables declared inside a class or union field, global variables declared at program scope or the return type of a function.
- The `event` and `device_queue` type cannot be used with the `global`, `local`, `priv` and `constant` address space storage classes (see the *Explicit address space storage classes* section).
- The values returned by applying the `sizeof` operator to the `device_queue` and `event` types is implementation-defined.

3.13.14. Examples

Example 1

```
#include <opencl_device_queue>
#include <opencl_work_item>
using namespace cl;

kernel void foo() {
    auto q = get_default_device_queue();
    q.enqueue_kernel(enqueue_policy::no_wait,
                     ndrange( 1 ),
                     [](){ uint tid = get_global_id(0); } );
}
```

Example 2

Example of using the explicit local pointer class with `enqueue_kernel` method.

```
#include <opencl_device_queue>
#include <opencl_work_item>
using namespace cl;

kernel void foo(device_queue q) {
    auto lambda = [](local_ptr<ushort16> p) { uint tid = get_global_id(0); };
    q.enqueue_kernel(enqueue_policy::no_wait,
                     ndrange{1},
                     lambda,
                     local_ptr<ushort16>::size_type{1});
}
```

Example 3

Example of enqueuing a functor to a `device_queue` object.

```
#include <opencl_device_queue>
#include <opencl_work_item>
using namespace cl;

struct Lambda {
    template <typename T>
    void operator ()(T t) const { uint tid = get_global_id(0); }
};

kernel void foo(device_queue q) {
    auto lambda = Lambda{};
    q.enqueue_kernel(enqueue_policy::no_wait,
                     ndrange{1},
                     lambda,
                     local_ptr<ushort16>::size_type{1});
}
```

3.14. Work-Item Functions

This section describes the library of 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.

3.14.1. Header <opencl_work_item> Synopsis

```
namespace cl
{
uint get_work_dim();
size_t get_global_size(uint dimindx);
size_t get_global_id(uint dimindx);
size_t get_local_size(uint dimindx);
size_t get_enqueued_local_size(uint dimindx);
size_t get_local_id(uint dimindx);
size_t get_num_groups(uint dimindx);
size_t get_group_id(uint dimindx);
size_t get_global_offset(uint dimindx);
size_t get_global_linear_id();
size_t get_local_linear_id();
size_t get_sub_group_size();
size_t get_max_sub_group_size();
size_t get_num_sub_groups();
size_t get_enqueued_num_sub_groups();
size_t get_sub_group_id();
size_t get_sub_group_local_id();

}
```

3.14.2. Work item operations

get_work_dim

```
uint get_work_dim()
```

Returns the number of dimensions in use. This is the value given to the `work_dim` argument specified in `clEnqueueNDRangeKernel`.

get_global_size

```
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`.

get_global_id

```
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`.

get_local_size

```
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 [19], 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`.

get_enqueued_local_size

```
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`.

get_local_id

```
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.

get_num_groups

```
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.

get_group_id

```
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.

get_global_offset

```
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.

get_global_linear_id

```
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)).

get_local_linear_id

```
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)`

get_sub_group_size

```
size_t get_sub_group_size()
```

Returns the number of work-items in the sub-group. This value is no more than the maximum sub-group size and is implementation-defined based on a combination of the compiled kernel and the dispatch dimensions. This will be a constant value for the lifetime of the sub-group.

get_max_sub_group_size

```
size_t get_max_sub_group_size()
```

Returns the maximum size of a sub-group within the dispatch. This value will be invariant for a given set of dispatch dimensions and a kernel object compiled for a given device.

get_num_sub_groups

```
size_t get_num_sub_groups()
```

Returns the number of sub-groups that the current work-group is divided into.

This number will be constant for the duration of a work-group's execution. If the kernel is executed with a non-uniform work-group size [17] values for any dimension, calls to this built-in from some work-groups may return different values than calls to this built-in from other work-groups.

get_enqueued_num_sub_groups

```
size_t get_enqueued_num_sub_groups()
```

Returns the same value as that returned by `get_num_sub_groups()` 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 sub groups in each of the work groups that make up the uniform region of the global range.

get_sub_group_id

```
size_t get_sub_group_id()
```

`get_sub_group_id()` returns the sub-group ID which is a number from `0` … `get_num_sub_groups()-1`.

For `clEnqueueTask`, this returns `0`.

get_sub_group_local_id

```
size_t get_sub_group_local_id()
```

Returns the unique work-item ID within the current sub-group. The mapping from `get_local_id(dimindx)` to `get_sub_group_local_id()` will be invariant for the lifetime of the work-group.

3.15. Work-group Functions

The OpenCL C++ library implements the following 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.

Here `gentype` matches: `int`, `uint`, `long`, `ulong`, `float`, `half` [4] or `double` [18].

3.15.1. Header <opencl_work_group> Synopsis

```
namespace cl
{
enum class work_group_op { add, min, max };

//logical operations
bool work_group_all(bool predicate);
bool work_group_any(bool predicate);
bool sub_group_all(bool predicate);
bool sub_group_any(bool predicate);

//broadcast functions
int work_group_broadcast(int a, size_t local_id);
```

```
uint work_group_broadcast(uint a, size_t local_id);
long work_group_broadcast(long a, size_t local_id);
ulong work_group_broadcast(ulong a, size_t local_id);
float work_group_broadcast(float a, size_t local_id);
#ifdef cl_khr_fp16
half work_group_broadcast(half a, size_t local_id);
#endif
#ifdef cl_khr_fp64
double work_group_broadcast(double a, size_t local_id);
#endif

int work_group_broadcast(int a, size_t local_id_x, size_t local_id_y);
uint work_group_broadcast(uint a, size_t local_id_x, size_t local_id_y);
long work_group_broadcast(long a, size_t local_id_x, size_t local_id_y);
ulong work_group_broadcast(ulong a, size_t local_id_x, size_t local_id_y);
float work_group_broadcast(float a, size_t local_id_x, size_t local_id_y);
#ifdef cl_khr_fp16
half work_group_broadcast(half a, size_t local_id_x, size_t local_id_y);
#endif
#ifdef cl_khr_fp64
double work_group_broadcast(double a, size_t local_id_x, size_t local_id_y);
#endif

int work_group_broadcast(int a, size_t local_id_x, size_t local_id_y,
                         size_t local_id_z);
uint work_group_broadcast(uint a, size_t local_id_x, size_t local_id_y,
                          size_t local_id_z);
long work_group_broadcast(long a, size_t local_id_x, size_t local_id_y,
                          size_t local_id_z);
ulong work_group_broadcast(ulong a, size_t local_id_x, size_t local_id_y,
                           size_t local_id_z);
float work_group_broadcast(float a, size_t local_id_x, size_t local_id_y,
                           size_t local_id_z);
#ifdef cl_khr_fp16
half work_group_broadcast(half a, size_t local_id_x, size_t local_id_y,
                          size_t local_id_z);
#endif
#ifdef cl_khr_fp64
double work_group_broadcast(double a, size_t local_id_x, size_t local_id_y,
                            size_t local_id_z);
#endif

int sub_group_broadcast(int a, size_t sub_group_local_id);
uint sub_group_broadcast(uint a, size_t sub_group_local_id);
long sub_group_broadcast(long a, size_t sub_group_local_id);
ulong sub_group_broadcast(ulong a, size_t sub_group_local_id);
float sub_group_broadcast(float a, size_t sub_group_local_id);
#ifdef cl_khr_fp16
half sub_group_broadcast(half a, size_t sub_group_local_id);
#endif
#ifdef cl_khr_fp64
```

```
double sub_group_broadcast(double a, size_t sub_group_local_id);
#endif

//numeric operations
template <work_group_op op> int work_group_reduce(int x);
template <work_group_op op> uint work_group_reduce(uint x);
template <work_group_op op> long work_group_reduce(long x);
template <work_group_op op> ulong work_group_reduce(ulong x);
template <work_group_op op> float work_group_reduce(float x);
#ifdef cl_khr_fp16
template <work_group_op op> half work_group_reduce(half x);
#endif
#ifdef cl_khr_fp64
template <work_group_op op> double work_group_reduce(double x);
#endif

template <work_group_op op> int work_group_scan_exclusive(int x);
template <work_group_op op> uint work_group_scan_exclusive(uint x);
template <work_group_op op> long work_group_scan_exclusive(long x);
template <work_group_op op> ulong work_group_scan_exclusive(ulong x);
template <work_group_op op> float work_group_scan_exclusive(float x);
#ifdef cl_khr_fp16
template <work_group_op op> half work_group_scan_exclusive(half x);
#endif
#ifdef cl_khr_fp64
template <work_group_op op> double work_group_scan_exclusive(double x);
#endif

template <work_group_op op> int work_group_scan_inclusive(int x);
template <work_group_op op> uint work_group_scan_inclusive(uint x);
template <work_group_op op> long work_group_scan_inclusive(long x);
template <work_group_op op> ulong work_group_scan_inclusive(ulong x);
template <work_group_op op> float work_group_scan_inclusive(float x);
#ifdef cl_khr_fp16
template <work_group_op op> half work_group_scan_inclusive(half x);
#endif
#ifdef cl_khr_fp64
template <work_group_op op> double work_group_scan_inclusive(double x);
#endif

template <work_group_op op> int sub_group_reduce(int x);
template <work_group_op op> uint sub_group_reduce(uint x);
template <work_group_op op> long sub_group_reduce(long x);
template <work_group_op op> ulong sub_group_reduce(ulong x);
template <work_group_op op> float sub_group_reduce(float x);
#ifdef cl_khr_fp16
template <work_group_op op> half sub_group_reduce(half x);
#endif
#ifdef cl_khr_fp64
template <work_group_op op> double sub_group_reduce(double x);
#endif
```

```
template <work_group_op op> int sub_group_scan_exclusive(int x);
template <work_group_op op> uint sub_group_scan_exclusive(uint x);
template <work_group_op op> long sub_group_scan_exclusive(long x);
template <work_group_op op> ulong sub_group_scan_exclusive(ulong x);
template <work_group_op op> float sub_group_scan_exclusive(float x);
#ifdef cl_khr_fp16
template <work_group_op op> half sub_group_scan_exclusive(half x);
#endif
#ifdef cl_khr_fp64
template <work_group_op op> double sub_group_scan_exclusive(double x);
#endif

template <work_group_op op> int sub_group_scan_inclusive(int x);
template <work_group_op op> uint sub_group_scan_inclusive(uint x);
template <work_group_op op> long sub_group_scan_inclusive(long x);
template <work_group_op op> ulong sub_group_scan_inclusive(ulong x);
template <work_group_op op> float sub_group_scan_inclusive(float x);
#ifdef cl_khr_fp16
template <work_group_op op> half sub_group_scan_inclusive(half x);
#endif
#ifdef cl_khr_fp64
template <work_group_op op> double sub_group_scan_inclusive(double x);
#endif

}
```

3.15.2. Logical operations

work_group_all

```
bool work_group_all(bool predicate)
```

Evaluates `predicate` for all work-items in the work-group and returns `true` if `predicate` evaluates to `true` for all work-items in the work-group.

work_group_any

```
bool work_group_any(bool predicate)
```

Evaluates `predicate` for all work-items in the work-group and returns `true` if `predicate` evaluates to `true` for any work-items in the work-group.

sub_group_all

```
bool sub_group_all(bool predicate)
```

Evaluates `predicate` for all work-items in the sub-group and returns `true` value if `predicate` evaluates to `true` for all work-items in the sub-group.

sub_group_any

```
bool sub_group_any(bool predicate)
```

Evaluates `predicate` for all work-items in the sub-group and returns `true` value if `predicate` evaluates to `true` for any work-items in the sub-group.

Example:

```
#include <opencl_work_item>
#include <opencl_work_group>
using namespace cl;

kernel void foo(int *p) {
    //...
    bool check = work_group_all(p[get_local_id(0)] == 0);
}
```

In this case `work_group_all` would return `true` for all work-items in work-group if all elements in `p`, in range specified by work-group's size, are `true`.

One could achieve similar result by using analogical call to `work_group_any`:

```
#include <opencl_work_item>
#include <opencl_work_group>
using namespace cl;

kernel void foo(int *p) {
    //...
    bool check = !work_group_any(p[get_local_id(0)] != 0);
}
```

3.15.3. Broadcast functions

work_group_broadcast

```
gentype work_group_broadcast(gentype a,
                             size_t local_id);

gentype work_group_broadcast(gentype a,
                             size_t local_id_x,
                             size_t local_id_y);

gentype work_group_broadcast(gentype a,
                                            size_t local_id_x,
                                            size_t local_id_y,
                                            size_t local_id_z);
```

Broadcast the value of `a` 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.

sub_group_broadcast

```
gentype sub_group_broadcast(gentype a,
                                           size_t sub_group_local_id);
```

Broadcast the value of `a` for work-item identified by `sub_group_local_id` (value returned by `get_sub_group_local_id`) to all work-items in the sub-group.

`sub_group_local_id` must be the same value for all work-items in the sub-group.

Example:

```
#include <opencl_work_item>
#include <opencl_work_group>
using namespace cl;

kernel void foo(int *p) {
   //...
   int broadcasted_value = work_group_broadcast(p[get_local_id(0)], 0);
}
```

Here we are broadcasting value passed to `work_group_broadcast` function by work-item with `local_id = 0` (which is `p[0]`). This function will return `p[0]` for all callers. Please note that `local_id` must be the same for all work-items, therefore something like this is invalid:

```
#include <opencl_work_item>
#include <opencl_work_group>
using namespace cl;

kernel void foo(int *p) {
    //...
    int broadcasted_value =
      work_group_broadcast(p[get_local_id(0)], get_local_id(0));
                    //invalid: second argument has different value
                    // for different work-items in work-group
}
```

3.15.4. Numeric operations

work_group_reduce

```
template <work_group_op op>
gentype work_group_reduce(gentype x);
```

Return result of reduction operation specified by `op` for all values of `x` specified by work-items in a work-group.

work_group_scan_exclusive

```
template <work_group_op op>
gentype work_group_scan_exclusive(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.

work_group_scan_inclusive

```
template <work_group_op op>
gentype work_group_scan_inclusive(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.

sub_group_reduce

```
template <work_group_op op>
gentype sub_group_reduce(gentype x);
```

Return result of reduction operation specified by op for all values of x specified by work-items in a sub-group.

sub_group_scan_exclusive

```
template <work_group_op op>
gentype sub_group_scan_exclusive(gentype x);
```

Do an exclusive scan operation specified by op of all values specified by work-items in a sub-group. The scan results are returned for each work-item.

The scan order is defined by increasing 1D linear global ID within the sub-group.

sub_group_scan_inclusive

```
template <work_group_op op>
gentype sub_group_scan_inclusive(gentype x);
```

Do an inclusive scan operation specified by op of all values specified by work-items in a sub-group. The scan results are returned for each work-item.

The scan order is defined by increasing 1D linear global ID within the sub-group.

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 is work_group_op::add, the identity I is 0. If op is work_group_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 is work_group_op::max, the identity I is INT_MIN, 0, LONG_MIN, 0 and -INF.

Consider the following example:

```
#include <opencl_work_item>
#include <opencl_work_group>
using namespace cl;

void foo(int *p)
{
    ...
    int prefix_sum_val =
                work_group_scan_inclusive<work_group_op::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<work_group_op::add>` with 3 and returns 3. Work-item 1 calls `work_group_scan_inclusive<work_group_op::add>` with 1 and returns 4. The full set of values returned by `work_group_scan_inclusive<work_group_op::add>` for work-items 0 … 7 is [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.

3.16. Synchronization Functions

The OpenCL C++ library implements the following synchronization functions.

3.16.1. Header <opencl_synchronization> Synopsis

```
namespace cl
{
void work_group_barrier(mem_fence flags,
                        memory_scope scope = memory_scope_work_group);
void sub_group_barrier(mem_fence flags,
                       memory_scope scope = memory_scope_work_group);

#ifdef cl_khr_subgroup_named_barrier
struct work_group_named_barrier: marker_type
{
    work_group_named_barrier(uint sub_group_count);
    work_group_named_barrier() = delete;
    work_group_named_barrier(const work_group_named_barrier&) = default;
    work_group_named_barrier(work_group_named_barrier&&) = default;

    work_group_named_barrier& operator=(
                                      const work_group_named_barrier&) = delete;
    work_group_named_barrier& operator=(work_group_named_barrier&&) = delete;
    work_group_named_barrier* operator&() = delete;

    void wait(mem_fence flags,
              memory_scope scope = memory_scope_work_group) const noexcept;
};

#endif

}
```

3.16.2. Synchronization operations

work_group_barrier

```
void work_group_barrier(mem_fence flags,
                        memory_scope scope  = memory_scope_work_group);
```

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 `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 ORed 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.

- `mem_fence::local` - 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`.
- `mem_fence::global` - The `work_group_barrier` function ensure that all global memory accesses become visible to the appropriate scope as given by `scope`.
- `mem_fence::image` - The `work_group_barrier` function will ensure that all image memory accesses become visible to the appropriate scope as given by `scope`. The value of `scope` must be `memory_scope_work_group` or `memory_scope_device`.

`mem_fence::image` cannot be used together with `mem_fence::local` or `mem_fence::global`.

The values of `flags` and `scope` must be the same for all work-items in the work-group.

sub_group_barrier

```
void sub_group_barrier(mem_fence flags,
                       memory_scope scope = memory_scope_work_group);
```

All work-items in a sub-group executing the kernel on a processor must execute this function before any are allowed to continue execution beyond the sub-group barrier. This function must be encountered by all work-items in a sub-group executing the kernel. These rules apply to ND-ranges implemented with uniform and non-uniform work-groups.

If `sub_group_barrier` is inside a conditional statement, then all work-items within the sub-group must enter the conditional if any work-item in the sub-group enters the conditional statement and executes the `sub_group_barrier`.

If `sub_group_barrier` is inside a loop, all work-items within the sub-group must execute the `sub_group_barrier` for each iteration of the loop before any are allowed to continue execution beyond the `sub_group_barrier`.

The `sub_group_barrier` function also queues a memory fence (reads and writes) to ensure correct ordering of memory operations to local or global memory.

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 ORed together the `sub_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.

`mem_fence::local` - The `sub_group_barrier` function will ensure that all local memory accesses become visible to all work-items in the sub-group. Note that the value of `scope` is ignored as the memory scope is always `memory_scope_work_group`.

`mem_fence::global` - The `sub_group_barrier` function ensure that all global memory accesses become visible to the appropriate scope as given by `scope`.

`mem_fence::image` - The `sub_group_barrier` function will ensure that all image memory accesses become visible to the appropriate scope as given by `scope`. The value of `scope` must be `memory_scope_work_group` or `memory_scope_device`.

`mem_fence::image` cannot be used together with `mem_fence::local` or `mem_fence::global`.

The values of `flags` and `scope` must be the same for all work-items in the sub-group.

3.16.3. Named barriers

This section describes the optional **cl_khr_sub_group_named_barrier** extension which exposes the ability to perform barrier synchronization on flexible portions of an execution domain.

If the **cl_khr_sub_group_named_barrier** extension is supported by a device, an application that wants to use it will need to define the `cl_khr_sub_group_named_barrier` macro before including the OpenCL C++ standard library headers or using the *-D* compiler option (*Preprocessor options* section).

An implementation shall support at least 8 named barriers per work-group. The exact maximum number can be queried using `clGetDeviceInfo` with `CL_DEVICE_MAX_NAMED_BARRIER_COUNT_KHR` from the OpenCL 2.2 Extension Specification.

Restrictions:

- The `work_group_named_barrier` type cannot be used with variables declared inside a class or union field, a pointer type, an array, global variables declared at program scope or the return type of a function.
- The `work_group_named_barrier` type cannot be used with the `global`, `priv` and `constant` address space storage classes (see the the *Explicit address space storage classes* section).
- The value returned by applying the `sizeof` operator to the `work_group_named_barrier` type is implementation-defined.

work_group_named_barrier

```
work_group_named_barrier(uint sub_group_count);
```

Initialize a new named barrier object to synchronize `sub_group_count` sub-groups in the current work-group. Construction of a named-barrier object is a work-group operation and hence must be called uniformly across the work-group. `sub_group_count` must be uniform across the work-group.

Named barrier objects can be reconstructed and assigned to underlying entities by the compiler, or reused. Reused barriers will always be the same size and act in phases such that when each participating sub-group has waited the wait count is set to 0 and the entire process can start again. The internal wait count will cycle through the range from `0` to `sub_group_count` in each phase of use of the barrier.

Named barrier objects can only be constructed within kernels, not within arbitrary functions.

wait

```
void wait(mem_fence flags,
          memory_scope scope = memory_scope_work_group) const noexcept;
```

All work-items in a sub-group executing the kernel on a processor must execute this method before any are allowed to continue execution beyond the barrier. This function must be encountered by all work-items in a sub-group executing the kernel.

These rules apply to ND-ranges implemented with uniform and non-uniform work-groups.

If `wait` is called inside a conditional statement, then all work-items within the sub-group must enter the conditional if any work-item in the sub-group enters the conditional statement and executes the call to wait.

If `wait` is called inside a loop, all work-items within the sub-group must execute the wait operation for each iteration of the loop before any are allowed to continue execution beyond the call to wait. The wait function causes the entire sub-group to wait until `sub_group_count` sub-groups have waited on the named barrier, where `sub_group_count` is the initialization value passed to the call to the constructor of the named barrier. Once the wait count equals `sub_group_count`, any sub-groups waiting at the named barrier will be released and the barrier's wait count reset to 0.

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 ORed together the wait acts as a combined wait operation for all address spaces specified by the flags ordering memory accesses both within and across the specified address spaces.

`mem_fence::local` - The wait function will ensure that all local memory accesses become visible to all work-items in the sub-group. Note that the value of `scope` is ignored as the memory scope is always `memory_scope_work_group`.

`mem_fence::global` - The wait function ensure that all global memory accesses become visible to the appropriate scope as given by `scope`.

`mem_fence::image` cannot be specified as a flag for this function.

The values of `flags` and `scope` must be the same for all work-items in the sub-group.

The below example shows how to use the named barriers:

```
#include <opencl_memory>
#include <opencl_synchronization>
#include <opencl_work_item>

using namespace cl;

void aFunction(work_group_named_barrier &b) {
    while(...) {
        // Do something in first set
        b.wait(mem_fence::global);
    }
}

kernel void aKernel() {
    // Initialize a set of named barriers
    local<work_group_named_barrier> a(4);
    local<work_group_named_barrier> b(2);
    local<work_group_named_barrier> c(2);

    if(get_sub_group_id() < 4) {
        a.wait(mem_fence::local);
        if(get_sub_group_id() < 2) {
            // Pass one of the named barriers to a function by reference
            // Barrier cannot be constructed in a non-kernel function
            aFunction(b);
        } else {
            // Do something else
            c.wait(mem_fence::local);
            // Continue
        }

        // Wait a second time on the first barrier
        a.wait(mem_fence::global);
    }


    // Back to work-group uniform control flow,
    // we can synchronize the entire group if necessary
    work_group_barrier(mem_fence::global);
}
```

3.17. Common Functions

This section describes the OpenCL C++ library common functions that take scalar or vector arguments. Vector versions of common functions operate component-wise. Descriptions are always per-component.

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

Here gentype matches: halfn [4], floatn or doublen [18]

3.17.1. Header <opencl_common> Synopsis

```
namespace cl
{
#ifdef cl_khr_fp16
halfn clamp(halfn x, halfn min, halfn max);
halfn degrees(halfn t);
halfn max(halfn x, halfn y);
halfn min(halfn x, halfn y);
halfn mix(halfn x, halfn y, halfn a);
halfn radians(halfn t);
halfn step(halfn edge, halfn x);
halfn smoothstep(halfn edge0, halfn edge1, halfn x);
halfn sign(halfn t);
#endif

#ifdef cl_khr_fp64
doublen clamp(doublen x, doublen min, doublen max);
doublen degrees(doublen t);
doublen max(doublen x, doublen y);
doublen min(doublen x, doublen y);
doublen mix(doublen x, doublen y, doublen a);
doublen radians(doublen t);
doublen step(doublen edge, doublen x);
doublen smoothstep(doublen edge0, doublen edge1, doublen x);
doublen sign(doublen t);
#endif

floatn clamp(floatn x, floatn min, floatn max);
floatn degrees(floatn t);
floatn max(floatn x, floatn y);
floatn min(floatn x, floatn y);
floatn mix(floatn x, floatn y, floatn a);
floatn radians(floatn t);
floatn step(floatn edge, floatn x);
floatn smoothstep(floatn edge0, floatn edge1, floatn x);
floatn sign(floatn t);

}
```

3.17.2. Common operations

clamp

```
gentype clamp(gentype x, gentype minval, gentype maxval);
```

Returns `fmin(fmax(x, minval), maxval)`.

Results are undefined if `minval > maxval`.

degrees

```
gentype degrees(gentype radians);
```

Converts radians to degrees, i.e. `(180 / π) * radians`.

max

```
gentype max(gentype x, gentype y);
```

Returns `y` if `x < y`, otherwise it returns `x`. If `x` or `y` are `infinite` or `NaN`, the return values are undefined.

min

```
gentype min(gentype x, gentype y);
```

Returns `y` if `y < x`, otherwise it returns `x`. If `x` or `y` are `infinite` or `NaN`, the return values are undefined.

mix

```
gentype mix(gentype x, gentype y, gentype a);
```

Returns the linear blend of `x` and `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.

radians

```
gentype radians(gentype degrees);
```

Converts degrees to radians, i.e. `(π / 180) * degrees`.

step

```
gentype step(gentype edge, gentype x);
```

Returns `0.0` if `x < edge`, otherwise it returns `1.0`.

smoothstep

```
gentype smoothstep(gentype edge0, gentype edge1, gentype 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`.

sign

```
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`.

3.18. Geometric Functions

This section describes the OpenCL C++ library geometric functions that take scalar or vector arguments. Vector versions of geometric functions operate component-wise. Descriptions are always per-component. The geometric functions are implemented using the round to nearest even rounding mode.

`floatn` is `float`, `float2`, `float3` or `float4`.

`halfn` [4] is `half`, `half2`, `half3` or `half4`.

`doublen` [18] is `double`, `double2`, `double3` or `double4`.

3.18.1. Header <opencl_geometric> Synopsis

```
namespace cl
{
#ifdef cl_khr_fp16
half3 cross(half3 p0, half3 p1);
half4 cross(half4 p0, half4 p1);
half dot(half p0, half p1);
half dot(half2 p0, half2 p1);
half dot(half3 p0, half3 p1);
half dot(half4 p0, half4 p1);
half distance(half p0, half p1);
half distance(half2 p0, half2 p1);
half distance(half3 p0, half3 p1;
half distance(half4 p0, half4 p1);
half length(half t);
half length(half2 t);
half length(half3 t);
half length(half4 t);
half normalize(half t);
half2 normalize(half2 t);
half3 normalize(half3 t);
half4 normalize(half4 t);
#endif

#ifdef cl_khr_fp64
double3 cross(double3 p0, double3 p1);
double4 cross(double4 p0, double4 p1);
double dot(double p0, double p1);
double dot(double2 p0, double2 p1);
double dot(double3 p0, double3 p1);
double dot(double4 p0, double4 p1);
double distance(double p0, double p1);
double distance(double2 p0, double2 p1);
double distance(double3 p0, double3 p1);
double distance(double4 p0, double4 p1);
double length(double t);
double length(double2 t);
double length(double3 t);
double length(double4 t);
double normalize(double t);
double2 normalize(double2 t);
double3 normalize(double3 t);
double4 normalize(double4 t);
#endif

float3 cross(float3 p0, float3 p1);
float4 cross(float4 p0, float4 p1);
float dot(float p0, float p1);
float dot(float2 p0, float2 p1);
float dot(float3 p0, float3 p1);
float dot(float4 p0, float4 p1);
float distance(float p0, float p1);
float distance(float2 p0, float2 p1);
float distance(float3 p0, float3 p1);
float distance(float4 p0, float4 p1);
float length(float t);
float length(float2 t);
float length(float3 t);
float length(float4 t);
float normalize(float t);
float2 normalize(float2 t);
float3 normalize(float3 t);
float4 normalize(float4 t);

float fast_distance(float p0, float p1);
float fast_distance(float2 p0, float2 p1);
float fast_distance(float3 p0, float3 p1);
float fast_distance(float4 p0, float4 p1);
float fast_length(float t);
float fast_length(float2 t);
float fast_length(float3 t);
float fast_length(float4 t);
float fast_normalize(float t);
float2 fast_normalize(float2 t);
float3 fast_normalize(float3 t);
float4 fast_normalize(float4 t);

}
```

3.18.2. Geometric operations

cross

```
float4 cross(float4 p0, float4 p1);
float3 cross(float3 p0, float3 p1);
double4 cross(double4 p0, double4 p1);
double3 cross(double3 p0, double3 p1);
half4 cross(half4 p0, half4 p1);
half3 cross(half3 p0, half3 p1);
```

Returns the cross product of `p0.xyz` and `p1.xyz`. The `w` component of `float4` result returned will be `0.0`.

dot

```
float dot(floatn p0, floatn p1);
double dot(doublen p0, doublen p1);
half dot(halfn p0, halfn p1);
```

Compute dot product.

distance

```
float distance(floatn p0, floatn p1);
double distance(doublen p0, doublen p1);
half distance(halfn p0, halfn p1);
```

Returns the distance between `p0` and `p1`.

This is calculated as `length(p0 - p1)`.

length

```
float length(floatn p);
double length(doublen p);
half length(halfn p);
```

Return the length of vector `p`, i.e., $\sqrt{p.x^2 + p.y^2 + \ldots}$

normalize

```
floatn normalize(floatn p);
doublen normalize(doublen p);
halfn normalize(halfn p);
```

Returns a vector in the same direction as `p` but with a length of `1`.

fast_distance

```
float fast_distance(floatn p0, floatn p1);
```

Returns `fast_length(p0 - p1)`.

fast_length

```
float fast_length(floatn p);
```

Returns the length of vector `p` computed as:

```
half_sqrt(p.x^2 + p.y^2 + ...)
```

fast_normalize

```
floatn fast_normalize(floatn 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 + ...)
```

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:

- If the sum of squares is greater than FLT_MAX then the value of the floating-point values in the result vector are undefined.
- If the sum of squares is less than FLT_MIN then the implementation may return back p.
- 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.

3.19. Math Functions

,The list of the OpenCL C++ library math functions is described in *Trigonometric functions*, *Logarithmic functions*, *Exponential functions*, *Floating-point functions*, *Comparison functions*, and *Other functions* sections.

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

- A list of built-in functions that have scalar or vector argument versions.
- 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.

Here gentype matches: halfn [4], floatn or doublen [18]

3.19.1. Header <opencl_math> Synopsis

```
namespace cl
{
//trigonometric functions
gentype acos(gentype x);
gentype acosh(gentype x);
gentype acospi(gentype x);
gentype asin(gentype x);
gentype asinh(gentype x);
gentype asinpi(gentype x);
gentype atan(gentype x);
gentype atanh(gentype x);
gentype atanpi(gentype x);
gentype atan2(gentype y, gentype x);
gentype atan2pi(gentype y, gentype x);
gentype cos(gentype x);
gentype cosh(gentype x);
gentype cospi(gentype x);
gentype sin(gentype x);
gentype sincos(gentype x, gentype * cosval);
gentype sinh(gentype x);
gentype sinpi(gentype x);
gentype tan(gentype x);
gentype tanh(gentype x);
gentype tanpi(gentype x);

//power functions
gentype cbrt(gentype x);
gentype pow(gentype x, gentype y);
gentype pown(gentype x, intn y);
gentype powr(gentype x, gentype y);
gentype rootn(gentype x, intn y);
gentype rsqrt(gentype x);
gentype sqrt(gentype x);

//logarithmic functions
intn ilogb(gentype x);
gentype lgamma(gentype x);
gentype lgamma_r(gentype x, intn* signp);
gentype log(gentype x);
gentype logb(gentype x);
gentype log2(gentype x);
gentype log10(gentype x);
gentype log1p(gentype x);

//exponential functions
gentype exp(gentype x);
gentype expm1(gentype x);
gentype exp2(gentype x);
gentype exp10(gentype x);
gentype ldexp(gentype x, intn k);

//floating-point functions
gentype ceil(gentype x);
gentype copysign(gentype x, gentype y);
gentype floor(gentype x);
gentype fma(gentype a, gentype b, gentype c);
gentype fmod(gentype x, gentype y);
gentype fract(gentype x, gentype* iptr);
gentype frexp(gentype x, intn* exp);
gentype modf(gentype x, gentype* iptr);
#ifdef cl_khr_fp16
halfn nan(ushortn nancode);
#endif
floatn nan(uintn nancode);
#ifdef cl_khr_fp64
doublen nan(ulong nancode);
#endif
gentype nextafter(gentype x, gentype y);
gentype remainder(gentype x, gentype y);
gentype remquo(gentype x, gentype y, intn* quo);
gentype rint(gentype x);
gentype round(gentype x);
gentype trunc(gentype x);

//comparison functions
gentype fdim(gentype x, gentype y);
gentype fmax(gentype x, gentype y);
gentype fmin(gentype x, gentype y);
gentype maxmag(gentype x, gentype y);
gentype minmag(gentype x, gentype y);

//other functions
gentype erf(gentype x);
gentype erfc(gentype x);
gentype fabs(gentype x);
gentype hypot(gentype x, gentype y);
gentype mad(gentype a, gentype b, gentype c);
gentype tgamma(gentype x);

//native functions
namespace native
{
floatn cos(floatn x);
floatn exp(floatn x);
floatn exp2(floatn x);
floatn exp10(floatn x);
floatn log(floatn x);
floatn log2(floatn x);
floatn log10(floatn x);
floatn recip(floatn x);
floatn rsqrt(floatn x);
floatn sin(floatn x);
floatn sqrt(floatn x);
floatn tan(floatn x);
floatn divide(floatn x, floatn y);
floatn powr(floatn x, floatn y);
}

//half_math functions
namespace half_math
{
floatn cos(floatn x);
floatn exp(floatn x);
floatn exp2(floatn x);
floatn exp10(floatn x);
floatn log(floatn x);
floatn log2(floatn x);
floatn log10(floatn x);
floatn recip(floatn x);
floatn rsqrt(floatn x);
floatn sin(floatn x);
floatn sqrt(floatn x);
floatn tan(floatn x);
floatn divide(floatn x, floatn y);
floatn powr(floatn x, floatn y);

}

}
```

3.19.2. Trigonometric functions

acos

```
gentype acos(gentype x);
```

Arc cosine function. Returns an angle in radians.

acosh

```
gentype acosh(gentype x);
```

Inverse hyperbolic cosine. Returns an angle in radians.

acospi

```
gentype acospi(gentype x);
```

Compute acos(`x`) / π.

asin

```
gentype asin(gentype x);
```

Arc sine function. Returns an angle in radians.

asinh

```
gentype asinh(gentype x);
```

Inverse hyperbolic sine. Returns an angle in radians.

asinpi

```
gentype asinpi(gentype x);
```

Compute asin(`x`) / π.

atan

```
gentype atan(gentype y_over_x);
```

Arc tangent function. Returns an angle in radians.

atan2

```
gentype atan2(gentype y, gentype x);
```

Arc tangent of `y / x`. Returns an angle in radians.

atanh

```
gentype atanh(gentype x);
```

Hyperbolic arc tangent. Returns an angle in radians.

atanpi

```
gentype atanpi(gentype x);
```

Compute atan(`x`) / π.

atan2pi

```
gentype atan2pi(gentype y, gentype x);
```

Compute atan2(`y`, `x`) / π.

cos

```
gentype cos(gentype x);
```

Compute cosine, where `x` is an angle in radians.

cosh

```
gentype cosh(gentype x);
```

Compute hyperbolic consine, where `x` is an angle in radians.

cospi

```
gentype cospi(gentype x);
```

Compute cos(π x).

sin

```
gentype sin(gentype x);
```

Compute sine, where `x` is an angle in radians.

sincos

```
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

sinh

```
gentype sinh(gentype x);
```

Compute hyperbolic sine, where `x` is an angle in radians

sinpi

```
gentype sinpi(gentype x);
```

Compute sin(π `x`).

tan

```
gentype tan(gentype x);
```

Compute tangent, where `x` is an angle in radians.

tanh

```
gentype tanh(gentype x);
```

Compute hyperbolic tangent, where `x` is an angle in radians.

tanpi

```
gentype tanpi(gentype x);
```

Compute tan(π `x`).

3.19.3. Power function

cbrt

```
gentype cbrt(gentype x);
```

Compute cube-root.

pow

```
gentype pow(gentype x, gentype y);
```

Compute `x` to the power `y`.

pown

```
floatn pown(gentype x, intn y);
```

Compute x to the power y, where y is an integer.

powr

```
gentype powr(gentype x, gentype y);
```

Compute x to the power y, where x is >= 0.

rootn

```
gentype powr(gentype x, gentype y);
```

Compute x to the power 1/y.

rsqrt

```
gentype rsqrt(gentype x);
```

Compute inverse square root.

sqrt

```
gentype sqrt(gentype x);
```

Compute square root.

3.19.4. Logarithmic functions

ilogb

```
intn ilogb(gentype x);
```

Return the exponent as an integer value.

lgamma

```
gentype lgamma(gentype x);

gentype lgamma_r(gentype x, intn *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.

log

```
gentype log(gentype x);
```

Compute natural logarithm.

log2

```
gentype log2(gentype x);
```

Compute a base 2 logarithm.

log10

```
gentype log10(gentype x);
```

Compute a base 10 logarithm.

log1p

```
gentype log1p(gentype x);
```

Compute $\log_e(1.0 + x)$.

logb

```
gentype logb(gentype x);
```

Compute the exponent of `x`, which is the integral part of $\log_r$|`x`|.

3.19.5. Exponential functions

exp

```
gentype exp(gentype x);
```

Compute the base e exponential of `x`.

exp2

```
gentype exp2(gentype x);
```

Exponential base 2 function.

exp10

```
gentype exp10(gentype x);
```

Exponential base 10 function.

expm1

```
gentype expm1(gentype x);
```

Compute e^x- 1.0.

ldexp

```
gentype ldexp(gentype x, intn k);
```

Multiply `x` by 2 to the power `k`.

3.19.6. Floating-point functions

ceil

```
gentype ceil(gentype x);
```

Round to integral value using the round to positive infinity rounding mode.

copysign

```
gentype copysign(gentype x, gentype y);
```

Returns x with its sign changed to match the sign of y.

floor

```
gentype floor(gentype x);
```

Round to integral value using the round to negative infinity rounding mode.

fma

```
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.

fmod

```
gentype fmod(gentype x, gentype y);
```

Modulus. Returns `x - y * trunc (x/y)`.

fract

`fract` [20]:

```
gentype fract(gentype x, gentype *iptr);
```

Returns `fmin(x - floor(x), 0x1.fffffep-1f)`.

`floor(x)` is returned in `iptr`.

frexp

```
gentype frexp(gentype x, intn *exp);
```

Extract mantissa and exponent from `x`. For each component the mantissa returned is a half with magnitude in the interval [1/2, 1) or 0. Each component of `x` equals mantissa returned $* \ 2^{exp}$.

modf

```
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`.

nan

```
floatn nan(uintn nancode);
doublen nan(ulongn nancode);
halfn nan(ushortn nancode);
```

Returns a quiet `NaN`. The nancode may be placed in the significand of the resulting `NaN`.

nextafter

```
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`.

remainder

```
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`.

remquo

```
gentype remquo(gentype x, gentype y, intn *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 at least the seven lower 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`.

rint

```
gentype rint(gentype x);
```

Round to integral value (using round to nearest even rounding mode) in floating-point format. Refer to the *Rounding Modes* section for description of rounding modes.

round

```
gentype round(gentype x);
```

Return the integral value nearest to `x` rounding halfway cases away from zero, regardless of the current rounding direction.

trunc

```
gentype trunc(gentype x);
```

Round to integral value using the round to zero rounding mode.

3.19.7. Comparison functions

fdim

```
gentype fdim(gentype x, gentype y);
```

`x - y` if `x > y`, `+0` if `x` is less than or equal to `y`.

fmax

```
gentype fmax(gentype x, gentype 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`.

fmin

`fmin` [21]:

```
gentype fmin(gentype x, gentype 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`.

fmod

```
gentype fmod(gentype x, gentype y);
```

Modulus. Returns `x - y * trunc (x/y)`.

maxmag

```
gentype maxmag(gentype x, gentype y);
```

Returns `x` if `|x| > |y|`, `y` if `|y| > |x|`, otherwise `fmax(x, y)`.

minmag

```
gentype minmag(gentype x, gentype y);
```

Returns `x` if `|x| < |y|`, `y` if `|y| < |x|`, otherwise `fmin(x, y)`.

3.19.8. Other functions

erfc

```
gentype erfc(gentype x);
```

Complementary error function.

erf

```
gentype erf(gentype x);
```

Error function encountered in integrating the normal distribution.

fabs

```
gentype fabs(gentype x);
```

Compute absolute value of a floating-point number.

hypot

```
gentype hypot(gentype x, gentype y);
```

Compute the value of the square root of $x^2 + y^2$ without undue overflow or underflow.

mad

```
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. It is implemented either as a correctly rounded fma, or as a multiply followed by an add, both of which are correctly rounded. On some hardware the mad instruction may provide better performance than expanded computation of `a * b + c`. [22]

tgamma

```
gentype tgamma(gentype x);
```

Compute the gamma function.

3.19.9. Native functions

This section describes the following functions:

- A subset of functions from previous sections that are defined in the `cl::native_math` namespace. 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_math namespace) described in the *Trigonometric functions*, *Logarithmic functions*, *Exponential functions*, *Floating-point functions*, *Comparison functions* and *Other functions* sections. The accuracy (and in some cases the input range(s)) of these functions is implementation-defined.
- Native functions for following basic operations: divide and reciprocal.
- Support for denormal values is implementation-defined for native functions.

native_math::cos

```
floatn native_math::cos(floatn x);
```

Compute cosine over an implementation-defined range, where `x` is an angle in radians. The maximum error is implementation-defined.

native_math::divide

```
floatn native_math::divide(floatn x, floatn y);
```

Compute `x / y` over an implementation-defined range. The maximum error is implementation-defined.

native_math::exp

```
floatn native_math::exp(floatn x);
```

Compute the base e exponential of `x` over an implementation-defined range. The maximum error is implementation-defined.

native_math::exp2

```
floatn native_math::exp2(floatn x);
```

Compute the base 2 exponential of x over an implementation-defined range. The maximum error is implementation-defined.

native_math::exp10

```
floatn native_math::exp10(floatn x);
```

Compute the base 10 exponential of x over an implementation-defined range. The maximum error is implementation-defined.

native_math::log

```
floatn native_math::log(floatn x);
```

Compute natural logarithm over an implementation-defined range. The maximum error is implementation-defined.

native_math::log2

```
floatn native_math::log2(floatn x);
```

Compute a base 2 logarithm over an implementation-defined range. The maximum error is implementation-defined.

native_math::log10

```
floatn native_math::log10(floatn x);
```

Compute a base 10 logarithm over an implementation-defined range. The maximum error is implementation-defined.

native_math::powr

```
floatn native_math::powr(floatn x, floatn 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.

native_math::recip

```
floatn native_math::recip(floatn x);
```

Compute reciprocal over an implementation-defined range. The maximum error is implementation-defined.

native_math::rsqrt

```
floatn native_math::rsqrt(floatn x);
```

Compute inverse square root over an implementation-defined range. The maximum error is implementation-defined.

native_math::sin

```
floatn native_math::sin(floatn x);
```

Compute sine over an implementation-defined range, where `x` is an angle in radians. The maximum error is implementation-defined.

native_math::sqrt

```
floatn native_math::sqrt(floatn x);
```

Compute square root over an implementation-defined range. The maximum error is implementation-defined.

native_math::tan

```
floatn native_math::tan(floatn x);
```

Compute tangent over an implementation-defined range, where `x` is an angle in radians. The maximum error is implementation-defined.

3.19.10. Half functions

This section describes the following functions:

- A subset of functions from previous sections that are defined in the `cl::half_math` namespace. These functions are implemented with a minimum of 10-bits of accuracy i.e. an ULP value <= 8192 ulp.
- half functions for following basic operations: divide and reciprocal.
- Support for denormal values is optional for half_math:: functions. The `half_math::` functions may return any result allowed by the *Edge Case Behavior in Flush To Zero Mode* section, even when `-cl-denorms-are-zero` is not in force.

half_math::cos

```
floatn half_math::cos(floatn x);
```

Compute cosine. `x` is an angle in radians and it must be in the range -2^{16} … $+2^{16}$.

half_math::divide

```
floatn half_math::divide(floatn x, floatn y);
```

Compute `x / y`.

half_math::exp

```
floatn half_math::exp(floatn x);
```

Compute the base e exponential of `x`.

half_math::exp2

```
floatn half_math::exp2(floatn x);
```

Compute the base 2 exponential of `x`.

half_math::exp10

```
floatn half_math::exp10(floatn x);
```

Compute the base 10 exponential of `x`.

half_math::log

```
floatn half_math::log(floatn x);
```

Compute natural logarithm.

half_math::log2

```
floatn half_math::log2(floatn x);
```

Compute a base 2 logarithm.

half_math::log10

```
floatn half_math::log10(floatn x);
```

Compute a base 10 logarithm.

half_math::powr

```
floatn half_math::powr(floatn x, floatn y);
```

Compute `x` to the power `y`, where `x` is `>= 0`.

half_math::recip

```
floatn half_math::recip(floatn x);
```

Compute reciprocal.

half_math::rsqrt

```
floatn half_math::rsqrt(floatn x);
```

Compute inverse square root.

half_math::sin

```
floatn half_math::sin(floatn x);
```

Compute sine. `x` is an angle in radians and it must be in the range -2^{16} … $+2^{16}$.

half_math::sqrt

```
floatn half_math::sqrt(floatn x);
```

Compute square root.

half_math::tan

```
floatn half_math::tan(floatn x);
```

Compute tangent. `x` is an angle in radians and it must be in the range -2^{16} … $+2^{16}$.

3.19.11. Floating-point 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:

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

The DEFAULT value is ON.

3.20. Integer Functions

This section describes the OpenCL C++ library integer functions that take scalar or vector arguments. Vector versions of integer functions operate component-wise. Descriptions are always per-component.

Here gentype matches: char*n*, uchar*n*, short*n*, ushort*n*, int*n*, uint*n*, long*n* and ulong*n*.

3.20.1. Header <opencl_integer> Synopsis

```
namespace cl
{
//bitwise functions
gentype clz(gentype x);
gentype ctz(gentype x);
gentype popcount(gentype x);
gentype rotate(gentype v, gentype i);

shortn upsample(charn hi, ucharn lo);
ushortn upsample(ucharn hi, ucharn lo);
intn upsample(shortn hi, ushortn lo);
uintn upsample(ushortn hi, ushortn lo);
longn upsample(intn hi, uintn lo);
ulongn upsample(uintn hi, uintn lo);

//numeric functions
ugentype abs(gentype x);
ugentype abs_diff(gentype x, gentype y);
gentype  add_sat(gentype x, gentype y);
gentype  hadd(gentype x, gentype y);
gentype  rhadd(gentype x, gentype y);
gentype  clamp(gentype x, gentype minval, gentype maxval);
gentype  clamp(gentype x, sgentype minval, sgentype maxval);
gentype  mad_hi(gentype a, gentype b, gentype c);
gentype  mad_sat(gentype a, gentype b, gentype c);
gentype  max(gentype x, gentype y);
gentype  max(gentype x, sgentype y);
gentype  min(gentype x, gentype y);
gentype  min(gentype x, sgentype y);
gentype  mul_hi(gentype x, gentype y);
gentype  sub_sat(gentype x, gentype y);

//24-bits functions
intn mad24(intn x, intn y, intn z);
uintn mad24(uintn x, uintn y, uintn z);
intn mul24(intn x, intn y);
uintn mul24(uintn x, uintn y);

}
```

3.20.2. Bitwise operations

clz

```
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.

ctz

```
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.

rotate

```
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 usual shift modulo rules described in the *Expressions* section). Bits shifted off the left side of the element are shifted back in from the right.

upsample

```
shortn upsample(charn hi, ucharn lo);
ushortn upsample(ucharn hi, ucharn lo);

intn upsample(shortn hi, ushortn lo);
uintn upsample(ushortn hi, ushortn lo);

longn upsample(intn hi, uintn lo);
ulongn upsample(uintn hi, uintn lo);
```

`result[i] = ((short)hi[i] << 8) | lo[i]`

`result[i] = ((ushort)hi[i] << 8) | lo[i]`

`result[i] = ((int)hi[i] << 16) | lo[i]`

`result[i] = ((uint)hi[i] << 16) | lo[i]`

`result[i] = ((long)hi[i] << 32) | lo[i]`

`result[i] = ((ulong)hi[i] << 32) | lo[i]`

popcount

```
gentype popcount(gentype x);
```

Returns the number of non-zero bits in `x`.

3.20.3. Numeric functions

abs

```
ugentype abs(gentype x);
```

Returns `|x|`.

abs_diff

```
ugentype abs_diff(gentype x, gentype y);
```

Returns `|x - y|` without modulo overflow.

add_sat

```
gentype add_sat(gentype x, gentype y);
```

Returns `x + y` and saturates the result.

hadd

```
gentype hadd(gentype x, gentype y);
```

Returns `(x + y) >> 1`. The intermediate sum does not modulo overflow.

rhadd

`rhadd` [23]:

```
gentype rhadd(gentype x, gentype y);
```

Returns `(x + y + 1) >> 1`. The intermediate sum does not modulo overflow.

clamp

```
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`.

mad_hi

```
gentype mad_hi(gentype a, gentype b, gentype c);
```

Returns `mul_hi(a, b) + c`.

mad_sat

```
gentype mad_sat(gentype a, gentype b, gentype c);
```

Returns `a * b + c` and saturates the result.

max

```
gentype max(gentype x, gentype y);
gentype max(gentype x, sgentype y);
```

Returns `y` if `x < y`, otherwise it returns `x`.

min

```
gentype min(gentype x, gentype y);
gentype min(gentype x, sgentype y);
```

Returns `y` if `y < x`, otherwise it returns `x`.

mul_hi

```
gentype mul_hi(gentype x, gentype y);
```

Computes `x * y` and returns the high half of the product of `x` and `y`.

sub_sat

```
gentype sub_sat(gentype x, gentype y);
```

Returns `x - y` and saturates the result.

3.20.4. 24-bits operations

In this section fast integer functions are described that can be used for optimizing performance of kernels.

mad24

```
intn mad24(intn x, intn y, intn z);
uintn mad24(uintn x, uintn y, uintn z);
```

Multiply 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.

```
intn mul24(intn x, intn y);
uintn mul24(uintn x, uintn 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.

3.21. Relational Functions

The relational and equality operators (<, <=, >, >=, !=, ==) can be used with scalar and vector built-in types and produce a scalar or vector boolean result respectively as described in the *Expressions* section.

Here gentype matches: charn, ucharn, shortn, ushortn, intn, uintn, longn, ulongn, halfn [4], floatn and doublen [18].

The relational functions isequal, isgreater, isgreaterequal, isless, islessequal, and islessgreater always return false if either argument is not a number (NaN). isnotequal returns true if one or both arguments are not a number (NaN).

3.21.1. Header <opencl_relational> Synopsis

```
namespace cl
{
#ifdef cl_khr_fp16
booln isequal(halfn x, halfn y);
booln isnotequal(halfn x, halfn y);
booln isgreater(halfn x, halfn y);
booln isgreaterequal(halfn x, halfn y);
booln isless(halfn x, halfn y);
booln islessequal(halfn x, halfn y);
booln islessgreater(halfn x, halfn y);
booln isordered(halfn x, halfn y);
booln isunordered(halfn x, halfn y);
booln isfinite(halfn t);
booln isinf(halfn t);
booln isnan(halfn t);
booln isnormal(halfn t);
booln signbit(halfn t);

#endif

#ifdef cl_khr_fp64
booln isequal(doublen x, doublen y);
booln isnotequal(doublen x, doublen y);
booln isgreater(doublen x, doublen y);
booln isgreaterequal(doublen x, doublen y);
booln isless(doublen x, doublen y);
booln islessequal(doublen x, doublen y);
booln islessgreater(doublen x, doublen y);
booln isordered(doublen x, doublen y);
booln isunordered(doublen x, doublen y);
booln isfinite(doublen t);
booln isinf(doublen t);
booln isnan(doublen t);
booln isnormal(doublen t);
booln signbit(doublen t);

#endif //cl_khr_fp64

booln isequal(floatn x, floatn y);
booln isnotequal(floatn x, floatn y);
booln isgreater(floatn x, floatn y);
booln isgreaterequal(floatn x, floatn y);
booln isless(floatn x, floatn y);
booln islessequal(floatn x, floatn y);
booln islessgreater(floatn x, floatn y);
booln isordered(floatn x, floatn y);
booln isunordered(floatn x, floatn y);
booln isfinite(floatn t);
booln isinf(floatn t);
booln isnan(floatn t);
booln isnormal(floatn t);
booln signbit(floatn t);

bool any(booln t);
bool all(booln t);

gentype bitselect(gentype a, gentype b, gentype c);
gentype select(gentype a, gentype b, booln c);

}
```

3.21.2. Comparison operations

isequal

```
booln isequal(gentype x, gentype y);
```

Returns the component-wise compare of `x == y`.

isnotequal

```
booln isnotequal(gentype x, gentype y);
```

Returns the component-wise compare of `x != y`.

isgreater

```
booln isgreater(gentype x, gentype y);
```

Returns the component-wise compare of `x > y`.

isgreaterequal

```
booln isgreaterequal(gentype x, gentype y);
```

Returns the component-wise compare of `x >= y`.

isless

```
booln isless(gentype x, gentype y);
```

Returns the component-wise compare of `x < y`.

islessequal

```
booln islessequal(gentype x, gentype y);
```

Returns the component-wise compare of `x <= y`.

islessgreater

```
booln islessgreater(gentype x, gentype y);
```

Returns the component-wise compare of `(x < y) || (x > y)`.

3.21.3. Test operations

isfinite

```
booln isfinite(gentype t);
```

Test for finite value.

isinf

```
booln isinf(gentype t);
```

Test for infinity value (positive or negative) .

isnan

```
booln isnan(gentype t);
```

Test for a NaN.

isnormal

```
booln isnormal(gentype t);
```

Test for a normal value.

isordered

```
booln isordered(gentype x, gentype y);
```

Test if arguments are ordered. `isordered()` takes arguments `x` and `y`, and returns the result of `isequal(x, x) && isequal(y, y)`.

isunordered

```
booln isunordered(gentype x, gentype y);
```

Test if arguments are unordered. `isunordered()` takes arguments `x` and `y`, returning `true` if `x` or `y` is NaN, and `false` otherwise.

signbit

```
booln signbit(gentype t);
```

Test for sign bit. Returns a `true` if the sign bit in the float is set else returns `false`.

any

```
bool any(booln t);
```

Returns `true` if any component of `t` is `true`; otherwise returns `false`.

all

```
bool all(booln t);
```

Returns `true` if all components of `t` are `true`; otherwise returns `false`.

3.21.4. Select operations

bitselect

```
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`.

select

```
gentype select(gentype a, gentype b, booln c);
```

For each component of a vector type,

`result[i] = c[i] ? b[i] : a[i]`.

For a scalar type, `result = c ? b : a`.

`booln` must have the same number of elements as `gentype`.

3.22. Vector Data Load and Store Functions

Functions described in this section allow user to read and write vector types from a pointer to memory. The results of these functions are undefined if the address being read from or written to is not correctly aligned as described in following subsections.

Here `gentype` matches: `charn`, `ucharn`, `shortn`, `ushortn`, `intn`, `uintn`, `longn`, `ulongn`, `halfn` [4], `floatn` and doublen [18].

3.22.1. Header <opencl_vector_load_store> Synopsis

```
namespace cl
{
//basic load & store
template <size_t N, class T>
make_vector_t<T, N> vload(size_t offset, const T* p);

template <size_t N, class T>
make_vector_t<T, N> vload(size_t offset, const constant_ptr<T> p);

template <class T>
void vstore(T data, size_t offset, vector_element_t<T>* p);

//half load & store
template <size_t N>
make_vector_t<float, N> vload_half(size_t offset, const half* p);

template <size_t N>
make_vector_t<float, N> vload_half(size_t offset, const constant_ptr<half> p);

template <rounding_mode rmode = rounding_mode::rte, class T>
void vstore_half(T data, size_t offset, half* p);

//half array load & store
template <size_t N>
make_vector_t<float, N> vloada_half(size_t offset, const half* p);

template <size_t N>
make_vector_t<float, N> vloada_half(size_t offset, const constant_ptr<half> p);

template <rounding_mode rmode = rounding_mode::rte, class T>
void vstorea_half(T data, size_t offset, half* p);

}
```

3.22.2. Basic load & store

vload

```
template <size_t N, class T>
make_vector_t<T, N> vload(size_t offset, const T* p);

template <size_t N, class T>
make_vector_t<T, N> vload(size_t offset, const constant_ptr<T> p);
```

Return `sizeof(make_vector_t<T, N>)` bytes of data read from address `(p + (offset * n))`.

Requirements:

- The address computed as `(p+(offset*n))` must be 8-bit aligned if `T` is `char`, or `uchar`; 16-bit aligned if `T` is `short`, `ushort`, or `half` [4]; 32-bit aligned if `T` is `int`, `uint`, or `float`; and 64-bit aligned if `T` is `long`, `ulong`, or `double` [18].
- `vload` function is only defined for n = 2, 3, 4, 8, 16.
- `half` version is only defined if the **cl_khr_fp16** extension is supported.
- `double` version is only defined if double precision is supported.

vstore

```
template <class T>
void vstore(T data, size_t offset, vector_element_t<T>* p);
```

Write `sizeof(T)` bytes given by data to address `(p+(offset*n))`.

Requirements:

- The address computed as `(p+(offset*n))` must be 8-bit aligned if `T` is `char`, or `uchar`; 16-bit aligned if `T` is `short`, `ushort`, or `half` [4]; 32-bit aligned if `T` is `int`, `uint`, or `float`; and 64-bit aligned if `T` is `long`, `ulong`, or `double` [18].
- `vstore` function is only defined for n = 2, 3, 4, 8, 16.
- `half` version is only defined if the **cl_khr_fp16** extension is supported.
- `double` version is only defined if double precision is supported.

3.22.3. half vload & vstore

vload_half

```
template <size_t N>
make_vector_t<float, N> vload_half(size_t offset, const half* p);

template <size_t N>
make_vector_t<float, N> vload_half(size_t offset, const constant_ptr<half> p);
```

Read `sizeof(halfn)` bytes of data from address `(p+(offset*n))`. The data read is interpreted as a `halfn` value. The `halfn` value read is converted to a `float` value and the `floatn` value is returned.

Requirements:

- The read address computed as `(p+(offset*n))` must be 16-bit aligned.
- `vload_half` function is only defined for n = 1, 2, 3, 4, 8, 16.

vstore_half

```
template <rounding_mode rmode = rounding_mode::rte, class T>
void vstore_half(T data, size_t offset, half* p);
```

The `T` value given by data is first converted to a `halfn` value using the appropriate rounding mode. The `half` value is then written to address computed as `(p+offset)`.

Requirements:

- The address computed as `(p+offset)` must be 16-bit aligned.
- `T` can be `floatn` or `doublen` [18].
- `double` version is only defined if double precision is supported.
- `vstore_half` function is only defined for n = 1, 2, 3, 4, 8, 16.

3.22.4. half array vload & vstore

vloada_half

```
template <size_t N>
make_vector_t<float, N> vloada_half(size_t offset, const half* p);

template <size_t N>
make_vector_t<float, N> vloada_half(size_t offset, const constant_ptr<half> p);
```

For `N` = 2, 4, 8 and 16 read `sizeof(halfn)` bytes of data from address `(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.

Requirements:

- The address computed as `(p+(offset*n))` must be aligned to `sizeof(halfn)` bytes.
- For n = 3, `vloada_half` reads a `half3` from address `(p+(offset*4))` and returns a `float3`. The address computed as `(p+(offset*4))` must be aligned to `sizeof(half)*4` bytes.
- `vloada_half` function is only defined for `N` = 2, 3, 4, 8, 16.

vstorea_half

```
template <rounding_mode rmode = rounding_mode::rte, class T>
void vstorea_half(T data, size_t offset, half* p);
```

The `T` 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))`. For n = 3, the `half3` value is written to the address computed as `(p+(offset*4))`. The address computed as `(p+(offset*4))` must be aligned to `sizeof(half)*4` bytes.

Requirements:

- The address computed as `(p+(offset* n))` must be aligned to `sizeof(halfn)` bytes.
- For n = 3, the address computed as `(p+(offset*4))` must be aligned to `sizeof(half)*4` bytes.
- `T` can be `floatn` or `doublen` [18].
- `double` version is only defined if double precision is supported.
- `vstorea_half` function is only defined for n = 2, 3, 4, 8, 16.

3.23. printf

The OpenCL C++ programming language implements the `printf` [24] function. This function is defined in header <*opencl_printf*>.

3.23.1. Header <opencl_printf> Synopsis

```
namespace cl
{
int printf(const char *format, ...);

}
```

3.23.2. printf function

printf

```
int printf(const char *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.

Limitations:

- Format must be known at compile time; otherwise it will be a compile time error.

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 host 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. [25]
- 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 (as per the *Implicit Type Conversions* section). 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.

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.) [26]

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:

vn

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 `ushort` 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` argument.

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 an `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 `uint`, `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. [27]

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` 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 [28] 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.

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 uchar and the resulting character is written.

s

The argument shall be a literal string. [29] 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.

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.

```
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.

```
#include <opencl_printf>

kernel void my_kernel( ... )
{
    cl::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:

```
#include <opencl_printf>

kernel void my_kernel(char *s, ... )
{
    cl::printf("%s\n", s);
}
```

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:

```
#include <opencl_printf>
using namespace cl;

kernel void my_kernel( ... )
{
    uint2 ui = uint2(0x12345678, 0x87654321);
    printf("unsigned short value = (%#v2hx)\n", ui);
    printf("unsigned char value = (%#v2hhx)\n", ui);
}
```

3.23.3. Differences between OpenCL C++ and C++14 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 *vn* 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 description of `CL_DEVICE_DOUBLE_FP_CONFIG` in *table 4.3*. 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. C++14 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.

3.24. Atomic Operations Library

The OpenCL C++ programming language implements a subset of the C++14 atomics (refer to *chapter 29* of the C++14 specification) and synchronization operations. These operations play a special role in making assignments in one work-item visible to another. Please note that this chapter only presents synopsis of the atomics library and differences from C++14 specification.

3.24.1. Header <opencl_atomic> Synopsis

```
namespace cl
{
enum memory_order;
enum memory_scope;

template<class T> struct atomic;
// specialization for scalar types T that satisfy cl::is_integral<T>
template<> struct atomic<integral>;
template<class T> struct atomic<T*>;

using atomic_int = atomic<int>;
using atomic_uint = atomic<unsigned int>;
#if defined(cl_khr_int64_base_atomics) && defined(cl_khr_int64_extended_atomics)
using atomic_long = atomic<long>;
using atomic_ulong = atomic<unsigned long>;
#endif
using atomic_float = atomic<float>;
#if defined(cl_khr_fp64) &&
    defined(cl_khr_int64_base_atomics) &&
    defined(cl_khr_int64_extended_atomics)
using atomic_double = atomic<double>;
#endif
#if (defined(cl_khr_int64_base_atomics) &&
     defined(cl_khr_int64_extended_atomics) &&
     __INTPTR_WIDTH__ == 64) ||
    __INTPTR_WIDTH__ == 32
using atomic_intptr_t = atomic<intptr_t>;
using atomic_uintptr_t = atomic<uintptr_t>;
#endif
#if (defined(cl_khr_int64_base_atomics) &&
     defined(cl_khr_int64_extended_atomics) &&
     __SIZE_WIDTH__ == 64) ||
    __SIZE_WIDTH__ == 32
using atomic_size_t = atomic<size_t>;
```

```
#endif
#if (defined(cl_khr_int64_base_atomics) &&
     defined(cl_khr_int64_extended_atomics) &&
     __PTRDIFF_WIDTH__ == 64) ||
    __PTRDIFF_WIDTH__ == 32
using atomic_ptrdiff_t = atomic<ptrdiff_t>;
#endif

// Please note that all operations taking memory_order as a parameter have,
// in addition to cpp14 specification, additional parameter for memory_scope
template <class T>
bool atomic_is_lock_free(const volatile atomic<T>*) noexcept;
template <class T>
bool atomic_is_lock_free(const atomic<T>*) noexcept;
template <class T>
void atomic_init(volatile atomic<T>*, T) noexcept;
template <class T>
void atomic_init(atomic<T>*, T) noexcept;
template <class T>
void atomic_store(volatile atomic<T>*, T) noexcept;
template <class T>
void atomic_store(atomic<T>*, T) noexcept;
template <class T>
void atomic_store_explicit(volatile atomic<T>*, T, memory_order,
                           memory_scope) noexcept;
template <class T>
void atomic_store_explicit(atomic<T>*, T, memory_order,
                           memory_scope) noexcept;
template <class T>
T atomic_load(const volatile atomic<T>*) noexcept;
template <class T>
T atomic_load(const atomic<T>*) noexcept;
template <class T>
T atomic_load_explicit(const volatile atomic<T>*, memory_order,
                       memory_scope) noexcept;
template <class T>
T atomic_load_explicit(const atomic<T>*, memory_order, memory_scope) noexcept;
template <class T>
T atomic_exchange(volatile atomic<T>*, T) noexcept;
template <class T>
T atomic_exchange(atomic<T>*, T) noexcept;
template <class T>
T atomic_exchange_explicit(volatile atomic<T>*, T, memory_order,
                           memory_scope) noexcept;
template <class T>
T atomic_exchange_explicit(atomic<T>*, T, memory_order, memory_scope) noexcept;
template <class T>
bool atomic_compare_exchange_weak(volatile atomic<T>*, T*, T) noexcept;
template <class T>
bool atomic_compare_exchange_weak(atomic<T>*, T*, T) noexcept;
template <class T>
```

```
bool atomic_compare_exchange_strong(volatile atomic<T>*, T*, T) noexcept;
template <class T>
bool atomic_compare_exchange_strong(atomic<T>*, T*, T) noexcept;
template <class T>
bool atomic_compare_exchange_weak_explicit(volatile atomic<T>*, T*, T,
                                           memory_order, memory_order,
                                           memory_scope) noexcept;
template <class T>
bool atomic_compare_exchange_weak_explicit(atomic<T>*, T*, T, memory_order,
                                           memory_order, memory_scope) noexcept;
template <class T>
bool atomic_compare_exchange_strong_explicit(volatile atomic<T>*, T*, T,
                                             memory_order, memory_order,
                                             memory_scope) noexcept;
template <class T>
bool atomic_compare_exchange_strong_explicit(atomic<T>*, T*, T,
                                             memory_order, memory_order,
                                             memory_scope) noexcept;

// Please note that all operations taking memory_order as a parameter have
// additional overloads, in addition to cpp14 specification, taking both
// memory_order and memory_scope parameters.
template <class T>
T atomic_fetch_add(volatile atomic<T>*, T) noexcept;
template <class T>
T atomic_fetch_add(atomic<T>*, T) noexcept;
template <class T>
T atomic_fetch_add_explicit(volatile atomic<T>*, T, memory_order,
                            memory_scope) noexcept;
template <class T>
T atomic_fetch_add_explicit(atomic<T>*, T, memory_order, memory_scope) noexcept;
template <class T>
T atomic_fetch_sub(volatile atomic<T>*, T) noexcept;
template <class T>
T atomic_fetch_sub(atomic<T>*, T) noexcept;
template <class T>
T atomic_fetch_sub_explicit(volatile atomic<T>*, T, memory_order,
                            memory_scope) noexcept;
template <class T>
T atomic_fetch_sub_explicit(atomic<T>*, T, memory_order, memory_scope) noexcept;
template <class T>
T atomic_fetch_and(volatile atomic<T>*, T) noexcept;
template <class T>
T atomic_fetch_and(atomic<T>*, T) noexcept;
template <class T>
T atomic_fetch_and_explicit(volatile atomic<T>*, T, memory_order,
                            memory_scope) noexcept;
template <class T>
T atomic_fetch_and_explicit(atomic<T>*, T, memory_order, memory_scope) noexcept;
template <class T>
T atomic_fetch_or(volatile atomic<T>*, T) noexcept;
```

```
template <class T>
T atomic_fetch_or(atomic<T>*, T) noexcept;
template <class T>
T atomic_fetch_or_explicit(volatile atomic<T>*, T, memory_order,
                           memory_scope) noexcept;
template <class T>
T atomic_fetch_or_explicit(atomic<T>*, T, memory_order, memory_scope) noexcept;
template <class T>
T atomic_fetch_xor(volatile atomic<T>*, T) noexcept;
template <class T>
T atomic_fetch_xor(atomic<T>*, T) noexcept;
template <class T>
T atomic_fetch_xor_explicit(volatile atomic<T>*, T, memory_order,
                            memory_scope) noexcept;
template <class T>
T atomic_fetch_xor_explicit(atomic<T>*, T, memory_order, memory_scope) noexcept;
//OpenCL specific min/max atomics:
T atomic_fetch_min(volatile atomic<T>*, T) noexcept;
template <class T>
T atomic_fetch_min(atomic<T>*, T) noexcept;
template <class T>
T atomic_fetch_min_explicit(volatile atomic<T>*, T, memory_order,
                            memory_scope) noexcept;
template <class T>
T atomic_fetch_min_explicit(atomic<T>*, T, memory_order, memory_scope) noexcept;
T atomic_fetch_max(volatile atomic<T>*, T) noexcept;
template <class T>
T atomic_fetch_max(atomic<T>*, T) noexcept;
template <class T>
T atomic_fetch_max_explicit(volatile atomic<T>*, T, memory_order,
                            memory_scope) noexcept;
template <class T>
T atomic_fetch_max_explicit(atomic<T>*, T, memory_order, memory_scope) noexcept;

template <class T>
void atomic_store(atomic<T>* object, T value) noexcept;
template <class T>
void atomic_store(volatile atomic<T>* object, T value) noexcept;
template <class T>
void atomic_store_explicit(atomic<T>* object, T value, memory_order,
                           memory_scope) noexcept;
template <class T>
void atomic_store_explicit(volatile atomic<T>* object, T value, memory_order,
                           memory_scope) noexcept;

template <class T>
T atomic_load(atomic<T>* object) noexcept;
template <class T>
T atomic_load(volatile atomic<T>* object) noexcept;
template <class T>
T atomic_load_explicit(atomic<T>* object, memory_order, memory_scope) noexcept;
```

```
template <class T>
T atomic_load_explicit(volatile atomic<T>* object, memory_order,
                       memory_scope) noexcept;

template <class T>
T atomic_exchange(atomic<T>* object, T value) noexcept;
template <class T>
T atomic_exchange(volatile atomic<T>* object, T value) noexcept;
template <class T>
T atomic_exchange_explicit(atomic<T>* object, T value, memory_order,
                           memory_scope) noexcept;
template <class T>
T atomic_exchange_explicit(volatile atomic<T>* object, T value, memory_order,
                           memory_scope) noexcept;

template <class T>
bool atomic_compare_exchange_strong(atomic<T>* object, T* expected,
                                    T desired) noexcept;
template <class T>
bool atomic_compare_exchange_strong(volatile atomic<T>* object, T* expected,
                                    T desired) noexcept;
template <class T>
bool atomic_compare_exchange_strong_explicit(atomic<T>* object, T* expected,
                                             T desired, memory_order,
                                             memory_scope) noexcept;
template <class T>
bool atomic_compare_exchange_strong_explicit(volatile atomic<T>* object,
                                             T* expected, T desired,
                                             memory_order,
                                             memory_scope) noexcept;
template <class T>
bool atomic_compare_exchange_weak(atomic<T>* object, T* expected,
                                  T desired) noexcept;
template <class T>
bool atomic_compare_exchange_weak(volatile atomic<T>* object, T* expected,
                                  T desired) noexcept;
template <class T>
bool atomic_compare_exchange_weak_explicit(atomic<T>* object, T* expected,
                                           T desired, memory_order,
                                           memory_scope) noexcept;
template <class T>
bool atomic_compare_exchange_weak_explicit(volatile atomic<T>* object,
                                           T* expected, T desired,
                                           memory_order, memory_scope) noexcept;

template <class T>
T atomic_fetch_add(atomic<T>* object, T value) noexcept;
template <class T>
T atomic_fetch_add(volatile atomic<T>* object, T value) noexcept;
template <class T>
T atomic_fetch_add_explicit(atomic<T>* object, T value, memory_order,
                            memory_scope) noexcept;
template <class T>
T atomic_fetch_add_explicit(volatile atomic<T>* object, T value, memory_order,
                            memory_scope) noexcept;
template <class T>
T atomic_fetch_and(atomic<T>* object, T value) noexcept;
template <class T>
T atomic_fetch_and(volatile atomic<T>* object, T value) noexcept;
template <class T>
T atomic_fetch_and_explicit(atomic<T>* object, T value, memory_order,
                            memory_scope) noexcept;
template <class T>
T atomic_fetch_and_explicit(volatile atomic<T>* object, T value, memory_order,
                            memory_scope) noexcept;
template <class T>
T atomic_fetch_or(atomic<T>* object, T value) noexcept;
template <class T>
T atomic_fetch_or(volatile atomic<T>* object, T value) noexcept;
template <class T>
T atomic_fetch_or_explicit(atomic<T>* object, T value, memory_order,
                           memory_scope) noexcept;
template <class T>
T atomic_fetch_or_explicit(volatile atomic<T>* object, T value, memory_order,
                           memory_scope) noexcept;
template <class T>
T atomic_fetch_sub(atomic<T>* object, T value) noexcept;
template <class T>
T atomic_fetch_sub(volatile atomic<T>* object, T value) noexcept;
template <class T>
T atomic_fetch_sub_explicit(atomic<T>* object, T value, memory_order,
                            memory_scope) noexcept;
template <class T>
T atomic_fetch_sub_explicit(volatile atomic<T>* object, T value, memory_order,
                            memory_scope) noexcept;
template <class T>
T atomic_fetch_xor(atomic<T>* object, T value) noexcept;
template <class T>
T atomic_fetch_xor(volatile atomic<T>* object, T value) noexcept;
template <class T>
T atomic_fetch_xor_explicit(atomic<T>* object, T value, memory_order,
                            memory_scope) noexcept;
template <class T>
T atomic_fetch_xor_explicit(volatile atomic<T>* object, T value, memory_order,
                            memory_scope) noexcept;

#if (defined(cl_khr_int64_base_atomics) &&
     defined(cl_khr_int64_extended_atomics) &&
     __SIZE_WIDTH__ == 64) ||
    __SIZE_WIDTH__ == 32
template <class T>
T* atomic_fetch_add(atomic<T*>* object, ptrdiff_t value) noexcept;
```

```
template <class T>
T* atomic_fetch_add(volatile atomic<T*>* object, ptrdiff_t value) noexcept;
template <class T>
T* atomic_fetch_add_explicit(atomic<T*>* object, ptrdiff_t value, memory_order,
                             memory_scope) noexcept;
template <class T>
T* atomic_fetch_add_explicit(volatile atomic<T*>* object, ptrdiff_t value,
                             memory_order, memory_scope) noexcept;
template <class T>
T* atomic_fetch_and(atomic<T*>* object, ptrdiff_t value) noexcept;
template <class T>
T* atomic_fetch_and(volatile atomic<T*>* object, ptrdiff_t value) noexcept;
template <class T>
T* atomic_fetch_and_explicit(atomic<T*>* object, ptrdiff_t value, memory_order,
                             memory_scope) noexcept;
template <class T>
T* atomic_fetch_and_explicit(volatile atomic<T*>* object, ptrdiff_t value,
                             memory_order, memory_scope) noexcept;
template <class T>
T* atomic_fetch_or(atomic<T*>* object, ptrdiff_t value) noexcept;
template <class T>
T* atomic_fetch_or(volatile atomic<T*>* object, ptrdiff_t value) noexcept;
template <class T>
T* atomic_fetch_or_explicit(atomic<T*>* object, ptrdiff_t value, memory_order,
                            memory_scope) noexcept;
template <class T>
T* atomic_fetch_or_explicit(volatile atomic<T*>* object, ptrdiff_t value,
                            memory_order, memory_scope) noexcept;
template <class T>
T* atomic_fetch_sub(atomic<T*>* object, ptrdiff_t value) noexcept;
template <class T>
T* atomic_fetch_sub(volatile atomic<T*>* object, ptrdiff_t value) noexcept;
template <class T>
T* atomic_fetch_sub_explicit(atomic<T*>* object, ptrdiff_t value, memory_order,
                             memory_scope) noexcept;
template <class T>
T* atomic_fetch_sub_explicit(volatile atomic<T*>* object, ptrdiff_t value,
                             memory_order, memory_scope) noexcept;
template <class T>
T* atomic_fetch_xor(atomic<T*>* object, ptrdiff_t value) noexcept;
template <class T>
T* atomic_fetch_xor(volatile atomic<T*>* object, ptrdiff_t value) noexcept;
template <class T>
T* atomic_fetch_xor_explicit(atomic<T*>* object, ptrdiff_t value, memory_order,
                             memory_scope) noexcept;
template <class T>
T* atomic_fetch_xor_explicit(volatile atomic<T*>* object, ptrdiff_t value,
                             memory_order, memory_scope) noexcept;
#endif

void atomic_fence(mem_fence flags, memory_order order,
                   memory_scope scope) noexcept;

#define ATOMIC_VAR_INIT(value) as described in cpp14 specification
[atomics.types.operations]

}
```

3.24.2. Order and scope

```
namespace cl
{
enum memory_order
{
    memory_order_relaxed,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};

enum memory_scope
{
    memory_scope_all_svm_devices,
    memory_scope_device,
    memory_scope_work_group,
    memory_scope_sub_group,
    memory_scope_work_item
};

}
```

An enumeration `memory_order` is described in section [atomics.order] of C++14 specification. [7]

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` [8]
- `memory_scope_sub_group`
- `memory_scope_work_group`
- `memory_scope_device`
- `memory_scope_all_svm_devices`

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.

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 C++14 atomic operations [9].

3.24.3. Atomic lock-free property

OpenCL C++ requires all atomic types to be lock free.

3.24.4. Atomic types

```
namespace cl
{

template <class T>
struct atomic
{
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(T, memory_order = memory_order_seq_cst,
               memory_scope = memory_scope_device) volatile noexcept;
    void store(T, memory_order = memory_order_seq_cst,
               memory_scope = memory_scope_device) noexcept;
    T load(memory_order = memory_order_seq_cst,
           memory_scope = memory_scope_device) const volatile noexcept;
    T load(memory_order = memory_order_seq_cst,
           memory_scope = memory_scope_device) const noexcept;
    operator T() const volatile noexcept;
    operator T() const noexcept;
    T exchange(T, memory_order = memory_order_seq_cst,
               memory_scope = memory_scope_device) volatile noexcept;
    T exchange(T, memory_order = memory_order_seq_cst,
               memory_scope = memory_scope_device) noexcept;
    bool compare_exchange_weak(T&, T, memory_order,
                               memory_order, memory_scope) volatile noexcept;
    bool compare_exchange_weak(T&, T, memory_order,
                               memory_order, memory_scope) noexcept;
    bool compare_exchange_strong(T&, T, memory_order,
                                 memory_order, memory_scope) volatile noexcept;
    bool compare_exchange_strong(T&, T, memory_order,
                                 memory_order, memory_scope) noexcept;
    bool compare_exchange_weak(T&, T, memory_order = memory_order_seq_cst,
                          memory_scope = memory_scope_device) volatile noexcept;
    bool compare_exchange_weak(T&, T, memory_order = memory_order_seq_cst,
                               memory_scope = memory_scope_device) noexcept;
    bool compare_exchange_strong(T&, T, memory_order = memory_order_seq_cst,
                          memory_scope = memory_scope_device) volatile noexcept;
    bool compare_exchange_strong(T&, T, memory_order = memory_order_seq_cst,
                                 memory_scope = memory_scope_device) noexcept;
    atomic() noexcept = default;
    constexpr atomic(T) noexcept;
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;
    T operator=(T) volatile noexcept;
    T operator=(T) noexcept;
};

template <>
struct atomic<integral>
{
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(integral, memory_order = memory_order_seq_cst,
               memory_scope = memory_scope_device) volatile noexcept;
    void store(integral, memory_order = memory_order_seq_cst,
               memory_scope = memory_scope_device) noexcept;
    integral load(memory_order = memory_order_seq_cst,
                  memory_scope = memory_scope_device) const volatile noexcept;
    integral load(memory_order = memory_order_seq_cst,
                  memory_scope = memory_scope_device) const noexcept;
    operator integral() const volatile noexcept;
    operator integral() const noexcept;
    integral exchange(integral, memory_order = memory_order_seq_cst,
                      memory_scope = memory_scope_device) volatile noexcept;
    integral exchange(integral, memory_order = memory_order_seq_cst,
                      memory_scope = memory_scope_device) noexcept;
    bool compare_exchange_weak(integral&, integral, memory_order,
                               memory_order, memory_scope) volatile noexcept;
    bool compare_exchange_weak(integral&, integral, memory_order, memory_order,
                               memory_scope) noexcept;
    bool compare_exchange_strong(integral&, integral, memory_order,
                                 memory_order, memory_scope) volatile noexcept;
    bool compare_exchange_strong(integral&, integral, memory_order,
                                 memory_order, memory_scope) noexcept;
    bool compare_exchange_weak(integral&, integral,
                          memory_order = memory_order_seq_cst,
                          memory_scope = memory_scope_device) volatile noexcept;
    bool compare_exchange_weak(integral&, integral,
                               memory_order = memory_order_seq_cst,
                               memory_scope = memory_scope_device) noexcept;
    bool compare_exchange_strong(integral&, integral,
                          memory_order = memory_order_seq_cst,
                          memory_scope = memory_scope_device) volatile noexcept;
    bool compare_exchange_strong(integral&, integral,
                                 memory_order = memory_order_seq_cst,
                                 memory_scope = memory_scope_device) noexcept;
    integral fetch_add(integral, memory_order = memory_order_seq_cst,
                       memory_scope = memory_scope_device) volatile noexcept;
    integral fetch_add(integral, memory_order = memory_order_seq_cst,
                       memory_scope = memory_scope_device) noexcept;
    integral fetch_sub(integral, memory_order = memory_order_seq_cst,
                       memory_scope = memory_scope_device) volatile noexcept;
    integral fetch_sub(integral, memory_order = memory_order_seq_cst,
                       memory_scope = memory_scope_device) noexcept;
    integral fetch_and(integral, memory_order = memory_order_seq_cst,
                       memory_scope = memory_scope_device) volatile noexcept;
    integral fetch_and(integral, memory_order = memory_order_seq_cst,
                       memory_scope = memory_scope_device) noexcept;
    integral fetch_or(integral, memory_order = memory_order_seq_cst,
                      memory_scope = memory_scope_device) volatile noexcept;
    integral fetch_or(integral, memory_order = memory_order_seq_cst,
                      memory_scope = memory_scope_device) noexcept;
    integral fetch_xor(integral, memory_order = memory_order_seq_cst,
                       memory_scope = memory_scope_device) volatile noexcept;
    integral fetch_xor(integral, memory_order = memory_order_seq_cst,
                       memory_scope = memory_scope_device) noexcept;
    integral fetch_min(integral, memory_order = memory_order_seq_cst,
                       memory_scope = memory_scope_device) volatile noexcept;
    integral fetch_min(integral, memory_order = memory_order_seq_cst,
                       memory_scope = memory_scope_device) noexcept;
    integral fetch_max(integral, memory_order = memory_order_seq_cst,
                       memory_scope = memory_scope_device) volatile noexcept;
    integral fetch_max(integral, memory_order = memory_order_seq_cst,
                       memory_scope = memory_scope_device) noexcept;
    atomic() noexcept = default;
    constexpr atomic(integral) noexcept;
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;
    integral operator=(integral) volatile noexcept;
    integral operator=(integral) noexcept;
    integral operator++(int) volatile noexcept;
    integral operator++(int) noexcept;
    integral operator--(int) volatile noexcept;
    integral operator--(int) noexcept;
    integral operator++() volatile noexcept;
    integral operator++() noexcept;
    integral operator--() volatile noexcept;
    integral operator--() noexcept;
    integral operator+=(integral) volatile noexcept;
    integral operator+=(integral) noexcept;
    integral operator-=(integral) volatile noexcept;
    integral operator-=(integral) noexcept;
    integral operator&=(integral) volatile noexcept;
    integral operator&=(integral) noexcept;
    integral operator|=(integral) volatile noexcept;
```

```
    integral operator|=(integral) noexcept;
    integral operator^=(integral) volatile noexcept;
    integral operator^=(integral) noexcept;
};

#if (defined(cl_khr_int64_base_atomics) &&
     defined(cl_khr_int64_extended_atomics) &&
     __SIZE_WIDTH__ == 64) ||
    __SIZE_WIDTH__ == 32
template <class T>
struct atomic<T*>
{
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(T*, memory_order = memory_order_seq_cst,
               memory_scope = memory_scope_device) volatile noexcept;
    void store(T*, memory_order = memory_order_seq_cst,
               memory_scope = memory_scope_device) noexcept;
    T* load(memory_order = memory_order_seq_cst,
            memory_scope = memory_scope_device) const volatile noexcept;
    T* load(memory_order = memory_order_seq_cst,
            memory_scope = memory_scope_device) const noexcept;
    operator T*() const volatile noexcept;
    operator T*() const noexcept;
    T* exchange(T*, memory_order = memory_order_seq_cst,
                memory_scope = memory_scope_device) volatile noexcept;
    T* exchange(T*, memory_order = memory_order_seq_cst,
                memory_scope = memory_scope_device) noexcept;
    bool compare_exchange_weak(T*&, T*, memory_order,
                               memory_order, memory_scope) volatile noexcept;
    bool compare_exchange_weak(T*&, T*, memory_order,
                               memory_order, memory_scope) noexcept;
    bool compare_exchange_strong(T*&, T*, memory_order,
                                 memory_order, memory_scope) volatile noexcept;
    bool compare_exchange_strong(T*&, T*, memory_order,
                                 memory_order, memory_scope) noexcept;
    bool compare_exchange_weak(T*&, T*, memory_order = memory_order_seq_cst,
                          memory_scope = memory_scope_device) volatile noexcept;
    bool compare_exchange_weak(T*&, T*, memory_order = memory_order_seq_cst,
                               memory_scope = memory_scope_device) noexcept;
    bool compare_exchange_strong(T*&, T*, memory_order = memory_order_seq_cst,
                          memory_scope = memory_scope_device) volatile noexcept;
    bool compare_exchange_strong(T*&, T*, memory_order = memory_order_seq_cst,
                                 memory_scope = memory_scope_device) noexcept;
    T* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst,
                 memory_scope = memory_scope_device) volatile noexcept;
    T* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst,
                 memory_scope = memory_scope_device) noexcept;
    T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst,
                 memory_scope = memory_scope_device) volatile noexcept;
    T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst,
                    memory_scope = memory_scope_device) noexcept;
    atomic() noexcept = default;
    constexpr atomic(T*) noexcept;
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;
    T* operator=(T*) volatile noexcept;
    T* operator=(T*) noexcept;
    T* operator++(int) volatile noexcept;
    T* operator++(int) noexcept;
    T* operator--(int) volatile noexcept;
    T* operator--(int) noexcept;
    T* operator++() volatile noexcept;
    T* operator++() noexcept;
    T* operator--() volatile noexcept;
    T* operator--() noexcept;
    T* operator+=(ptrdiff_t) volatile noexcept;
    T* operator+=(ptrdiff_t) noexcept;
    T* operator-=(ptrdiff_t) volatile noexcept;
    T* operator-=(ptrdiff_t) noexcept;
};
#endif

}
```

The *opencl_atomic* header defines general specialization for class template `atomic<T>`.

There are explicit specializations for integral types. Each of these specializations provides set of extra operators suitable for integral types.

There is an explicit specialization of the atomic template for pointer types.

All atomic classes have deleted copy constructor and deleted copy assignment operators.

There are several typedefs for atomic types specified as follows:

```
namespace cl
{
using atomic_int = atomic<int>;
using atomic_uint = atomic<uint>;
#if defined(cl_khr_int64_base_atomics) && defined(cl_khr_int64_extended_atomics)
using atomic_long = atomic<long>;
using atomic_ulong = atomic<ulong>;
#endif
using atomic_float = atomic<float>;
#if defined(cl_khr_fp64) &&
    defined(cl_khr_int64_base_atomics) &&
    defined(cl_khr_int64_extended_atomics)
using atomic_double = atomic<double>;
#endif
#if (defined(cl_khr_int64_base_atomics) &&
     defined(cl_khr_int64_extended_atomics) &&
     __INTPTR_WIDTH__ == 64) ||
    __INTPTR_WIDTH__ == 32
using atomic_intptr_t = atomic<intptr_t>;
using atomic_uintptr_t = atomic<uintptr_t>;
#endif
#if (defined(cl_khr_int64_base_atomics) &&
     defined(cl_khr_int64_extended_atomics) &&
     __SIZE_WIDTH__ == 64) ||
    __SIZE_WIDTH__ == 32
using atomic_size_t = atomic<size_t>;
#endif
#if (defined(cl_khr_int64_base_atomics) &&
     defined(cl_khr_int64_extended_atomics) &&
     __PTRDIFF_WIDTH__ == 64) ||
    __PTRDIFF_WIDTH__ == 32
using atomic_ptrdiff_t = atomic<ptrdiff_t>;
#endif

}
```

3.24.5. Flag type and operations

```
namespace cl
{
struct atomic_flag
{
    bool test_and_set(memory_order = memory_order_seq_cst,
                      memory_scope = memory_scope_device) volatile noexcept;
    bool test_and_set(memory_order = memory_order_seq_cst,
                      memory_scope = memory_scope_device) noexcept;
    void clear(memory_order = memory_order_seq_cst,
               memory_scope = memory_scope_device) volatile noexcept;
    void clear(memory_order = memory_order_seq_cst,
               memory_scope = memory_scope_device) noexcept;
    atomic_flag() noexcept = default;
    atomic_flag(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) volatile = delete;
};

bool atomic_flag_test_and_set(volatile atomic_flag*) noexcept;
bool atomic_flag_test_and_set(atomic_flag*) noexcept;
bool atomic_flag_test_and_set_explicit(volatile atomic_flag*, memory_order,
                                       memory_scope) noexcept;
bool atomic_flag_test_and_set_explicit(atomic_flag*, memory_order,
                                       memory_scope) noexcept;
void atomic_flag_clear(volatile atomic_flag*) noexcept;
void atomic_flag_clear(atomic_flag*) noexcept;
void atomic_flag_clear_explicit(volatile atomic_flag*, memory_order,
                                memory_scope) noexcept;
void atomic_flag_clear_explicit(atomic_flag*, memory_order,
                                memory_scope) noexcept;

#define ATOMIC_FLAG_INIT as described in cpp14 specification [atomics.flag]

}
```

3.24.6. Fences

atomic_fence

```
void atomic_fence(mem_fence flags, memory_order order, memory_scope scope) noexcept;
```

Orders loads or/and stores of a work-item executing a kernel.

`flags` must be set to `mem_fence::global`, `mem_fence::local`, `mem_fence::image` or a combination of these values ORed together; otherwise the behavior is undefined. The behavior of calling `atomic_fence` with `mem_fence::global` and `mem_fence::local` ORed together is equivalent to calling `atomic_fence` individually for each of the fence values set in `flags`. `mem_fence::image` cannot be specified ORed with `mem_fence::global` and `mem_fence::local`.

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 `image_access::read_write`, the `atomic_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. Only a scope of `memory_order_acq_rel` is valid for `atomic_fence` when passed the `mem_fence::image` flag.

3.24.7. 64-bit Atomics

The optional extensions **cl_khr_int64_base_atomics** and **cl_khr_int64_extended_atomics** implement atomic operations on 64-bit signed and unsigned integers to locations in global and local memory.

An application that wants to use 64-bit atomic types will need to define `cl_khr_int64_base_atomics` and `cl_khr_int64_extended_atomics` macros in the code before including the OpenCL C++ standard library headers or using *-D* compiler option (see the *Preprocessor options* section).

3.24.8. Restrictions

- The generic `atomic<T>` class template is only available if `T` is `int`, `uint`, `long`, `ulong` [10], `float`, `double` [11], `intptr_t` [12], `uintptr_t`, `size_t`, `ptrdiff_t`.
- 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 data types cannot be declared inside a kernel or non-kernel function unless they are declared as `static` keyword or in `local<T>` and `global<T>` containers.
- The atomic operations on the private memory can result in undefined behavior.
- `memory_order_consume` is not supported by OpenCL C++.

3.24.9. Examples

Example 1

Examples of using atomic with and without an explicit address space storage class.

```
#include <opencl_memory>
#include <opencl_atomic>
using namespace cl;

atomic<int> a; // OK: atomic in the global memory
local<atomic<int>> b; // OK: atomic in the local memory
global<atomic<int>> c; // OK: atomic in the global memory

kernel void foo() {
    static global<atomic<int>> d; // OK: atomic in the global memory
    atomic<global<int>> e; // error: class members cannot be
                           //        in address space
    local<atomic<int>> f; // OK: atomic in the local memory
    static atomic<int> g; // OK: atomic in the global memory
    atomic<int> h; // undefined behavior
}
```

3.25. Array Library

OpenCL C++ implements part of array library (*chapter 23.3.2*, *[array]*) from the C++14 standard.

For the detailed description please refer to C++14 standard.

3.25.1. Header <opencl_array> Synopsis

```
namespace cl
{
template<class T, size_t N>
struct array
{
    //types:
    typedef T value_type;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef T& reference;
    typedef const T& const_reference;
    typedef implementation-defined iterator;
    typedef implementation-defined const_iterator;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef cl::reverse_iterator<iterator> reverse_iterator;
    typedef cl::reverse_iterator<const_iterator> const_reverse_iterator;

    value_type __elems[N]; // exposition only

    // no explicit construct/copy/destroy for aggregate type

    // iterators:
```

```
    iterator begin() noexcept;
    const_iterator begin() const noexcept;
    iterator end() noexcept;
    const_iterator end() const noexcept;

    reverse_iterator rbegin() noexcept;
    const_reverse_iterator rbegin() const noexcept;
    reverse_iterator rend() noexcept;
    const_reverse_iterator rend() const;

    const_iterator cbegin() const noexcept;
    const_iterator cend() const noexcept;
    const_reverse_iterator crbegin() const noexcept;
    const_reverse_iterator crend() const noexcept;

    // capacity:
    constexpr size_type size() const noexcept;
    constexpr size_type max_size() const noexcept;
    constexpr bool empty() const noexcept;

    // element access:
    reference operator[](size_type n) noexcept;
    const_reference operator[](size_type n) const noexcept;
    reference front() noexcept;
    const_reference front() const noexcept;
    reference back() noexcept;
    const_reference back() const noexcept;

    pointer data();
    const_pointer data() const noexcept;
};

template <class T> class tuple_size;
template <size_t I, class T> class tuple_element;

template <class T, size_t N>
struct tuple_size<array<T, N>>;

template <size_t I, class T, size_t N>
struct tuple_element<I, array<T, N>>;

template <size_t I, class T, size_t N>
constexpr T& get(array<T, N>& a) noexcept;
template <size_t I, class T, size_t N>
constexpr T&& get(array<T, N>&& a) noexcept;
template <size_t I, class T, size_t N>
constexpr const T& get(const array<T, N>& a) noexcept;

}
```

3.26. Limits Library

OpenCL C++ standard library implements modified version of the numeric limits library described in chapter *18.3 [support.limits]* of C++14 standard.

3.26.1. Header <opencl_limits> Synopsis

```
namespace cl
{

template<class T> class numeric_limits;

enum float_round_style;
enum float_denorm_style;

#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

template<> class numeric_limits<bool>;
template<> class numeric_limits<bool2>;
template<> class numeric_limits<bool3>;
template<> class numeric_limits<bool4>;
template<> class numeric_limits<bool8>;
template<> class numeric_limits<bool16>;
template<> class numeric_limits<char>;
template<> class numeric_limits<char2>;
template<> class numeric_limits<char3>;
template<> class numeric_limits<char4>;
template<> class numeric_limits<char8>;
template<> class numeric_limits<char16>;
template<> class numeric_limits<uchar>;
template<> class numeric_limits<uchar2>;
template<> class numeric_limits<uchar3>;
template<> class numeric_limits<uchar4>;
template<> class numeric_limits<uchar8>;
template<> class numeric_limits<uchar16>;
template<> class numeric_limits<short>;
```

```
template<> class numeric_limits<short2>;
template<> class numeric_limits<short3>;
template<> class numeric_limits<short4>;
template<> class numeric_limits<short8>;
template<> class numeric_limits<short16>;
template<> class numeric_limits<ushort>;
template<> class numeric_limits<ushort2>;
template<> class numeric_limits<ushort3>;
template<> class numeric_limits<ushort4>;
template<> class numeric_limits<ushort8>;
template<> class numeric_limits<ushort16>;
template<> class numeric_limits<int>;
template<> class numeric_limits<int2>;
template<> class numeric_limits<int3>;
template<> class numeric_limits<int4>;
template<> class numeric_limits<int8>;
template<> class numeric_limits<int16>;
template<> class numeric_limits<uint>;
template<> class numeric_limits<uint2>;
template<> class numeric_limits<uint3>;
template<> class numeric_limits<uint4>;
template<> class numeric_limits<uint8>;
template<> class numeric_limits<uint16>;
template<> class numeric_limits<long>;
template<> class numeric_limits<long2>;
template<> class numeric_limits<long3>;
template<> class numeric_limits<long4>;
template<> class numeric_limits<long8>;
template<> class numeric_limits<long16>;
template<> class numeric_limits<ulong>;
template<> class numeric_limits<ulong2>;
template<> class numeric_limits<ulong3>;
template<> class numeric_limits<ulong4>;
template<> class numeric_limits<ulong8>;
template<> class numeric_limits<ulong16>;

#ifdef cl_khr_fp16
#define HALF_DIG        3
#define HALF_MANT_DIG   11
#define HALF_MAX_10_EXP +4
#define HALF_MAX_EXP    +16
#define HALF_MIN_10_EXP -4
#define HALF_MIN_EXP    -13
#define HALF_RADIX      2
#define HALF_MAX        0x1.ffcp15h
#define HALF_MIN        0x1.0p-14h
#define HALF_EPSILON    0x1.0p-10h

template<> class numeric_limits<half>;
#endif
```

```
#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
template<> class numeric_limits<float>;

#ifdef cl_khr_fp64
#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

template<> class numeric_limits<double>;
#endif

}
```

3.26.2. Class numeric_limits

```
namespace cl
{
template<class T>
class numeric_limits
{
public:
  static constexpr bool is_specialized = false;
  static constexpr T min() noexcept { return T(); }
  static constexpr T max() noexcept { return T(); }
  static constexpr T lowest() noexcept { return T(); }
  static constexpr int digits = 0;
  static constexpr int digits10 = 0;
  static constexpr int max_digits10 = 0;
  static constexpr bool is_signed = false;
  static constexpr bool is_integer = false;
  static constexpr bool is_exact = false;
  static constexpr int radix = 0;
  static constexpr T epsilon() noexcept { return T(); }
  static constexpr T round_error() noexcept { return T(); }
  static constexpr int min_exponent = 0;
  static constexpr int min_exponent10 = 0;
  static constexpr int max_exponent = 0;
  static constexpr int max_exponent10 = 0;
  static constexpr bool has_infinity = false;
  static constexpr bool has_quiet_NaN = false;
  static constexpr bool has_signaling_NaN = false;
  static constexpr float_denorm_style has_denorm = denorm_absent;
  static constexpr bool has_denorm_loss = false;
  static constexpr T infinity() noexcept { return T(); }
  static constexpr T quiet_NaN() noexcept { return T(); }
  static constexpr T signaling_NaN() noexcept { return T(); }
  static constexpr T denorm_min() noexcept { return T(); }
  static constexpr bool is_iec559 = false;
  static constexpr bool is_bounded = false;
  static constexpr bool is_modulo = false;
  static constexpr bool traps = false;
  static constexpr bool tinyness_before = false;
  static constexpr float_round_style round_style = round_toward_zero;
  static constexpr bool is_scalar = false;
  static constexpr bool is_vector = false;
};

template<class T> class numeric_limits<const T>;
template<class T> class numeric_limits<volatile T>;
template<class T> class numeric_limits<const volatile T>;

}
```

3.26.3. has_denorm numeric_limits class member

has_denorm class member value depends on a macro:

- HAS_SINGLE_FP_DENORM for type float.
- HAS_HALF_FP_DENORM for type half.
- HAS_DOUBLE_FP_DENORM for type double.

If a macro is defined, has_denorm is set to denorm_present. Otherwise it is denorm_absent.

3.26.4. Floating-point macros and limits

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++ library and the corresponding macro names available to the application.

Table 22. Float Built-in Macros

Macro in OpenCL C++	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_EPSILON	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.

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

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

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

Table 23. Double Built-in Macros

Macro in OpenCL cpp	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`

If half precision arithmetic operations are supported, the following macros and constants for half precision floating-point are available:

```
#define HALF_DIG        3
#define HALF_MANT_DIG   11
#define HALF_MAX_10_EXP +4
#define HALF_MAX_EXP    +16
#define HALF_MIN_10_EXP -4
#define HALF_MIN_EXP    -13
#define HALF_RADIX      2
#define HALF_MAX        0x1.ffcp15h
#define HALF_MIN        0x1.0p-14h
#define HALF_EPSILON    0x1.0p-10h
```

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

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

Table 24. Half Built-in Macros

Macro in OpenCL cpp	**Macro for application**
`HALF_DIG`	`CL_HALF_DIG`
`HALF_MANT_DIG`	`CL_HALF_MANT_DIG`
`HALF_MAX_10_EXP`	`CL_HALF_MAX_10_EXP`
`HALF_MAX_EXP`	`CL_HALF_MAX_EXP`
`HALF_MIN_10_EXP`	`CL_HALF_MIN_10_EXP`
`HALF_MIN_EXP`	`CL_HALF_MIN_EXP`
`HALF_RADIX`	`CL_HALF_RADIX`
`HALF_MAX`	`CL_HALF_MAX`
`HALF_MIN`	`CL_HALF_MIN`
`HALF_EPSILON`	`CL_HALF_EPSILON`

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.

Table 25. Float Symbolic Constants

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 constant will also be available:

Table 26. Double Symbolic Constants

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.

3.26.5. Integer macros and limits

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++ library and the corresponding macro names available to the application.

Table 27. Integer built-in macros

Macro in OpenCL cpp	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`

3.27. Math Constants Library

OpenCL C++ implements math constant library. The purpose of this library is to provide the commonly used constants for half, float and double data types.

3.27.1. Header <opencl_math_constants> Synopsis

```
namespace cl
```

```
{
template<class T> class math_constants;

#ifdef cl_khr_fp16
#define M_E_H           see below
#define M_LOG2E_H       see below
#define M_LOG10E_H      see below
#define M_LN2_H         see below
#define M_LN10_H        see below
#define M_PI_H          see below
#define M_PI_2_H        see below
#define M_PI_4_H        see below
#define M_1_PI_H        see below
#define M_2_PI_H        see below
#define M_2_SQRTPI_H    see below
#define M_SQRT2_H       see below
#define M_SQRT1_2_H     see below

template<> class math_constants<half>;
template<> class math_constants<half2>;
template<> class math_constants<half3>;
template<> class math_constants<half4>;
template<> class math_constants<half8>;
template<> class math_constants<half16>;
#endif

#define M_E_F           see below
#define M_LOG2E_F       see below
#define M_LOG10E_F      see below
#define M_LN2_F         see below
#define M_LN10_F        see below
#define M_PI_F          see below
#define M_PI_2_F        see below
#define M_PI_4_F        see below
#define M_1_PI_F        see below
#define M_2_PI_F        see below
#define M_2_SQRTPI_F    see below
#define M_SQRT2_F       see below
#define M_SQRT1_2_F     see below

template<> class math_constants<float>;
template<> class math_constants<float2>;
template<> class math_constants<float3>;
template<> class math_constants<float4>;
template<> class math_constants<float8>;
template<> class math_constants<float16>;

#ifdef cl_khr_fp64
#define M_E         see below
#define M_LOG2E     see below
#define M_LOG10E    see below
#define M_LN2       see below
#define M_LN10      see below
#define M_PI        see below
#define M_PI_2      see below
#define M_PI_4      see below
#define M_1_PI      see below
#define M_2_PI      see below
#define M_2_SQRTPI  see below
#define M_SQRT2     see below
#define M_SQRT1_2   see below

template<> class math_constants<double>;
template<> class math_constants<double2>;
template<> class math_constants<double3>;
template<> class math_constants<double4>;
template<> class math_constants<double8>;
template<> class math_constants<double16>;
#endif

template<class T>
constexpr T e_v = math_constants<T>::e();
template<class T>
constexpr T log2e_v = math_constants<T>::log2e();
template<class T>
constexpr T log10e_v = math_constants<T>::log10e();
template<class T>
constexpr T ln2_v = math_constants<T>::ln2();
template<class T>
constexpr T ln10_v = math_constants<T>::ln10();
template<class T>
constexpr T pi_v = math_constants<T>::pi();
template<class T>
constexpr T pi_2_v = math_constants<T>::pi_2();
template<class T>
constexpr T pi_4_v = math_constants<T>::pi_4();
template<class T>
constexpr T one_pi_v = math_constants<T>::one_pi();
template<class T>
constexpr T two_pi_v = math_constants<T>::two_pi();
template<class T>
constexpr T two_sqrtpi_v = math_constants<T>::two_sqrtpi();
template<class T>
constexpr T sqrt2_v = math_constants<T>::sqrt2();
template<class T>
constexpr T sqrt1_2_v = math_constants<T>::sqrt1_2();

}
```

3.27.2. Class math_constants

```
namespace cl
{
template<class T>
class math_constants
{
public:
  static constexpr T e() noexcept { return T(); }
  static constexpr T log2e() noexcept { return T(); }
  static constexpr T log10e() noexcept { return T(); }
  static constexpr T ln2() noexcept { return T(); }
  static constexpr T ln10() noexcept { return T(); }
  static constexpr T pi() noexcept { return T(); }
  static constexpr T pi_2() noexcept { return T(); }
  static constexpr T pi_4() noexcept { return T(); }
  static constexpr T one_pi() noexcept { return T(); }
  static constexpr T two_pi() noexcept { return T(); }
  static constexpr T two_sqrtpi() noexcept { return T(); }
  static constexpr T sqrt2() noexcept { return T(); }
  static constexpr T sqrt1_2() noexcept { return T(); }
};

}
```

3.27.3. Half Constants

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

Table 28. Half Constants

Constant	Description
`e`	Value of e
`log2e`	Value of $\log_2 e$
`log10e`	Value of $\log_{10} e$
`ln2`	Value of $\log_e 2$
`ln10`	Value of $\log_e 10$
`pi`	Value of π
`pi_2`	Value of π / 2
`pi_4`	Value of π / 4
`one_pi`	Value of 1 / π
`two_pi`	Value of 2 / π
`two_sqrtpi`	Value of 2 / √π
`sqrt2`	Value of √2

Constant	Description
`sqrt1_2`	Value of 1 / √2

3.27.4. Float Constants

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

Table 29. Float Constants

Constant	Description
`e`	Value of e
`log2e`	Value of $\log_2 e$
`log10e`	Value of $\log_{10} e$
`ln2`	Value of $\log_e 2$
`ln10`	Value of $\log_e 10$
`pi`	Value of π
`pi_2`	Value of π / 2
`pi_4`	Value of π / 4
`one_pi`	Value of 1 / π
`two_pi`	Value of 2 / π
`two_sqrtpi`	Value of 2 / √π
`sqrt2`	Value of √2
`sqrt1_2`	Value of 1 / √2

3.27.5. Double Constants

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

Table 30. Double Constants

Constant	Description
`e`	Value of e
`log2e`	Value of $\log_2 e$
`log10e`	Value of $\log_{10} e$
`ln2`	Value of $\log_e 2$
`ln10`	Value of $\log_e 10$
`pi`	Value of π
`pi_2`	Value of π / 2
`pi_4`	Value of π / 4
`one_pi`	Value of 1 / π

Constant	Description
`two_pi`	Value of 2 / π
`two_sqrtpi`	Value of 2 / √π
`sqrt2`	Value of √2
`sqrt1_2`	Value of 1 / √2

3.28. Tuple Library

OpenCL C++ standard library implements most of the tuples described in *chapter 20.4 [tuple]* of C++14 standard.

The following parts of tuple library are not supported:

- allocator related traits (C++14 standard, *section 20.4.2.8*)

3.28.1. Header <opencl_tuple> Synopsis

```
namespace cl
{
// class template tuple:
template <class... Types> class tuple;

// tuple creation functions:
const unspecified ignore;
template <class... Types>
constexpr tuple<VTypes ...> make_tuple(Types&&...);
template <class... Types>
constexpr tuple<Types&&...> forward_as_tuple(Types&&...) noexcept;
template<class... Types>
constexpr tuple<Types&...> tie(Types&...) noexcept;
template <class... Tuples>
constexpr tuple<Ctypes ...> tuple_cat(Tuples&&...);

//  tuple helper classes:
template <class T> class tuple_size; // undefined
template <class T> class tuple_size<const T>;
template <class T> class tuple_size<volatile T>;
template <class T> class tuple_size<const volatile T>;
template <class... Types> class tuple_size<tuple<Types...> >;
template <size_t I, class T> class tuple_element; // undefined
template <size_t I, class T> class tuple_element<I, const T>;
template <size_t I, class T> class tuple_element<I, volatile T>;
template <size_t I, class T> class tuple_element<I, const volatile T>;
template <size_t I, class... Types> class tuple_element<I, tuple<Types...> >;
template <size_t I, class T>
using tuple_element_t = typename tuple_element<I, T>::type;

// element access:
```

```
template <size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...>>&
get(tuple<Types...>&) noexcept;
template <size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...>>&&
get(tuple<Types...>&&) noexcept;
template <size_t I, class... Types>
constexpr const tuple_element_t<I, tuple<Types...>>&
get(const tuple<Types...>&) noexcept;
template <class T, class... Types>
constexpr T& get(tuple<Types...>& t) noexcept;
template <class T, class... Types>
constexpr T&& get(tuple<Types...>&& t) noexcept;
template <class T, class... Types>
constexpr const T& get(const tuple<Types...>& t) noexcept;

// relational operators:
template<class... TTypes, class... UTypes>
constexpr bool operator==(const tuple<TTypes...>&, const tuple<UTypes...>&);
template<class... TTypes, class... UTypes>
constexpr bool operator<(const tuple<TTypes...>&, const tuple<UTypes...>&);
template<class... TTypes, class... UTypes>
constexpr bool operator!=(const tuple<TTypes...>&, const tuple<UTypes...>&);
template<class... TTypes, class... UTypes>
constexpr bool operator>(const tuple<TTypes...>&, const tuple<UTypes...>&);
template<class... TTypes, class... UTypes>
constexpr bool operator<=(const tuple<TTypes...>&, const tuple<UTypes...>&);
template<class... TTypes, class... UTypes>
constexpr bool operator>=(const tuple<TTypes...>&, const tuple<UTypes...>&);

// specialized algorithms:
template <class... Types>
void swap(tuple<Types...>& x, tuple<Types...>& y) noexcept(see cpp14 standard);

}
```

3.29. Type Traits Library

OpenCL C++ supports type traits defined in C++14 specification with following changes:

- the *Unary Type Traits* section describes additions and changes to *UnaryTypeTraits*.
- the *Binary type traits* section describes additions and changes to *BinaryTypeTraits*.
- the *Transformation traits* section describes additions and changes to *TransformationTraits*.

This section specifies only OpenCL specific type traits and modifications. All C++ type traits are described in *chapter 20.10 [meta]* of C++14 standard.

3.29.1. Header <opencl_type_traits> Synopsis

```
namespace cl
{
// helper class:
template <class T, T v> struct integral_constant;
typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;

// primary type categories:
template <class T> struct is_void;
template <class T> struct is_null_pointer;
template <class T> struct is_integral;
template <class T> struct is_floating_point;
template <class T> struct is_array;
template <class T> struct is_pointer;
template <class T> struct is_lvalue_reference;
template <class T> struct is_rvalue_reference;
template <class T> struct is_member_object_pointer;
template <class T> struct is_member_function_pointer;
template <class T> struct is_enum;
template <class T> struct is_union;
template <class T> struct is_class;
template <class T> struct is_function;

// composite type categories:
template <class T> struct is_reference;
template <class T> struct is_arithmetic;
template <class T> struct is_fundamental;
template <class T> struct is_object;
template <class T> struct is_scalar;
template <class T> struct is_compound;
template <class T> struct is_member_pointer;

// type properties:
template <class T> struct is_const;
template <class T> struct is_volatile;
template <class T> struct is_private;
template <class T> struct is_local;
template <class T> struct is_global;
template <class T> struct is_constant;
template <class T> struct is_generic;
template <class T> struct is_vector;
template <class T> struct is_trivial;
template <class T> struct is_trivially_copyable;
template <class T> struct is_standard_layout;
template <class T> struct is_pod;
template <class T> struct is_literal_type;
template <class T> struct is_empty;
template <class T> struct is_polymorphic;
template <class T> struct is_abstract;
template <class T> struct is_final;
template <class T> struct is_signed;
template <class T> struct is_unsigned;
template <class T, class... Args> struct is_constructible;
template <class T> struct is_default_constructible;
template <class T> struct is_copy_constructible;
template <class T> struct is_move_constructible;
template <class T, class U> struct is_assignable;
template <class T> struct is_copy_assignable;
template <class T> struct is_move_assignable;
template <class T> struct is_destructible;
template <class T, class... Args> struct is_trivially_constructible;
template <class T> struct is_trivially_default_constructible;
template <class T> struct is_trivially_copy_constructible;
template <class T> struct is_trivially_move_constructible;
template <class T, class U> struct is_trivially_assignable;
template <class T> struct is_trivially_copy_assignable;
template <class T> struct is_trivially_move_assignable;
template <class T> struct is_trivially_destructible;
template <class T, class... Args> struct is_nothrow_constructible;
template <class T> struct is_nothrow_default_constructible;
template <class T> struct is_nothrow_copy_constructible;
template <class T> struct is_nothrow_move_constructible;
template <class T, class U> struct is_nothrow_assignable;
template <class T> struct is_nothrow_copy_assignable;
template <class T> struct is_nothrow_move_assignable;
template <class T> struct is_nothrow_destructible;
template <class T> struct has_virtual_destructor;

// type property queries:
template <class T> struct alignment_of;
template <class T> struct rank;
template <class T, unsigned I = 0> struct extent;
// type relations:
template <class T, class U> struct is_same;
template <class Base, class Derived> struct is_base_of;
template <class From, class To> struct is_convertible;

// const-volatile modifications:
template <class T> struct remove_const;
template <class T> struct remove_volatile;
template <class T> struct remove_cv;
template <class T> struct add_const;
template <class T> struct add_volatile;
template <class T> struct add_cv;
template <class T>
using remove_const_t = typename remove_const<T>::type;
template <class T>
using remove_volatile_t = typename remove_volatile<T>::type;
template <class T>
```

```
using remove_cv_t = typename remove_cv<T>::type;
template <class T>
using add_const_t = typename add_const<T>::type;
template <class T>
using add_volatile_t = typename add_volatile<T>::type;
template <class T>
using add_cv_t = typename add_cv<T>::type;

// as modifications
template <class T> struct remove_constant;
template <class T> struct remove_local;
template <class T> struct remove_global;
template <class T> struct remove_private;
template <class T> struct remove_generic;
template <class T> struct remove_as;
template <class T> struct remove_attrs;
template <class T> struct add_constant;
template <class T> struct add_local;
template <class T> struct add_global;
template <class T> struct add_private;
template <class T> struct add_generic;
template <class T>
using remove_constant_t = typename remove_constant<T>::type;
template <class T>
using remove_local_t = typename remove_local<T>::type;
template <class T>
using remove_global_t = typename remove_global<T>::type;
template <class T>
using remove_private_t = typename remove_private<T>::type;
template <class T>
using remove_generic_t = typename remove_generic<T>::type;
template <class T>
using remove_as_t = typename remove_as<T>::type;
template <class T>
using remove_attrs_t = typename remove_attrs<T>::type;
template <class T>
using add_constant_t = typename add_constant<T>::type;
template <class T>
using add_local_t = typename add_local<T>::type;
template <class T>
using add_global_t = typename add_global<T>::type;
template <class T>
using add_private_t = typename add_private<T>::type;
template <class T>
using add_generic_t = typename add_generic<T>::type;

// reference modifications:
template <class T> struct remove_reference;
template <class T> struct add_lvalue_reference;
template <class T> struct add_rvalue_reference;
template <class T>
```

```
using remove_reference_t = typename remove_reference<T>::type;
template <class T>
using add_lvalue_reference_t = typename add_lvalue_reference<T>::type;
template <class T>
using add_rvalue_reference_t = typename add_rvalue_reference<T>::type;

// sign modifications:
template <class T> struct make_signed;
template <class T> struct make_unsigned;
template <class T>
using make_signed_t = typename make_signed<T>::type;
template <class T>
using make_unsigned_t = typename make_unsigned<T>::type;

// array modifications:
template <class T> struct remove_extent;
template <class T> struct remove_all_extents;
template <class T>
using remove_extent_t = typename remove_extent<T>::type;
template <class T>
using remove_all_extents_t = typename remove_all_extents<T>::type;

// pointer modifications:
template <class T> struct remove_pointer;
template <class T> struct add_pointer;
template <class T>
using remove_pointer_t = typename remove_pointer<T>::type;
template <class T>
using add_pointer_t = typename add_pointer<T>::type;

// built-in vector queries
template <class T> struct is_vector_type;
template <class T> struct vector_size;

// built-in vector modifications
template <class T> struct vector_element;
template <class T, uint DIM> struct make_vector;
template <class T>
using vector_element_t = typename vector_element<T>::type;
template <class T, uint DIM>
using make_vector_t = typename make_vector<T,DIM>::type;

// other transformations:
template <cl::size_t Len,
cl::size_t Align = default-alignment>
struct aligned_storage;
template <cl::size_t Len, class... Types> struct aligned_union;
template <class T> struct decay;
template <bool, class T = void> struct enable_if;
template <bool, class T, class F> struct conditional;
template <class... T> struct common_type;
template <class T> struct underlying_type;
template <class> class result_of; // not defined
template <class F, class... ArgTypes> class result_of<F(ArgTypes...)>;
template <cl::size_t Len,
cl::size_t Align = default-alignment >
using aligned_storage_t = typename aligned_storage<Len,Align>::type;
template <cl::size_t Len, class... Types>
using aligned_union_t = typename aligned_union<Len,Types...>::type;
template <class T>
using decay_t = typename decay<T>::type;
template <bool b, class T = void>
using enable_if_t = typename enable_if<b,T>::type;
template <bool b, class T, class F>
using conditional_t = typename conditional<b,T,F>::type;
template <class... T>
using common_type_t = typename common_type<T...>::type;
template <class T>
using underlying_type_t = typename underlying_type<T>::type;
template <class T>
using result_of_t = typename result_of<T>::type;
template <class...>
using void_t = void;

}
```

3.29.2. Unary Type Traits

Additional type property predicates

Table 31. Additional type property predicates

Template	Condition
`template <class T> struct is_private;`	Implementation defined.
`template <class T> struct is_local;`	Implementation defined.
`template <class T> struct is_global;`	Implementation defined.
`template <class T> struct is_constant;`	Implementation defined.
`template <class T> struct is_generic;`	Implementation defined.
`template <class T> struct is_vector;`	`T` is built-in vector type.

Additional type property queries

Table 32. Additional type property queries

Template	Value
`template <class T> struct vector_size;`	If `T` names a built-in vector type, an integer value representing number of `T`'s components; otherwise `1`.

3.29.3. Binary type traits

Changed relationships traits

Table 33. Changed relationship traits

Template	Condition
`template <class T, class U> struct is_same;`	`T` and `U` name the same type with the same *cv qualifications*.

3.29.4. Transformation traits

Address space and vector modifications

Table 34. Address space and vector traits

Template	Comments
`template <class T> Xstruct remove_private;`	Implementation defined.
`template <class T> Xstruct remove_local;`	Implementation defined.
`template <class T> Xstruct remove_global;`	Implementation defined.
`template <class T> Xstruct remove_constant;`	Implementation defined.
`template <class T> Xstruct remove_generic;`	Implementation defined.
`template <class T> Xstruct remove_as;`	Implementation defined.
`template <class T> Xstruct remove_attrs;`	Implementation defined.
`template <class T> Xstruct add_private;`	Implementation defined.
`template <class T> Xstruct add_local;`	Implementation defined.
`template <class T> Xstruct add_global;`	Implementation defined.
`template <class T> Xstruct add_constant;`	Implementation defined.
`template <class T> Xstruct add_generic;`	Implementation defined.
`template <class T> Xstruct vector_element;`	If `T` is a built-in vector type, member typedef `T` shall name type of `T`'s component; otherwise it shall name `T`.
`template <class T, size_t Dim> struct make_vector;`	If type `U` exists and names a built-in vector type with `Dim` components of type `T`, member typedef type shall name `U`; otherwise it shall name `T`.

3.30. Iterator Library

OpenCL C++ implements part of iterator library (*chapter 24, [iterators]*) from the C++14 standard. Primitives (C++14 standard, *section 24.4*), iterator operations (C++14 standard, *section 24.4.4,*), predefined iterators (C++14 standard, *section 24.5*) and range access (C++14 standard, *section 24.7*) are supported.

For the detailed description please refer to C++14 standard.

3.30.1. Header <opencl_iterator> Synopsis

```
namespace cl
{
// primitives:
template<class Iterator> struct iterator_traits;
template<class T> struct iterator_traits<T*>;
template<class Category, class T, class Distance = ptrdiff_t,
class Pointer = T*, class Reference = T&> struct iterator;

struct input_iterator_tag { };
struct output_iterator_tag { };
struct forward_iterator_tag: public input_iterator_tag { };
struct bidirectional_iterator_tag: public forward_iterator_tag { };
struct random_access_iterator_tag: public bidirectional_iterator_tag { };

// iterator operations:
template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n);
template <class InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
template <class ForwardIterator>
ForwardIterator next(ForwardIterator x,
typename std::iterator_traits<ForwardIterator>::difference_type n = 1);
template <class BidirectionalIterator>
BidirectionalIterator prev(BidirectionalIterator x,
typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);

// predefined iterators:
template <class Iterator> class reverse_iterator;
template <class Iterator1, class Iterator2>
bool operator==(
const reverse_iterator<Iterator1>& x,
const reverse_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
bool operator<(
const reverse_iterator<Iterator1>& x,
const reverse_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
bool operator!=(
const reverse_iterator<Iterator1>& x,
const reverse_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
bool operator>(
const reverse_iterator<Iterator1>& x,
const reverse_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
bool operator>=(
const reverse_iterator<Iterator1>& x,
const reverse_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
bool operator<=(
const reverse_iterator<Iterator1>& x,
const reverse_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
auto operator-(
const reverse_iterator<Iterator1>& x,
const reverse_iterator<Iterator2>& y) ->decltype(y.base() - x.base());
template <class Iterator>
reverse_iterator<Iterator>
operator+(typename reverse_iterator<Iterator>::difference_type n,
          const reverse_iterator<Iterator>& x);
template <class Iterator>
reverse_iterator<Iterator> make_reverse_iterator(Iterator i);
template <class Container> class back_insert_iterator;
template <class Container>
back_insert_iterator<Container> back_inserter(Container& x);
template <class Container> class front_insert_iterator;
template <class Container>
front_insert_iterator<Container> front_inserter(Container& x);
template <class Container> class insert_iterator;
template <class Container>
insert_iterator<Container> inserter(Container& x,
                                    typename Container::iterator i);
template <class Iterator> class move_iterator;
template <class Iterator1, class Iterator2>
bool operator==(
const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
bool operator!=(
const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
bool operator<(
const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
bool operator<=(
const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
bool operator>(
const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
bool operator>=(
const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class Iterator1, class Iterator2>
auto operator-(
const move_iterator<Iterator1>& x,
const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());
template <class Iterator>
move_iterator<Iterator> operator+(
                            typename move_iterator<Iterator>::difference_type n,
                            const move_iterator<Iterator>& x);
template <class Iterator>
move_iterator<Iterator> make_move_iterator(Iterator i);

// range access:
template <class C> auto begin(C& c) -> decltype(c.begin());
template <class C> auto begin(const C& c) -> decltype(c.begin());
template <class C> auto end(C& c) -> decltype(c.end());
template <class C> auto end(const C& c) -> decltype(c.end());
template <class T, size_t N> constexpr T* begin(T (&array)[N]) noexcept;
template <class T, size_t N> constexpr T* end(T (&array)[N]) noexcept;
template <class C>
constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
-> decltype(std::begin(c));
template <class C>
constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
-> decltype(std::end(c));
template <class C> auto rbegin(C& c) -> decltype(c.rbegin());
template <class C> auto rbegin(const C& c) -> decltype(c.rbegin());
template <class C> auto rend(C& c) -> decltype(c.rend());
template <class C> auto rend(const C& c) -> decltype(c.rend());
template <class T, size_t N> reverse_iterator<T*> rbegin(T (&array)[N]);
template <class T, size_t N> reverse_iterator<T*> rend(T (&array)[N]);
template <class E> reverse_iterator<const E*> rbegin(initializer_list<E> il);
template <class E> reverse_iterator<const E*> rend(initializer_list<E> il);
template <class C> auto crbegin(const C& c) -> decltype(std::rbegin(c));
template <class C> auto crend(const C& c) -> decltype(std::rend(c));

}
```

3.31. General Utilities Library

OpenCL C++ implements part of utility library (*chapter 20.2, [utilities]*) from the C++14 standard.

For the detailed description please refer to C++14 standard.

3.31.1. Header <opencl_utility> Synopsis

```
namespace cl
{
template<class T>
add_rvalue_reference_t<T> declval( );

template <class T>
constexpr T&& forward(typename remove_reference_t& t) noexcept;
template <class T>
constexpr T&& forward(typename remove_reference_t&& t) noexcept;
template <class T>
constexpr typename remove_reference_t&& move(T&& t) noexcept;

template<class T>
void swap(T& a, T& b) noexcept;

template <class T>
  constexpr conditional_t<
  !is_nothrow_move_constructible<T>::value && is_copy_constructible<T>::value,
  const T&, T&&> move_if_noexcept(T& x) noexcept;

}
```

Chapter 4. OpenCL Numerical Compliance

This section describes features of the C++14 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 and half precision floating-point is a requirement. Double precision floating-point is an optional feature.

4.1. 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 +infinity.
- Round toward -infinity.
- Round toward zero.

Round to nearest even is currently the only rounding mode required [30] 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. Static and dynamic selection of rounding mode is not supported.

4.2. 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.

4.3. 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 ISO / IEC 9899 : TC2.

4.4. 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 specified in the *Conversions Library* section 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.

> Jean-Michel Muller. On the definition of ulp(x). RR-5504, INRIA. 2005, pp.16. <inria-00070503> Currently hosted at https://hal.inria.fr/inria-00070503/document.

ULP values for single precision built-in math functions [31] table 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.

Table 35. ULP values for single precision built-in math functions

Function	**Min Accuracy - ULP values [32]**
x + *y*	Correctly rounded
x - *y*	Correctly rounded
x * *y*	Correctly rounded
1.0 / *x*	<= 2.5 ulp
x / *y*	<= 2.5 ulp

Function	Min Accuracy - ULP values [32]
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

Function	Min Accuracy - ULP values [32]
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
native_math::cos	Implementation-defined
native_math::divide	Implementation-defined
native_math::exp	Implementation-defined
native_math::exp2	Implementation-defined
native_math::exp10	Implementation-defined

Function	Min Accuracy - ULP values [32]
native_math::log	Implementation-defined
native_math::log2	Implementation-defined
native_math::log10	Implementation-defined
native_math::powr	Implementation-defined
native_math::recip	Implementation-defined
native_math::rsqrt	Implementation-defined
native_math::sin	Implementation-defined
native_math::sqrt	Implementation-defined
native_math::tan	Implementation-defined

ULP values for single precision builtin math functions for embedded profile 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 36. ULP values for single precision built-in math functions for embedded profile

Function	Min Accuracy - ULP values [32]
$x + y$	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

Function	Min Accuracy - ULP values [32]
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

Function	Min Accuracy - ULP values [32]
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_math::cos	Implementation-defined
native_math::divide	Implementation-defined
native_math::exp	Implementation-defined
native_math::exp2	Implementation-defined
native_math::exp10	Implementation-defined
native_math::log	Implementation-defined
native_math::log2	Implementation-defined
native_math::log10	Implementation-defined
native_math::powr	Implementation-defined

Function	Min Accuracy - ULP values [32]
native_math::recip	Implementation-defined
native_math::rsqrt	Implementation-defined
native_math::sin	Implementation-defined
native_math::sqrt	Implementation-defined
native_math::tan	Implementation-defined

ULP values for single precision built-in math functions with unsafe math optimizations 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 ULP values for single precision built-in math functions with unsafe math optimizations table. The minimum accuracy of math functions not defined in ULP values for single precision built-in math functions with unsafe math optimizations table when the *-cl-unsafe-math-optimizations* compiler option is specified is as defined in ULP values for single precision built-in math functions table when operating in the full profile, and as defined in ULP values for single precision built-in math functions for embedded profile table 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 37. ULP values for single precision built-in math functions with unsafe math optimizations in the full and embedded profiles

Function	Min Accuracy - ULP values [32]
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_PI_F for x < 0 and y > 0 and atan(y/x) - M_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.

Function	**Min Accuracy - ULP values** [32]
atan2pi(y, x)	Implemented as atan2(y, x) * M_1_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)).
atanh(x)	Defined for x in the domain (-1, 1). For x in [-2^{-10}, 2^{-10}], implemented as x. For x outside of [-2^{-10}, 2^{-10}], implemented as 0.5f * log((1.0f + x) / (1.0f - x)). For non-derived implementations, the error is <= 8192 ulp.
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

Function	**Min Accuracy - ULP values** [32]
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(fabs(x))). For x < 0 and odd y, derived implementations implement this as -exp2(y * log2(fabs(x)) [33]. 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(fabs(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 non-zero, 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.

Function	Min Accuracy - ULP values [32]
tanh	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 implementations implement as (exp(x) - exp(-x)) / (exp(x) + exp(-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.

ULP values for double precision built-in math functions 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 38. ULP values for double precision built-in math functions

Function	Min Accuracy - ULP values [32]
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

Function	**Min Accuracy - ULP values [32]**
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

Function	Min Accuracy - ULP values [32]
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	Correctly rounded
tan	<= 5 ulp
tanh	<= 5 ulp
tanpi	<= 6 ulp
tgamma	<= 16 ulp
trunc	Correctly rounded

ULP values for half precision built-in math functions table describes the minimum accuracy of half 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 half precision built-in math functions

Function	Min Accuracy - ULP values [32]
x + y	Correctly rounded
x - y	Correctly rounded
x * y	Correctly rounded
1.0 / x	Correctly rounded
x / y	Correctly rounded
acos	<= 2 ulp
acospi	<= 2 ulp
asin	<= 2 ulp
asinpi	<= 2 ulp
atan	<= 2 ulp
atan2	<= 2 ulp
atanpi	<= 2 ulp
atan2pi	<= 2 ulp
acosh	<= 2 ulp

Function	Min Accuracy - ULP values [32]
asinh	<= 2 ulp
atanh	<= 2 ulp
cbrt	<= 2 ulp
ceil	Correctly rounded
copysign	0 ulp
cos	<= 2 ulp
cosh	<= 2 ulp
cospi	<= 2 ulp
erfc	<= 4 ulp
erf	<= 4 ulp
exp	<= 2 ulp
exp2	<= 2 ulp
exp10	<= 2 ulp
expm1	<= 2 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	<= 2 ulp
ilogb	0 ulp
ldexp	Correctly rounded
log	<= 2 ulp
log2	<= 2 ulp
log10	<= 2 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

Function	Min Accuracy - ULP values [32]
modf	0 ulp
nan	0 ulp
nextafter	0 ulp
pow(x, y)	<= 4 ulp
pown(x, y)	<= 4 ulp
powr(x, y)	<= 4 ulp
remainder	0 ulp
remquo	0 ulp
rint	Correctly rounded
rootn	<= 4 ulp
round	Correctly rounded
rsqrt	<= 1 ulp
sin	<= 2 ulp
sincos	<= 2 ulp for sine and cosine values
sinh	<= 2 ulp
sinpi	<= 2 ulp
sqrt	Correctly rounded
tan	<= 2 ulp
tanh	<= 2 ulp
tanpi	<= 2 ulp
tgamma	<= 4 ulp
trunc	Correctly rounded

4.5. Edge Case Behavior

The edge case behavior of the math functions (see the *Math Functions* section) shall conform to sections F.9 and G.6 of ISO/IEC 9899:TC 2, except where noted below in the *Additional Requirements Beyond ISO/IEC 9899:TC2* section.

4.5.1. Additional Requirements Beyond ISO/IEC 9899: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.

The usual allowances for rounding error (see the *Relative Error as ULPs* section) or flushing behavior (see the *Edge Case Behavior in Flush To Zero Mode* section) shall not apply for those values for which *section F.9* of ISO/IEC 9899:,TC2, or the *Additional Requirements Beyond ISO/IEC 9899:TC2* and the *Edge Case Behavior in Flush To Zero Mode* sections 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(±0) returns 1.
- 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(+inf, iptr) returns +0 and +inf in iptr.
- fract(-inf, iptr) returns -0 and -inf 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.

4.5.2. Changes to ISO/IEC 9899: 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.

4.5.3. 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 Beyond ISO/IEC 9899:TC2* 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)`. [35]

Chapter 5. 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 `image::read`. The sampler specified in `image::read` is used to determine how to sample the image and return an appropriate color.

5.1. Image Coordinates

This affects the interpretation of image coordinates. If image coordinates specified to `image::read` 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 `image::read` must always be the unnormalized image coordinate value.

Let (u,v,w) represent the unnormalized image coordinate values.

5.2. 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 `addressing_mode::repeat` nor `addressing_mode::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 `image::read` is undefined.

5.2.1. filtering_mode::nearest

When filter mode is `filtering_mode::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

$$\begin{aligned} i &= address\_mode((int)floor(u)) \\ j &= address\_mode((int)floor(v)) \\ k &= address\_mode((int)floor(w)) \end{aligned}$$

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 below table describes the `address_mode` function.

Table 40. Addressing modes to generate texel location

Addressing Mode	Result of address_mode(coord)
`clamp_to_edge`	clamp (coord, 0, size - 1)

Addressing Mode	**Result of address_mode(coord)**
`clamp`	clamp (coord, -1, size)
`none`	Coord

The size term in Addressing modes to generate texel location table is w_t for u, h_t for v and d_t for w.

The clamp function used in Addressing modes to generate texel location 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.

5.2.2. filtering_mode::linear

When filter mode is `filtering_mode::linear`, a 2 x 2 square of image elements for a 2D image or a 2 x 2 x 2 cube of image elements for a 3D image is selected. This 2 x 2 square or 2 x 2 x 2 cube is obtained as follows.

Let

$$\begin{aligned}
i0 &= address\_mode((int)floor(u - 0.5))\\
j0 &= address\_mode((int)floor(v - 0.5))\\
k0 &= address\_mode((int)floor(w - 0.5))\\
i1 &= address\_mode((int)floor(u - 0.5) + 1)\\
j1 &= address\_mode((int)floor(v - 0.5) + 1)\\
k1 &= address\_mode((int)floor(w - 0.5) + 1)\\
a &= frac(u - 0.5)\\
b &= frac(v - 0.5)\\
c &= frac(w - 0.5)
\end{aligned}$$

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

$$\begin{aligned}
T \quad = \quad & (1-a)*(1-b)*(1-c)*T_{i0\,j0k0}\\
& + a*(1-b)*(1-c)*T_{i1\,j0k0}\\
& + (1-a)*b*(1-c)*T_{i0\,j1k0}\\
& + a*b*(1-c)*T_{i1\,j1k0}\\
& + (1-a)*(1-b)*c*T_{i0\,j0k1}\\
& + a*(1-b)*c*T_{i1\,j0k1}\\
& + (1-a)*b*c*T_{i0\,j1k1}\\
& + a*b*c*T_{i1\,j1k1}
\end{aligned}$$

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

$$\begin{aligned} T = {} & (1-a)*(1-b)*T_{i0j0} \\ & + a*(1-b)*T_{i1j0} \\ & + (1-a)*b*T_{i0j1} \\ & + a*b*T_{i1j1} \end{aligned}$$

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 `addressing_mode::repeat`.

If values in (s,t,r) are INF or NaN, the behavior of the built-in image read functions is undefined.

5.2.3. filtering_mode::nearest

When filter mode is `filtering_mode::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.

5.2.4. filtering_mode::linear

When filter mode is `filtering_mode::linear`, a 2 x 2 square of image elements for a 2D image or a 2 x 2 x 2 cube of image elements for a 3D image is selected. This 2 x 2 square or 2 x 2 x 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

$$\begin{aligned} T \quad = \quad & (1-a)*(1-b)*(1-c)*T_{i0\,j0k0} \\ & + a*(1-b)*(1-c)*T_{i1\,j0k0} \\ & + (1-a)*b*(1-c)*T_{i0\,j1k0} \\ & + a*b*(1-c)*T_{i1\,j1k0} \\ & + (1-a)*(1-b)*c*T_{i0\,j0k1} \\ & + a*(1-b)*c*T_{i1\,j0k1} \\ & + (1-a)*b*c*T_{i0\,j1k1} \\ & + a*b*c*T_{i1\,j1k1} \end{aligned}$$

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

$$\begin{aligned} T \quad = \quad & (1-a)*(1-b)*T_{i0\,j0} \\ & + a*(1-b)*T_{i1\,j0} \\ & + (1-a)*b*T_{i0\,j1} \\ & + a*b*T_{i1\,j1} \end{aligned}$$

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 `addressing_mode::repeat`. The `addressing_mode::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.

5.2.5. filtering_mode::nearest

When filter mode is `filtering_mode::nearest`, the image element at location (i,j,k) becomes the image element value, with i,j and k computed as

$$\begin{aligned}
s' &= 2.0f * rint(0.5f * s) \\
s' &= fabs(s - s') \\
u &= s' * w_t \\
i &= (int)floor(u) \\
i &= min(i, w_t - 1) \\
t' &= 2.0f * rint(0.5f * t) \\
t' &= fabs(t - t') \\
v &= t' * h_t \\
j &= (int)floor(v) \\
j &= min(j, h_t - 1) \\
r' &= 2.0f * rint(0.5f * r) \\
r' &= fabs(r - r') \\
w &= r' * d_t \\
k &= (int)floor(w) \\
k &= min(k, d_t - 1)
\end{aligned}$$

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.

5.2.6. filtering_mode::linear

When filter mode is `filtering_mode::linear`, a 2 x 2 square of image elements for a 2D image or a 2 x 2 x 2 cube of image elements for a 3D image is selected. This 2 x 2 square or 2 x 2 x 2 cube is obtained as follows.

Let

$$
\begin{aligned}
s' &= 2.0f * rint(0.5f * s)\\
s' &= fabs(s - s')\\
u &= s' * w_t\\
i0 &= (int)floor(u - 0.5f)\\
i1 &= i0 + 1\\
i0 &= max(i0, 0)\\
i1 &= min(i1, w_t - 1)\\
t' &= 2.0f * rint(0.5f * t)\\
t' &= fabs(t - t')\\
v &= t' * h_t\\
j0 &= (int)floor(v - 0.5f)\\
j1 &= j0 + 1\\
j0 &= max(j0, 0)\\
j1 &= min(j1, h_t - 1)\\
r' &= 2.0f * rint(0.5f * r)\\
r' &= fabs(r - r')\\
w &= r' * d_t\\
k0 &= (int)floor(w - 0.5f)\\
k1 &= k0 + 1\\
k0 &= max(k0, 0)\\
k1 &= min(k1, d_t - 1)\\
a &= frac(u - 0.5)\\
b &= frac(v - 0.5)\\
c &= frac(w - 0.5)
\end{aligned}
$$

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

$$
\begin{aligned}
T = {} & (1-a)*(1-b)*(1-c)*T_{i0j0k0}\\
& + a*(1-b)*(1-c)*T_{i1j0k0}\\
& + (1-a)*b*(1-c)*T_{i0j1k0}\\
& + a*b*(1-c)*T_{i1j1k0}\\
& + (1-a)*(1-b)*c*T_{i0j0k1}\\
& + a*(1-b)*c*T_{i1j0k1}\\
& + (1-a)*b*c*T_{i0j1k1}\\
& + a*b*c*T_{i1j1k1}
\end{aligned}
$$

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

$$
\begin{aligned}
T = {} & (1-a)*(1-b)*T_{i0j0}\\
& + a*(1-b)*T_{i1j0}\\
& + (1-a)*b*T_{i0j1}\\
& + a*b*T_{i1j1}
\end{aligned}
$$

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 `filtering_mode::nearest` and addressing mode set to one of the following modes - `addressing_mode::none`, `addressing_mode::clamp_to_edge` or `addressing_mode::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 `image::read` with a sampler that uses unnormalized coordinates, filter mode set to `filtering_mode::nearest`, addressing mode set to `addressing_mode::none`, `addressing_mode::clamp_to_edge` or `addressing_mode::clamp` and finally performing the interpolation of color values read from the image to generate the filtered color value.

5.3. Conversion Rules

In this section we discuss conversion rules that are applied when reading and writing images in a kernel.

5.3.1. Conversion rules for normalized integer

In this section we discuss converting normalized integer channel data types to half-precision and single-precision floating-point values and vice-versa.

Converting normalized integer channel data types to half precision floating-point values

For images created with image channel data type of `CL_UNORM_INT8` and `CL_UNORM_INT16`, `image::read` will convert the channel values from an 8-bit or 16-bit unsigned integer to normalized half precision floating-point values in the range [0.0h … 1.0h].

For images created with image channel data type of `CL_SNORM_INT8` and `CL_SNORM_INT16`, `image::read` will convert the channel values from an 8-bit or 16-bit signed integer to normalized half precision floating-point values in the range [-1.0h … 1.0h].

These conversions are performed as follows:

- `CL_UNORM_INT8` (8-bit unsigned integer) → `half`

$$normalized\_half\_value(x) = round\_to\_half(\frac{x}{255})$$

- `CL_UNORM_INT_101010` (10-bit unsigned integer) → `half`

$$normalized\_half\_value(x) = round\_to\_half(\frac{x}{1023})$$

- `CL_UNORM_INT16` (16-bit unsigned integer) → `half`

$$normalized\_half\_value(x) = round\_to\_half(\frac{x}{65535})$$

- `CL_SNORM_INT8` (8-bit signed integer) → `half`

$$normalized\_half\_value(x) = max(-1.0h, round\_to\_half(\frac{x}{127}))$$

- `CL_SNORM_INT16` (16-bit signed integer) → `half`

$$normalized\_half\_value(x) = max(-1.0h, round\_to\_half(\frac{x}{32767}))$$

The precision of the above conversions is <= 1.5 ulp except for the following cases.

For `CL_UNORM_INT8`:

- 0 must convert to 0.0h and
- 255 must convert to 1.0h

For `CL_UNORM_INT_101010`:

- 0 must convert to 0.0h and
- 1023 must convert to 1.0h

For `CL_UNORM_INT16`:

- 0 must convert to 0.0h and
- 65535 must convert to 1.0h

For `CL_SNORM_INT8`:

- -128 and -127 must convert to -1.0h,
- 0 must convert to 0.0h and
- 127 must convert to 1.0h

For `CL_SNORM_INT16`:

- -32768 and -32767 must convert to -1.0h,
- 0 must convert to 0.0h and
- 32767 must convert to 1.0h

Converting half precision 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`, `image::write` will convert the half precision 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`, `image::write` will convert the half precision floating-point color value to an 8-bit or 16-bit signed integer.

OpenCL implementations may choose to approximate the rounding mode used in the conversions described below. When approximate rounding is used instead of the preferred rounding, the result of the conversion must satisfy the bound given below.

The conversions from half precision floating-point values to normalized integer values are performed is as follows:

- `half` → `CL_UNORM_INT8` (8-bit unsigned integer)

$$f(x) = max(0, min(255,255 \times x))$$

$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_uint8(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$f_{approx}(x) = \begin{cases} round\_to\_impl\_uint8(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$|f(x) - f_{approx}(x)| \leq 0.6,\ x \neq \infty \text{and} x \neq NaN$$

- `half` → `CL_UNORM_INT16` (16-bit unsigned integer)

$$f(x) = max(0, min(65535, 65535 \times x))$$

$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_uint16(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$f_{approx}(x) = \begin{cases} round\_to\_impl\_uint16(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$|f(x) - f_{approx}(x)| \leq 0.6,\ x \neq \infty \text{and} x \neq NaN$$

- `half` → `CL_SNORM_INT8` (8-bit signed integer)

$$f(x) = max(-128, min(127,127 \times x))$$

$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_uint8(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$f_{approx}(x) = \begin{cases} round\_to\_impl\_uint8(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$|f(x) - f_{approx}(x)| \leq 0.6,\ x \neq \infty \text{and} x \neq NaN$$

- `half` → `CL_SNORM_INT16` (16-bit signed integer)

$$f(x) = max(-32768, min(32767, 32767 \times x))$$

$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_uint16(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$f_{approx}(x) = \begin{cases} round\_to\_impl\_uint16(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$|f(x) - f_{approx}(x)| \leq 0.6,\ x \neq \infty \text{and} x \neq NaN$$

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`, `image::read` will convert the channel values from an 8-bit or 16-bit unsigned integer to normalized floating-point values in the range [0.0f … 1.0f].

For images created with image channel data type of `CL_SNORM_INT8` and `CL_SNORM_INT16`, `image::read` will convert the channel values from an 8-bit or 16-bit signed integer to normalized floating-point values in the range [-1.0f … 1.0f].

These conversions are performed as follows:

- `CL_UNORM_INT8` (8-bit unsigned integer) → `float`

$$normalized\_float\_value(x) = round\_to\_float(\frac{x}{255})$$

- `CL_UNORM_INT_101010` (10-bit unsigned integer) → `float`

$$normalized\_float\_value(x) = round\_to\_float(\frac{x}{1023})$$

- `CL_UNORM_INT16` (16-bit unsigned integer) → `float`

$$normalized\_float\_value(x) = round\_to\_float(\frac{x}{65535})$$

- `CL_SNORM_INT8` (8-bit signed integer) → `float`

$$normalized\_float\_value(x) = max(-1.0f, round\_to\_float(\frac{x}{127}))$$

- `CL_SNORM_INT16` (16-bit signed integer) → `float`

$$normalized\_float\_value(x) = max(-1.0f, round\_to\_float(\frac{x}{32767}))$$

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`, `image::write` 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`, `image::write` will convert the floating-point color value to an 8-bit or 16-bit signed integer.

OpenCL implementations may choose to approximate the rounding mode used in the conversions described below. When approximate rounding is used instead of the preferred rounding, the result of the conversion must satisfy the bound given below.

The conversions from half precision floating-point values to normalized integer values are performed is as follows:

- `float` → `CL_UNORM_INT8` (8-bit unsigned integer)

$$f(x) = max(0, min(255,255 \times x))$$

$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_uint8(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$f_{approx}(x) = \begin{cases} round\_to\_impl\_uint8(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$|f(x) - f_{approx}(x)| \leq 0.6,\ x \neq \infty \text{and} x \neq NaN$$

- `float` → `CL_UNORM_INT_101010` (10-bit unsigned integer)

$$f(x) = max(0, min(1023, 1023 \times x))$$

$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_uint10(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$f_{approx}(x) = \begin{cases} round\_to\_impl\_uint10(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$|f(x) - f_{approx}(x)| \leq 0.6,\ x \neq \infty \text{and} x \neq NaN$$

- `float` → `CL_UNORM_INT16` (16-bit unsigned integer)

$$f(x) = max(0, min(65535, 65535 \times x))$$

$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_uint16(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$f_{approx}(x) = \begin{cases} round\_to\_impl\_uint16(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$|f(x) - f_{approx}(x)| \leq 0.6,\ x \neq \infty \text{and} x \neq NaN$$

- `float` → `CL_SNORM_INT8` (8-bit signed integer)

$$f(x) = max(-128, min(127, 127 \times x))$$

$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_uint8(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$f_{approx}(x) = \begin{cases} round\_to\_impl\_uint8(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$|f(x) - f_{approx}(x)| \leq 0.6,\ x \neq \infty \text{and} x \neq NaN$$

- `float` → `CL_SNORM_INT16` (16-bit signed integer)

$$f(x) = max(-32768, min(32767, 32767 \times x))$$

$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_uint16(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$f_{approx}(x) = \begin{cases} round\_to\_impl\_uint16(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$|f(x) - f_{approx}(x)| \leq 0.6,\ x \neq \infty \text{and} x \neq NaN$$

5.3.2. Conversion rules for half precision floating-point channel data type

For images created with a channel data type of `CL_HALF_FLOAT`, the conversions of half to float and half to half are lossless (as described in the *Built-in Half Data Type* section). 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.

5.3.3. 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.

5.3.4. Conversion rules for signed and unsigned 8-bit, 16-bit and 32-bit integer channel data types

Calls to `image::read` 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 `image::read` 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 `image::write` will perform one of the following conversions:

32 bit signed integer → 8-bit signed integer

```
convert_cast<char,saturate::on>(i)
```

32 bit signed integer → 16-bit signed integer

```
convert_cast<short,saturate::on>(i)
```

32 bit signed integer → 32-bit signed integer

```
no conversion is performed
```

Calls to image::write will perform one of the following conversions:

32 bit unsigned integer → 8-bit unsigned integer

```
convert_cast<uchar,saturate::on>(i)
```

32 bit unsigned integer → 16-bit unsigned integer

```
convert_cast<ushort,saturate::on>(i)
```

32 bit unsigned integer → 32-bit unsigned integer

```
no conversion is performed
```

The conversions described in this section must be correctly saturated.

5.3.5. 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 image::read 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 image::sample 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 image::write 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.

1. The following process is used by image::read and image::sample to convert a normalized 8-bit unsigned integer sRGB color value x to a floating-point linear RGB color value y:

   a. Convert a normalized 8-bit unsigned integer sRGB value x to a floating-point sRGB value r as per rules described in the *Converting floating-point values to normalized integer channel data types* section.

   $$r = normalized\_float\_value(x)$$

   b. Convert a floating-point sRGB value r to a floating-point linear RGB color value y:

   $$c_{linear}(x) = \begin{cases} \frac{r}{12.92} & r \geq 0 \text{and} r \leq 0.04045 \\ (\frac{r + 0.055}{1.055})^{2.4} & r > 0.04045 \text{and} \leq 1 \end{cases}$$

   $$y = c_{linear}(r)$$

2. The following process is used by `image::write` to convert a linear RGB floating-point color value y to a normalized 8-bit unsigned integer sRGB value x:

   a. Convert a floating-point linear RGB value y to a normalized floating point sRGB value r:

   $$c_{linear}(x) = \begin{cases} 0 & y \geq NaN \text{or} y < 0 \\ 12.92 \times y & y \geq 0 \text{and} y < 0.0031308 \\ 1.055 \times y^{(\frac{1}{2.4})} & y \geq 0.0031308 \text{and} y \leq 1 \\ 1 & y > 1 \end{cases}$$

   $$r = c_{sRGB}(y)$$

   b. Convert a normalized floating-point sRGB value r to a normalized 8-bit unsigned integer sRGB value x as per rules described in Converting normalized integer channel data types to half precision floating-point values section.

   $$g(r) = \begin{cases} f_{preferred}(r) & \text{if rounding mode is round to even} \\ f_{approx}(r) & \text{if implementation-defined rounding mode} \end{cases}$$

   $$x = g(r)$$

The accuracy required of using `image::read` and `image::sample` to convert a normalized 8-bit unsigned integer sRGB color value x to a floating-point linear RGB color value y is given by:

$$|x - 255 \times c_{sRGB}(y)| \leq 0.5$$

The accuracy required of using `image::write` to convert a linear RGB floating-point color value y to a normalized 8-bit unsigned integer sRGB value x is given by:

$$|x - 255 \times c_{sRGB}(y)| \leq 0.6$$

5.4. 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 6. Compiler options

The compiler options are categorized as preprocessor options, options for controlling the OpenCL C++ version, options that control FP16 and FP64 support. This specification defines a standard set of options that must be supported by the compiler when building program executables online or offline from OpenCL C++ to an IL. These may be extended by a set of vendor or platform specific options.

6.1. Preprocessor options

These options control the OpenCL C++ preprocessor which is run on each program source before actual compilation.

```
-D name
```

Predefine name as a macro, with definition 1.

```
-D name=definition
```

The contents of definition are tokenized and processed as if they appeared during translation phase three in a `#define` directive. In particular, the definition will be truncated by embedded newline characters.

6.2. Options Controlling the OpenCL C++ version

The following option controls the version of OpenCL C++ that the compiler accepts.

```
-cl-std=
```

Determine the OpenCL C++ language version to use. A value for this option must be provided. Valid values are:

- c++ - Support all OpenCL C++ programs that use the OpenCL C++ language features defined in the *OpenCL C++ Programming Language* section.

6.3. Double and half-precision floating-point options

The following option controls the double and half floating-point support that the compiler accepts.

```
-cl-fp16-enable
```

This option enables full half data type support. The option defines `cl_khr_fp16` macro. The default is disabled.

```
-cl-fp64-enable
```

This option enables double data type support. The option defines `cl_khr_fp64` macro. The default is disabled.

6.4. Other options

```
-cl-zero-init-local-mem-vars
```

This option enables software zero-initialization of variables allocated in local memory.

[2] The `double` data type is an optional type that is supported if `CL_DEVICE_DOUBLE_FP_CONFIG` in table 4.3 for a device is not zero.

[3] The question mark ? in numerical selector refers to special undefined component of vector; reading from it results in undefined value, writing to it is discarded.

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

[5] 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.

[6] The `as_type<T>` function is intended to reflect the organization of data in register. The `as_type<T>` construct is 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. So, for example if an implementation stores all single precision data as double in register, it should implement `as_type<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.

[7] `memory_order_consume` is not supported in OpenCL C++

[8] This value for `memory_scope` can only be used with `atomic_fence` with flags set to `mem_fence::image`.

[9] We can't require C++14 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 C++14.

[10] 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.

[11] 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.

[12] 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 only if the **cl_khr_int64_base_atomics** and **cl_khr_int64_extended_atomics** extensions are supported and have been enabled.

[13] The `*_ms` types are supported only if the *cl_khr_gl_msaa_sharing* and **cl_khr_gl_depth_images** extensions are supported and have been enabled.

[14] Immediate meaning not side effects resulting from child kernels. The side effects would include stores to global memory and pipe reads and writes.

[15] 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.

[17] i.e. the `global_work_size` values specified to `clEnqueueNDRangeKernel` are not evenly divisible by the `local_work_size` values for each dimension.

[18] Only if double precision is supported and has been enabled.

[19] Refer to the *Memory order and scope* section for description of `memory_scope`.

[20] 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.

[21] fmin and fmax behave as defined by C++14 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.

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

[23] 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.

[24] The primary purpose of the printf function is to help in debugging OpenCL kernels.

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

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

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

[28] 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.

[29] 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.

[30] Except for the embedded profile whether either round to zero or round to nearest rounding mode may be supported for single precision floating-point.

[31] The ULP values for built-in math functions `lgamma` and `lgamma_r` is currently undefined.

[32] 0 ulp is used for math functions that do not require rounding.

[33] 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()`.

[35] 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.