Metal Shading Language Guide

Developer

Contents

Tables

Introduction

The Metal shading language is a unified programming language for writing both graphics and compute kernel functions that are used by apps written with the Metal framework.

The Metal shading language is designed to work together with the Metal framework, which manages the execution, and optionally the compilation, of the Metal shading language code. The Metal shading language uses clang and LLVM so developers get a compiler that delivers close to the metal performance for code executing on the GPU.

At a Glance

This document describes the Metal unified graphics and compute shading language. The Metal shading language is a C++ based programming language that developers can use to write code that is executed on the GPU for graphics and general-purpose data-parallel computations. Since the Metal shading language is based on C++, developers will find it familiar and easy to use. With the Metal shading language, both graphics and compute programs can be written with a single, unified language, which allows tighter integration between the two.

How to Use This Document

Developers who are writing code with the Metal framework will want to read this document, because they will need to use the Metal shading language to write graphics and compute programs to be executed on the GPU. This document is organized into the following chapters:

- Metal and C++11 (page 8) covers the similarities and differences between the Metal shading language and C++11.
- Metal Data Types (page 10) lists the Metal shading language data types, including types that represent vectors, matrices, buffers, textures, and samplers. It also discusses type alignment and type conversion.
- Operators (page 30) lists the Metal shading language operators.
- Functions, Variables, and Qualifiers (page 36) details how functions and variables are declared, sometimes with qualifiers that restrict how they are used.
- Metal Standard Library (page 62) defines a collection of built-in Metal shading language functions.

- Compiler Options (page 93) details the options for the Metal shading language compiler, including pre-processor directives, options for math intrinsics, and options that control optimization.
- Numerical Compliance (page 95) describes requirements for representing floating-point numbers, including accuracy in mathematical operations.

See Also

C++11

Stroustrup, Bjarne. The C++ Programming Language. Harlow: Addison-Wesley, 2013.

Metal Framework

The *Metal Programming Guide* provides a detailed introduction to writing apps with the Metal framework.

The *Metal Framework Reference* details individual classes in the Metal framework.

Metal and C++11

The Metal shading language is based on the *C++11 Specification* (a.k.a., the ISO/IEC JTC1/SC22/WG21 N3290 Language Specification) with specific extensions and restrictions. Please refer to the C++11 Specification for a detailed description of the language grammar.

This section and its subsections describe modifications and restrictions to the C++11 language supported in the Metal shading language.

For more information about Metal shading language pre-processing directives and compiler options, see Compiler Options (page 93) of this document.

Overloading

The Metal shading language supports overloading as defined by section 13 of the *C++11 Specification*. The function overloading rules are extended to include the address space qualifier of an argument. The Metal shading language graphics and kernel functions cannot be overloaded. (For definition of graphics and kernel functions, see Function Qualifiers (page 36).)

Templates

The Metal shading language supports templates as defined by section 14 of the *C++11 Specification*.

Preprocessing Directives

The Metal shading language supports the pre-processing directives defined by section 16 of the *C++11 Specification*.

Restrictions

The following C++11 features are not available in the Metal shading language (section numbers in this list refer to the *C++11 Specification*):

- lambda expressions (section 5.1.2)
- recursive function calls (section 5.2.2, item 9)
- dynamic_cast operator (section 5.2.7)
- type identification (section 5.2.8)
- new and delete operators (sections 5.3.4 and 5.3.5)
- noexcept operator (section 5.3.7)
- goto statement (section 6.6)
- register, thread_local storage qualifiers (section 7.1.1)
- virtual function qualifier (section 7.1.2)
- derived classes (sections 10 and 11)
- exception handling (section 15)

The C++ standard library must not be used in the Metal shading language code. Instead of the C++ standard library, Metal has its own standard library that is described in Metal Standard Library (page 62).

The Metal shading language restricts the use of pointers:

- Arguments to Metal graphics and kernel functions that are pointers declared in a program must be declared with the Metal `device`, `threadgroup`, or `constant` address space qualifier. (See Address Space Qualifiers for Variables and Arguments (page 37) for more about address space qualifiers.)
- Function pointers are not supported.

A Metal function cannot be called `main`.

Metal Pixel Coordinate System

In Metal, the origin of the pixel coordinate system of a texture or a framebuffer attachment is defined at the top left corner.

Metal Data Types

This chapter details the Metal shading language data types, including types that represent vectors and matrices. Atomic data types, buffers, textures, samplers, arrays, and user-defined structs are also discussed. Type alignment and type conversion are also described.

Scalar Data Types

Metal supports the scalar types listed in Table 2-1 (page 10). Metal does **not** support the `double`, `long`, `unsigned long`, `long long`, `unsigned long long`, and `long double` data types.

Table 2-1 Metal scalar data types

Type	Description
`bool`	A conditional data type that has the value of either `true` or `false`. The value `true` expands to the integer constant 1, and the value `false` expands to the integer constant 0.
`char`	A signed two's complement 8-bit integer.

`unsigned char`
`uchar`
	An unsigned 8-bit integer.

`short`	A signed two's complement 16-bit integer.

`unsigned short`
`ushort`
	An unsigned 16-bit integer.

`int`	A signed two's complement 32-bit integer.

`unsigned int`
`uint`
	An unsigned 32-bit integer.

`half`	A 16-bit floating-point. The half data type must conform to the IEEE 754 binary16 storage format.
`float`	A 32-bit floating-point. The float data type must conform to the IEEE 754 single precision storage format.

Type	Description
`size_t`	An unsigned integer type of the result of the `sizeof` operator. This is a 64-bit unsigned integer.
`ptrdiff_t`	A signed integer type that is the result of subtracting two pointers. This is a 64-bit signed integer.
`void`	The `void` type comprises an empty set of values; it is an incomplete type that cannot be completed.

Note: Metal supports the following suffixes that specify the type for a literal:

- the standard `f` or `F` suffix to specify a single precision floating-point literal value (e.g., `0.5f` or `0.5F`).
- the `h` or `H` suffix to specify a half precision floating-point literal value (e.g., `0.5h` or `0.5H`).
- the `u` or `U` suffix for unsigned integer literals.

Vector and Matrix Data Types

The Metal shading language supports a subset of the vector and matrix data types implemented by the system vector math library.

The vector type names supported are:

`booln`

`charn, shortn, intn, ucharn, ushortn, uintn`

`halfn` and `floatn`

`n` is 2, 3, or 4 representing a 2-, 3- or 4- component vector type.

The matrix type names supported are:

`halfnxm` and `floatnxm`

where `n` and `m` are number of columns and rows. `n` and `m` can be 2, 3, or 4. A matrix is composed of several vectors. For example, a `floatnx3` matrix is composed of `n` `float3` vectors. Similarly, a `halfnx4` matrix is composed of `n` `half4` vectors.

Accessing Vector Components

Vector components can be accessed using an array index. Array index 0 refers to the first component of the vector, index 1 to the second component, and so on. The following examples show various ways to access array components:

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

float x = pos[0];    // x = 1.0
float z = pos[2];    // z = 3.0

float4 vA = float4(1.0f, 2.0f, 3.0f, 4.0f);
float4 vB;

for (int i=0; i<4; i++)
    vB[i] = vA[i] * 2.0f // vB = (2.0, 4.0, 6.0, 8.0);
```

Metal supports using the period (`.`) as a selection operator to access vector components, using letters that may indicate coordinate or color data: `<vector_data_type>.xyzw` or `<vector_data_type>.rgba`.

In the following code, the vector test is initialized, and then components are accessed using the `.xyzw` or `.rgba` selection syntax:

```
int4 test = int4(0, 1, 2, 3);
int a = test.x;  //  a = 0
int b = test.y;  //  b = 1
int c = test.z;  //  c = 2
int d = test.w;  //  d = 3
int e = test.r;  //  e = 0
int f = test.g;  //  f = 1
int g = test.b;  //  g = 2
int h = test.a;  //  h = 3
```

The component selection syntax allows multiple components to be selected.

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

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

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

The component group notation can occur on the left hand side of an expression. To form the lvalue, swizzling may be applied. The resulting lvalue may be either the scalar or vector type, depending on number of components specified. Each component must be a supported scalar or vector type. The resulting lvalue of vector type must not contain duplicate components.

```
float4 pos = float4(1.0f, 2.0f, 3.0f, 4.0f);
// pos = (5.0, 2.0, 3.0, 6.0)
pos.xw = float2(5.0f, 6.0f);

// pos = (8.0, 2.0, 3.0, 7.0)
pos.wx = float2(7.0f, 8.0f);

// pos = (3.0, 5.0, 9.0, 7.0)
pos.xyz = float3(3.0f, 5.0f, 9.0f);
```

The following methods of vector component access are not permitted and result in a compile-time error:

- Accessing components beyond those declared for the vector type is an error. 2-component vector data types can only access `.xy` or `.rg` elements. 3-component vector data types can only access `.xyz` or `.rgb` elements. For instance:

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


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

- Accessing the same component twice on the left-hand side is ambiguous; for instance,

  ```
  // illegal - 'x' used twice
  pos.xx = float2(3.0f, 4.0f);


  // illegal - mismatch between float2 and float4
  ```

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

- The `.rgba` and `.xyzw` qualifiers cannot be intermixed in a single access; for instance,

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

pos.x = 1.0f;    // OK
pos.g = 2.0f;    // OK
pos.xg = float2(3.0f, 4.0f); // illegal - mixed qualifiers used
float3 coord = pos.ryz;  // illegal - mixed qualifiers used
```

- A pointer or reference to a vector with swizzles; for instance

```
float4 pos = float4(1.0f, 2.0f, 3.0f, 4.0f);
my_func(&pos.xy);    // illegal
```

The `sizeof` operator on a vector type returns the size of the vector, which is given as the number of components * size of each component. For example, `sizeof(float4)` returns 16 and `sizeof(half4)` returns 8.

Accessing Matrix Components

The `floatnxm` and `halfnxm` matrices can be accessed as an array of n `floatm` or n `halfm` entries.

The components of a matrix can be accessed using the array subscripting syntax. Applying a single subscript to a matrix treats the matrix as an array of column vectors. The first column is column 0. A second subscript would then operate on the resulting vector, as defined earlier for vectors. Hence, two subscripts select a column and then a row.

```
float4x4 m;
// sets the 2nd column to all 2.0
m[1] = float4(2.0f);
// sets the 1st element of the 1st column to 1.0
m[0][0] = 1.0f;
// sets the 4th element of the 3rd column to 3.0
m[2][3] = 3.0f;
```

Accessing a component outside the bounds of a matrix with a non-constant expression results in undefined behavior. Accessing a matrix component that is outside the bounds of the matrix with a constant expression generates a compile-time error.

Vector Constructors

Constructors can be used to create vectors from a set of scalars or vectors. When a vector is initialized, its parameter signature determines how it is constructed. For instance, if the vector is initialized with only a single scalar parameter, all components of the constructed vector are set to that scalar value.

If a vector is constructed from multiple scalars, one or more vectors, or a mixture of these, the vector's components are constructed in order from the components of the arguments. The arguments are consumed from left to right. Each argument has all its components consumed, in order, before any components from the next argument are consumed.

This is a complete list of constructors that are available for `float4`:

```
float4(float x);
float4(float x, float y, float z, float w);
float4(float2 a, float2 b);
float4(float2 a, float b, float c);
float4(float a, float b, float2 c);
float4(float a, float2 b, float c);
float4(float3 a, float b);
float4(float a, float3 b);
float4(float4 x);
```

This is a complete list of constructors that are available for `float3`:

```
float3(float x);

float3(float x, float y, float z);

float3(float a, float2 b);

float3(float2 a, float b);

float3(float3 x);
```

This is a complete list of constructors that are available for `float2`:

```
float2(float x);

float2(float x, float y);

float2(float2 x);
```

The following examples illustrate uses of the constructors:

```
float x = 1.0f, y = 2.0f, z = 3.0f, w = 4.0f;
float4 a = float4(0.0f);
float4 b = float4(x, y, z, w);
float2 c = float2(5.0f, 6.0f);

float2 a = float2(x, y);
float2 b = float2(z, w);
float4 x = float4(a.xy, b.xy);
```

Under-initializing a vector constructor is a compile-time error.

Matrix Constructors

Constructors can be used to create matrices from a set of scalars, vectors or matrices. When a matrix is initialized, its parameter signature determines how it is constructed. For example, if a matrix is initialized with only a single scalar parameter, the result is a matrix that contains that scalar for all components of the matrix's diagonal, with the remaining components initialized to 0.0. For example, a call to

```
float4x4(fval);
```

where `fval` is a scalar floating-point value, constructs a matrix with these initial contents:

```
fval    0.0    0.0    0.0

0.0    fval    0.0    0.0

0.0    0.0    fval    0.0

0.0    0.0    0.0    fval
```

A matrix can also be constructed from another matrix that is of the same size, i.e., has the same number of rows and columns. For example,

```
float3x4(float3x4);

float3x4(half3x4);
```

Matrix components are constructed and consumed in column-major order. The matrix constructor must have just enough values specified in its arguments to initialize every component in the constructed matrix object. Providing more arguments than are needed results in an error. Under-initializing a matrix constructor also results in a compile-time error.

A matrix of type T with n columns and m rows can also be constructed from n vectors of type T with m components. The following examples are legal constructors:

```
float2x2(float2, float2);
float3x3(float3, float3, float3);
float3x2(float2, float2, float2);
```

The following are examples of matrix constructors that are not supported. A matrix cannot be constructed from multiple scalar values, nor from combinations of vectors and scalars.

```
// both cases below are not supported
float2x2(float a00, float a01, float a10, float a11);
float2x3(float2 a, float b, float2 c, float d);
```

Atomic Data Types

The Metal atomic data type is restricted for use by atomic functions implemented by the Metal shading language, as described in Atomic Functions (page 89). These atomic functions are a subset of the C++11 atomic and synchronization functions. Metal atomic functions must operate on Metal atomic data.

The Metal atomic types are defined as: `atomic_int` and `atomic_uint`.

Buffers

Metal implements buffers as a pointer to a built-in or user defined data type described in the `device` or `constant` address space. (Refer to Address Space Qualifiers for Variables and Arguments (page 37) for a full description of these address qualifiers.) These buffers can be declared in program scope or passed as arguments to a function.

Example:

```
device float4    *device_buffer;
struct my_user_data {
    float4 a;
    float  b;
    int2   c;
};
constant my_user_data *user_data;
```

Textures

The texture data type is a handle to one-, two-, or three-dimensional texture data that corresponds to all or a portion of a single mipmap level of a texture. The following templates define specific texture data types:

```
enum class access { sample, read, write };

texture1d<T, access a = access::sample>
texture1d_array<T, access a = access::sample>
texture2d<T, access a = access::sample>
texture2d_array<T, access a = access::sample>
texture3d<T, access a = access::sample>
texturecube<T, access a = access::sample>
texture2d_ms<T, access a = access::read>
```

Textures with depth formats must be declared as one of the following texture data types:

```
enum class depth_format { depth_float };
depth2d<T, access a = access::sample,
           depth_format d = depth_format::depth_float>
depth2d_array<T, access a = access::sample,
                 depth_format d = depth_format::depth_float>
depthcube<T, access a = access::sample,
             depth_format d = depth_format::depth_float>
depth2d_ms<T, access a = access::read,
              depth_format d = depth_format::depth_float>
```

`T` specifies the color type returned when reading from a texture or the color type specified when writing to the texture. For texture types (except depth texture types), `T` can be `half`, `float`, `short`, `ushort`, `int`, or `uint`. For depth texture types, `T` must be `float`.

Note: If T is `int` or `short`, the data associated with the texture must use a signed integer format. If T is `uint` or `ushort`, the data associated with the texture must use an unsigned integer format. If T is `half`, the data associated with the texture must either be a normalized (signed or unsigned integer) or half precision format. If T is `float`, the data associated with the texture must either be a normalized (signed or unsigned integer), half or single precision format.

The `access` qualifier describes how the texture can be accessed. The supported access qualifiers are:

- `sample` - The texture object can be sampled. `sample` implies the ability to read from a texture with and without a sampler.
- `read` - Without a sampler, a graphics, or kernel function can only read the texture object.
- `write` – A graphics or kernel function can write to the texture object.

The `depth_format` qualifier describes the depth texture format. The only supported value is `depth_format`.

Note: For multisampled textures, only the `read` qualifier is supported.

For cube textures, only the `sample` and `read` qualifiers are supported.

For depth textures, only the `sample` and `read` qualifiers are supported.

A pointer or a reference to a texture type is not supported and will result in a compilation error.

The following example uses these access qualifiers with texture object arguments.

```
void foo (texture2d<float> imgA [[ texture(0) ]],
    texture2d<float, access::read> imgB [[ texture(1) ]],
    texture2d<float, access::write> imgC [[ texture(2) ]])
{
    ...
}
```

(See Attribute Qualifiers to Locate Resources (page 40) for description of the `texture` attribute qualifier.)

Samplers

In the Metal shading language, the `sampler` type identifies how to sample a texture. The Metal framework allows you to create a corresponding `MTLSamplerState` object and pass it in an argument to a graphics or kernel function. A `sampler` object can also be described in Metal shading language program source instead of in the Metal framework. For these cases we only allow a subset of the sampler state to be specified: the addressing mode, filter mode, normalized coordinates, and comparison function.

Table 2-2 (page 20) describes the list of supported sampler state enums and their associated values (and defaults). These states can be specified when a sampler is initialized in Metal shading language program source.

Table 2-2 Sampler State Enumeration Values

Enum Name	Valid Values	Description
`coord`	`normalized` (default) `pixel`	Specifies whether the texture coordinates when sampling from a texture are normalized or unnormalized values.

`address`	`clamp_to_edge` (default)
	`clamp_to_zero`
	`mirrored_repeat`
	`repeat`
		Sets the addressing mode for all texture coordinates.

`s_address`
`t_address`
`r_address`
	`clamp_to_edge` (default)
	`clamp_to_zero`
	`mirrored_repeat`
	`repeat`
		Sets the addressing mode for an individual texture coordinate.

`filter`	`nearest` (default) `linear`	Sets the magnification and minification filtering modes for texture sampling.
`mag_filter`	`nearest` (default) `linear`	Sets the magnification filtering mode for texture sampling.
`min_filter`	`nearest` (default) `linear`	Sets the minification filtering modes for texture sampling.

`mip_filter`	`none` (default)
	`nearest`
	`linear`
		Sets the mipmap filtering mode for texture sampling. If `none`, then only one level-of-detail is active

Enum Name	Valid Values	Description

`compare_func`	`none` (default)
	`les`
	`less_equal`
	`greater`
	`greater_equal`
	`equal`
	`not_equal`
		Sets comparison test to use with r texture coordinate for shadow maps.
		The `compare_func` can only be specified for samplers declared in Metal shading language source.

For the addressing mode, `clamp_to_zero` is similar to the OpenGL clamp to border addressing mode except that the border color value is always (0.0, 0.0, 0.0, 1.0) when sampling outside a texture that does not have an alpha component or is (0.0, 0.0, 0.0, 0.0) when sampling outside the texture that has an alpha component.

The enumeration types used by the sampler data type as described in Table 2-2 (page 20) are specified as follows. (If `coord` is set to `pixel`, the `min_filter` and `mag_filter` values must be the same, the `mip_filter` and `compare_func` values must be `none`, and the address modes must be either `clamp_to_zero` or `clamp_to_edge`.)

```
enum class coord { normalized, pixel };
enum class filter { nearest, linear };
enum class min_filter { nearest, linear };
enum class mag_filter { nearest, linear };
enum class s_address { clamp_to_zero, clamp_to_edge,
                       repeat, mirrored_repeat };
enum class t_address { clamp_to_zero, clamp_to_edge,
                       repeat, mirrored_repeat };
enum class r_address { clamp_to_zero, clamp_to_edge,
                       repeat, mirrored_repeat };
enum class address { clamp_to_zero, clamp_to_edge,
                     repeat, mirrored_repeat };
enum class mip_filter { none, nearest, linear };

// can only be used with depth_sampler
enum class compare_func { none, less, less_equal, greater,
                  greater_equal, equal, not_equal };
```

The Metal shading language implements a sampler object as follows:

```
struct sampler {

    public:

        // full version of sampler constructor
        template<typename... Ts>
        constexpr sampler(Ts... sampler_params){};
    private:
};
```

`Ts` must be the enumeration types listed above that can be used by the sampler data type. If the same enumeration type is declared multiple times in a given sampler constructor, the last listed value will take effect.

The following Metal program source illustrates several ways to declare samplers. (The attribute qualifiers (`sampler(n)`, `buffer(n)`, and `texture(n)`) that appear in the code below are explained in Attribute Qualifiers to Locate Resources (page 40).). Note that samplers or constant buffers declared in program source do not need these attribute qualifers.

```
constexpr sampler s(coord::pixel,
                address::clamp_to_zero,
                filter::linear);

constexpr sampler a(coord::normalized);

constexpr sampler b(address::repeat);

constexpr sampler s(address::clamp_to_zero,
                    filter::linear,
                    compare_func::less);

kernel void
my_kernel(device float4 *p [[ buffer(0) ]],
          texture2d<float4> img [[ texture(0) ]],
          sampler smp [[ sampler(3) ]],
          ...)
{
    ...
}
```

Note: Samplers that are initialized in the Metal program source must be declared with the `constexpr` qualifier.

A pointer or a reference to a sampler type is not supported and will result in a compilation error.

Arrays and Structs

Arrays and structs are supported with the following restrictions:

- Arrays of texture and sampler types are not supported.
- The `texture` and `sampler` types cannot be declared in a struct.
- Arguments to graphics and kernel functions cannot be declared to be of type `size_t`, `ptrdiff_t`, or a struct and/or union that contain members declared to be one of these built-in scalar types.
- Members of a struct must belong to the same address space.

Alignment and Size of Types

Table 2-3 (page 23) lists the alignment and size of the scalar and vector data types.

Table 2-3 Alignment and Size of Scalar and Vector Data Types

Type	Alignment (in bytes)	Size (in bytes)
`bool`	1	1
`char` `uchar`	1	1
`char2` `uchar2`	2	2
`char3` `uchar3`	4	4
`char4` `uchar4`	4	4
`short` `ushort`	2	2
`short2` `ushort2`	4	4
`short3` `ushort3`	8	8
`short4` `ushort4`	8	8

Type	Alignment (in bytes)	Size (in bytes)
`int` `uint`	4	4
`int2` `uint2`	8	8
`int3` `uint3`	16	16
`int4` `uint4`	16	16
`half`	2	2
`half2`	4	4
`half3`	8	8
`half4`	8	8
`float`	4	4
`float2`	8	8
`float3`	16	16
`float4`	16	16

Table 2-4 (page 24) lists the alignment and size of the matrix data types.

Table 2-4 Alignment and Size of Matrix Data Types

Type	Alignment (in bytes)	Size (in bytes)
`half2x2`	4	8
`half2x3`	8	16
`half2x4`	8	16
`half3x2`	4	12
`half3x3`	8	24

Type	Alignment (in bytes)	Size (in bytes)
`half3x4`	8	24
`half4x2`	4	16
`half4x3`	8	32
`half4x4`	8	32
`float2x2`	8	16
`float2x3`	16	32
`float2x4`	16	32
`float3x2`	8	24
`float3x3`	16	48
`float3x4`	16	48
`float4x2`	8	32
`float4x3`	16	64
`float4x4`	16	64

Since a matrix is composed of vectors, each column of a matrix has the alignment of its vector component. For example, each column of a `floatnx3` matrix is a `float3` vector that is aligned on a 16-byte boundary, as shown in Table 2-3 (page 23). Similarly, each column of a `halfnx2` matrix is a `half2` vector that is aligned on a 4-byte boundary.

The `alignas` alignment specifier can be used to specify the alignment requirement of a type or an object. The `alignas` specifier may be applied to the declaration of a variable or a data member of a struct or class. It may also be applied to the declaration of a struct, class or enumeration type.

The Metal shading language compiler is responsible for aligning data items to the appropriate alignment as required by the data type. For arguments to a graphics or kernel function declared to be a pointer to a data type, the Metal shading language compiler can assume that the pointee is always appropriately aligned as required by the data type.

Packed Vector Data Types

The vector data types described in Vector and Matrix Data Types (page 11) are aligned to the size of the vector. There are a number of use cases where developers require their vector data to be tightly packed. For example – a vertex struct that may contain position, normal, tangent vectors and texture coordinates tightly packed and passed as a buffer to a vertex function.

The packed vector type names supported are:

> `packed_charn`, `packed_shortn`, `packed_intn`,
>
> `packed_ucharn`, `packed_ushortn`, `packed_uintn`,
>
> `packed_halfn`, and `packed_floatn`

where `n` is 2, 3, or 4 representing a 2-, 3- or 4- component vector type. (The `packed_booln` vector type names are reserved.)

Table 2-5 (page 26) lists the alignment and size of the packed vector data types.

Table 2-5 Alignment and Size of Packed Vector Data Types

Packed Vector Type	Alignment (in bytes)	sizeof (in bytes)
`packed_char2` `packed_uchar2`	1	2
`packed_char3` `packed_uchar3`	1	3
`packed_char4` `packed_uchar4`	1	4
`packed_short2` `packed_ushort2`	2	4
`packed_short3` `packed_ushort3`	2	6
`packed_short4` `packed_ushort4`	2	8
`packed_int2` `packed_uint2`	4	8

Packed Vector Type	**Alignment (in bytes)**	**sizeof (in bytes)**
`packed_int3` `packed_uint3`	4	12
`packed_int4` `packed_uint4`	4	16
`packed_half2`	2	4
`packed_half3`	2	6
`packed_half4`	2	8
`packed_float2`	4	8
`packed_float3`	4	12
`packed_float4`	4	16

Packed vector data types are typically used as a data storage format. Loads and stores from a packed vector data type to an aligned vector data type and vice-versa, copy constructor and assignment operator are supported. The arithmetic, logical, and relational operators are also supported for packed vector data types.

Example:

```
device float4 *buffer;
device packed_float4 *packed_buffer;

int i;
packed_float4 f ( buffer[i] );
pack_buffer[i] = buffer[i];

// operator to convert from packed_float4 to float4.
buffer[i] = float4( packed_buffer[i] );
```

Components of a packed vector data type can be accessed with an array index. However, components of a packed vector data type cannot be accessed with the `.xyzw` or `.rgba` selection syntax.

Example:

```
packed_float4 f;
f[0] = 1.0f;  // OK
f.x = 1.0f;   // Illegal - compilation error
```

Implicit Type Conversions

Implicit conversions between scalar built-in types (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 is converted to the floating-point value 5.0.

All vector types are considered to have a higher conversion rank than scalar types. Implicit conversions from a vector type to another vector or scalar type are not permitted and a compilation error results. For example, the following attempt to convert from a 4-component integer vector to a 4-component floating-point vector fails.

```
int4 i; float4 f = i;    // compile error
```

Implicit conversions from scalar-to-vector types are supported. The scalar value is replicated in each element of the vector. The scalar may also be subject to the usual arithmetic conversion to the element type used by the vector or matrix.

For example:

```
float4 f = 2.0f;    // f = (2.0f, 2.0f, 2.0f, 2.0f)
```

Implicit conversions from scalar-to-matrix types and vector-to-matrix types are not supported and a compilation error results. Implicit conversions from a matrix type to another matrix, vector or scalar type are not permitted and a compilation error results.

Implicit conversions for pointer types follow the rules described in the *C++11 Specification*.

Type Conversions and Re-interpreting Data

The `static_cast` operator is used to convert from a scalar or vector type to another scalar or vector type with no saturation and with a default rounding mode (i.e., when converting to floating-point, round to zero or round to the nearest even number depending on the rounding mode supported by the GPU; when converting to integer, round toward zero). If the source type is a scalar or vector boolean, the value `false` is converted to zero and the value `true` is converted to one.

The Metal shading language adds an `as_type<type-id>` operator to allow any scalar or vector data type (that is not a pointer) to be reinterpreted as another scalar or vector data type of the same size. The bits in the operand are returned directly without modification as the new type. The usual type promotion for function arguments is not performed.

For example, `as_type<float>(0x3f800000)` returns 1.0f, which is the value of the bit pattern `0x3f800000` if viewed as an IEEE-754 single precision value.

It is an error to use the `as_type<type-id>` operator to reinterpret data to a type of a different number of bytes.

Examples:

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

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

int i;
// Legal.
short2 j = as_type<short2>(i);

half4 f;
// Error. Result and operand have different sizes
float4 g = as_type<float4>(f);

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

Operators

This chapter lists and describes the Metal shading language operators.

Scalar and Vector Operators

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

   - The two operands are scalars. In this case, the operation is applied, and the result is a scalar.
   - One operand is a scalar, and the other is a vector. In this case, the scalar is converted to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is performed component-wise, which results in a same size vector.
   - The two operands are vectors of the same size. In this case, the operation is performed component-wise, which results in a same size vector.

   Division on integer types that results in a value that lies outside of the range bounded by the maximum and minimum representable values of the integer type, such as TYPE_MIN/-1 for signed integer types, or division by zero does not cause an exception but results in an unspecified value. Division by zero for floating-point types results in ±infinity or NaN, as prescribed by the IEEE-754 standard. (For details about numerical accuracy of floating-point operations, see Numerical Compliance (page 95).)

2. The operator modulus (%) operates on scalar and vector integer data types. All arithmetic operators return a result of the same built-in type as the type of the operands, after operand type conversion. The following cases are valid:

   - The two operands are scalars. In this case, the operation is applied, and the result is a scalar.
   - One operand is a scalar, and the other is a vector. In this case, the scalar is converted to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is performed component-wise, which results in a same size vector.
   - The two operands are vectors of the same size. In this case, the operation is performed component-wise, which results in a same size vector.

The resulting value is undefined for any component computed with a second operand that is zero, while results for other components with non-zero operands remain defined. If both operands are non-negative, the remainder is non-negative. If one or both operands are negative, results are undefined.

3. The arithmetic unary operators (+ and -) operate on scalar and vector, integer and floating-point types.

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

5. The relational operators greater than (>), less than (<), greater than or equal (>=), and less than or equal (<=) operate on scalar and vector, integer and floating-point types. The result is a Boolean (`bool` type) scalar or vector. After operand type conversion, the following cases are valid:

   - The two operands are scalars. In this case, the operation is applied, resulting in a `bool`.
   - One operand is a scalar, and the other is a vector. In this case, the scalar is converted to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is performed component-wise, which results in a Boolean vector.
   - The two operands are vectors of the same type. In this case, the operation is performed component-wise, which results in a Boolean vector.

   The relational operators always return `false` if either argument is a NaN.

   For vector operands, to test whether any or all elements in the result of a vector relational operator test true, use the `any` or `all` built-in functions defined in Relational Functions (page 65).

6. The equality operators, equal (==) and not equal (!=), operate on scalar and vector, integer and floating-point types. All equality operators result in a Boolean (`bool` type) scalar or vector. After operand type conversion, the following cases are valid:

   - The two operands are scalars. In this case, the operation is applied, resulting in a `bool`.
   - One operand is a scalar, and the other is a vector. In this case, the scalar is converted to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is performed component-wise, resulting in a Boolean vector.
   - The two operands are vectors of the same type. In this case, the operation is performed component-wise resulting in a same size Boolean vector.

All other cases of implicit conversions are illegal. If one or both arguments is "Not a Number" (NaN), the equality operator equal (==) returns `false`. If one or both arguments is "Not a Number" (NaN), the equality operator not equal (!=) returns `true`.

7. The bitwise operators and (&), or (|), exclusive or (^), not (~) operate on all scalar and vector built-in types except the built-in scalar and vector floating-point types. For built-in vector types, the operators are applied component-wise. If one operand is a scalar and the other is a vector, the scalar is converted to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is performed component-wise resulting in a same size vector.
8. The logical operators and (&&), or (||) operate on two Boolean expressions. The result is a scalar or vector Boolean.
9. The logical unary operator not (!) operates on a Boolean expression. The result is a scalar or vector Boolean.
10. The ternary selection operator (?:) operates on three expressions (*exp1* ? *exp2* : *exp3*). This operator evaluates the first expression *exp1*, which must result in a scalar Boolean. If the result is `true`, it selects to evaluate the second expression; otherwise it evaluates the third expression. Only one of the second and third expressions is evaluated. The second and third expressions can be any type, as long their types match, or there is a conversion in Implicit Type Conversions (page 28) 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 in which case the scalar is widened to the same type as the vector type. This resulting matching type is the type of the entire expression.
11. The ones' complement operator (~). The operand must be of a scalar or vector integer type, and the result is the ones' complement of its operand.

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

    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`, 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`, 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.

12. The assignment operator behaves as described by the C++11 Specification. For the `lvalue = expression` assignment operation, if `expression` is a scalar type and `lvalue` is a vector type, the scalar is converted to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is performed component-wise, which results in a same size vector.

Note: Operators not described above that are supported by C++11 (such as `sizeof(T)`, unary (&) operator, and comma (`,`) operator) behave as described in the *C++11 Specification*.

Unsigned integers shall obey the laws of arithmetic modulo 2^n, where n is the number of bits in the value representation of that particular size of integer. The result of signed integer overflow is undefined.

For integral operands the divide (`/`) operator yields the algebraic quotient with any fractional part discarded. (This is often called truncation towards zero.) If the quotient `a/b` is representable in the type of the result, `(a/b)*b + a%b` is equal to a.

Matrix Operators

The arithmetic operators add (+), subtract (-) operate on matrices. Both matrices must have the same numbers of rows and columns. The operation is done component-wise resulting in the same size matrix. The arithmetic operator multiply (*), operates on:

- a scalar and a matrix,
- a matrix and a scalar,
- a vector and a matrix,
- a matrix and a vector,
- or a matrix and a matrix.

If one operand is a scalar, the scalar value is multiplied to each component of the matrix resulting in the same size matrix. A right vector operand is treated as a column vector and a left vector operand as a row vector. For vector – matrix, matrix – vector and matrix – matrix multiplication, the number of columns of the left operand is required to be equal to the number of rows of the right operand. The multiply operation does a linear algebraic multiply, yielding a vector or a matrix that has the same number of rows as the left operand and the same number of columns as the right operand.

The examples below presume these vector, matrix, and scalar variables are initialized:

```
float3 v;
```

```
float3x3 m;
float a = 3.0f;
```

The following matrix-to-scalar multiplication

```
float3x3 m1 = m * a;
```

is equivalent to:

```
m1[0][0] = m[0][0] * a;
m1[0][1] = m[0][1] * a;
m1[0][2] = m[0][2] * a;
m1[1][0] = m[1][0] * a;
m1[1][1] = m[1][1] * a;
m1[1][2] = m[1][2] * a;
m1[2][0] = m[2][0] * a;
m1[2][1] = m[2][1] * a;
m1[2][2] = m[2][2] * a;
```

The following vector-to-matrix multiplication

```
float3 u = v * m;
```

is equivalent to:

```
u.x = dot(v, m[0]);
u.y = dot(v, m[1]);
u.z = dot(v, m[2]);
```

The following matrix-to-vector multiplication

```
float3 u = m * v;
```

is equivalent to:

```
u = v.x * m[0];
u += v.y * m[1];
u += v.z * m[2];
```

The following matrix-to-matrix multiplication

```
float3x3    m, n, r;
r = m * n;
```

is equivalent to:

```
r[0] = m[0] * n[0].x;
r[0] += m[1] * n[0].y;
r[0] += m[2] * n[0].z;

r[1] = m[0] * n[1].x;
r[1] += m[1] * n[1].y;
r[1] += m[2] * n[1].z;

r[2] = m[0] * n[2].x;
r[2] += m[1] * n[2].y;
r[2] += m[2] * n[2].z;
```

Note: The order of partial sums for the vector-to-matrix, matrix-to-vector and matrix-to-matrix multiplication operations described above is undefined.

Functions, Variables, and Qualifiers

This chapter describes how functions, arguments, and variables are declared. It also details how qualifiers are often used with functions, arguments, and variables to specify restrictions.

Function Qualifiers

The Metal shading language supports the following qualifiers that restrict how a function may be used:

- `kernel` - A data-parallel function that is executed over a 1-, 2- or 3-dimensional grid.
- `vertex` - A vertex function that is executed for each vertex in the vertex stream and generates per-vertex output.
- `fragment` – A fragment function that is executed for each fragment in the fragment stream and their associated data and generates per-fragment output.

A function qualifier is used at the start of a function, before its return type. The following example shows the syntax for a compute function.

```
kernel void
foo(...)
{
    ...
}
```

For functions declared with the `kernel` qualifier, the return type must be `void`.

Only a graphics function can be declared with one of the `vertex` or `fragment` qualifiers. For graphics functions, the return type identifies whether the output generated by the function is either per-vertex or per-fragment. The return type for a graphics function may be `void` indicating that the function does not generate output.

Functions that use a `kernel`, `vertex` or `fragment` function qualifier cannot call functions that also use these qualifiers, or a compilation error results.

Address Space Qualifiers for Variables and Arguments

The Metal shading language implements address space qualifiers to specify the region of memory where a function variable or argument is allocated. These qualifiers describe disjoint address spaces for variables:

- `device` (for more details, see device Address Space (page 37))
- `threadgroup` (see threadgroup Address Space (page 38))
- `constant` (see constant Address Space (page 39))
- `thread` (see thread Address Space (page 39))

All arguments to a graphics (vertex or fragment) or compute function that are a pointer or reference to a type must be declared with an address space qualifier. For graphics functions, an argument that is a pointer or reference to a type must be declared in the `device` or `constant` address space. For kernel functions, an argument that is a pointer or reference to a type must be declared in the `device`, `threadgroup`, or `constant` address space. The following example introduces the use of several address space qualifiers. (The `threadgroup` qualifier is supported here for the pointer `l_data` only if `foo` is called by a kernel function, as detailed in threadgroup Address Space (page 38))

```
void foo(device int *g_data,
         threadgroup int *l_data,
         constant float *c_data)
{
    ...
}
```

The address space for a variable at program scope must be `constant`.

Any variable that is a pointer or reference must be declared with one of the address space qualifiers discussed in this section. If an address space qualifier is missing on a pointer or reference type declaration, a compilation error occurs.

device Address Space

The `device` address space name refers to buffer memory objects allocated from the device memory pool that are both readable and writeable.

A buffer memory object can be declared as a pointer or reference to a scalar, vector or user-defined struct. The actual size of the buffer memory object is determined when the memory object is allocated via appropriate Metal framework API calls in the host code.

Some examples are:

```
// an array of a float vector with 4 components
device float4 *color;

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

// an array of Foo elements
device Foo *my_info;
```

Since texture objects are always allocated from the device address space, the `device` address qualifier is not needed for texture types. The elements of a texture object cannot be directly accessed. Functions to read from and write to a texture object are provided.

threadgroup Address Space

The `threadgroup` address space name is used to allocate variables used by a kernel function that are shared by all theads of a threadgroup. Variables declared in the `threadgroup` address space *cannot* be used in graphics functions.

Variables allocated in the `threadgroup` address space in a kernel function are allocated for each threadgroup executing the kernel and exist only for the lifetime of the threadgroup that is executing the kernel.

The example below shows how variables allocated in the `threadgroup` address space can be passed either as arguments or be declared inside a kernel function. (The qualifier `[[ threadgroup(0) ]]` in the code below is explained in Attribute Qualifiers to Locate Resources (page 40).)

```
kernel void
my_func(threadgroup float *a [[ threadgroup(0) ]], ...)
{
    // A float allocated in threadgroup address space
    threadgroup float x;

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

constant Address Space

The `constant` address space name refers to buffer memory objects allocated from the device memory pool but are read-only. Variables in program scope must be declared in the `constant` address space and initialized during the declaration statement. The values used to initialize them must be a compile-time constant. Variables in program scope have the same lifetime as the program, and their values persist between calls to any of the compute or graphics functions in the program.

```
constant float samples[] = { 1.0f, 2.0f, 3.0f, 4.0f };
```

Pointers or references to the `constant` address space are allowed as arguments to functions.

Writing to variables declared in the `constant` address space is a compile-time error. Declaring such a variable without initialization is also a compile-time error.

Note: To decide which address space (`device` or `constant`), a read-only buffer passed to a graphics or kernel function should use, look at how the buffer is accessed inside the graphics or kernel function. The `constant` address space is optimized for multiple instances executing a graphics or kernel function accessing the same location in the buffer. Some examples of this access pattern are accessing light or material properties for lighting / shading, matrix of a matrix array used for skinning, filter weight accessed from a filter weight array for convolution. If multiple executing instances of a graphics or kernel function are accessing the buffer using an index such as the vertex ID, fragment coordinate, or the thread position in grid, then the buffer should be allocated in the `device` address space.

thread Address Space

The `thread` address space refers to the per-thread memory address space. Variables allocated in this address space are not visible to other threads. Variables declared inside a graphics or kernel function are allocated in the `thread` address space.

```
kernel void my_func(...)
{
    // A float allocated in the per-thread address space
    float x;
    // A pointer to variable x in per-thread address space
    thread float p = &x;
    ...
}
```

Function Arguments and Variables

All inputs and outputs to graphics or kernel functions are passed as arguments (except for initialized variables in the `constant` address space and samplers declared in program scope). Arguments to graphics and kernel functions can be one of the following:

- device buffer – a pointer or reference to any data type in the `device` address space (see Buffers (page 17))
- constant buffer – a pointer or reference to any data type in the `constant` address space (see Buffers (page 17))
- `texture` object (see Textures (page 18))
- `sampler` object (see Samplers (page 20))
- a buffer shared between threads in a threadgroup – a pointer to a type in the `threadgroup` address space. (This buffer can only be used as an argument with kernel functions.)

Buffers (device and constant) specified as argument values to a graphics or kernel function cannot alias; i.e., a buffer passed as an argument value cannot overlap another buffer passed to a separate argument of the same graphics or kernel function.

The arguments to these functions are often specified with attribute qualifiers to provide further guidance on their use. Attribute qualifiers are used to specify:

- the resource location for the argument (see Attribute Qualifiers to Locate Resources (page 40)),
- built-in variables that support communicating data between fixed-function and programmable pipeline stages (see Attribute Qualifiers to Locate Per-Vertex Inputs (page 42)),
- which data is sent down the pipeline from vertex function to fragment function (see stage_in Qualifier (page 50)).

Attribute Qualifiers to Locate Buffers, Textures, and Samplers

For each argument, an attribute qualifier must be specified to identify the location of a buffer, texture, or sampler to use for this argument type. The Metal framework API uses this attribute to identify the location for these argument types.

- device and constant buffers – `[[ buffer(index) ]]`
- texture – `[[ texture (index) ]]`
- sampler – `[[ sampler (index) ]]`
- threadgroup buffer – `[[ threadgroup (index) ]]`

The `index` value is an unsigned integer that identifies the location of a buffer, texture, or sampler argument that is being assigned. The proper syntax is for the attribute qualifier to follow the argument/variable name.

The example below is a simple kernel function, `add_vectors`, that adds an array of two buffers in the device address space, `inA` and `inB`, and returns the result in the buffer `out`. The attribute qualifiers (`buffer(index)`) specify the buffer locations for the function arguments.

```
kernel void
add_vectors(const device float4 *inA [[ buffer(0) ]],
            const device float4 *inB [[ buffer(1) ]],
            device float4 *out [[ buffer(2) ]],
            uint id [[ thread_position_in_grid ]])
{
    out[id] = inA[id] + inB[id];
}
```

The example below shows attribute qualifiers used for function arguments of several different types (a buffer, a texture, and a sampler):

```
kernel void

my_kernel(device float4 *p [[ buffer(0) ]],

          texture2d<float> img [[ texture(0) ]],

          sampler sam [[ sampler(1) ]])

{

    ...

}
```

Vertex function example that specifies resources and outputs to device memory

The following example is a vertex function, `render_vertex`, which outputs to device memory in the array `xform_pos_output`, which is a function argument specified with the `device` qualifier (introduced in Function Arguments and Variables (page 40)). All the `render_vertex` function arguments are specified with qualifiers (`buffer(0)`, `buffer(1)`, `buffer(2)`, and `buffer(3)`), as introduced in Attribute Qualifiers to Locate Resources (page 40). (The `position` qualifier shown in this example is discussed in Attribute Qualifiers to Locate Per-Vertex Inputs (page 42).)

```
#include <metal_stdlib>
using namespace metal;

struct VertexOutput {
    float4 position [[position]];
    float4 color;
    float2 texcoord;
};

struct VertexInput {
    float4 position;
    float3 normal;
    float2 texcoord;
};

constexpr constant uint MAX_LIGHTS = 4;

struct LightDesc {
    uint   num_lights;
    float4 light_position[MAX_LIGHTS];
    float4 light_color[MAX_LIGHTS];
    float4 light_attenuation_factors[MAX_LIGHTS];
};

vertex VertexOutput
render_vertex(const device VertexInput* v_in [[ buffer(0) ]],
              constant float4x4& mvp_matrix [[ buffer(1) ]],
              constant LightDesc& light_desc [[ buffer(2) ]],
              device float4* xform_pos_output [[ buffer(3) ]],
              uint v_id [[ vertex_id ]] )
{
    VertexOutput v_out;
    v_out.position = v_in[v_id].position * mvp_matrix;
    v_out.color = do_lighting(v_in[v_id].position,
                    v_in[v_id].normal,
                    light_desc);

    v_out.texcoord = v_in[v_id].texcoord;

    // output position to a buffer
    xform_pos_output[v_id] = v_out.position;

    return v_out;
}
```

Attribute Qualifiers to Locate Per-Vertex Inputs

A vertex function can read per-vertex inputs by indexing into a buffer(s) passed as arguments to the vertex function using the vertex and instance IDs. In addition, per-vertex inputs can also be passed as an argument to a vertex function by declaring them with the `[[stage_in]]` attribute qualifier. For per-vertex inputs passed as an argument declared with the `stage_in` qualifier, each element of the per-vertex input must specify the vertex attribute location as

```
[[ attribute(index) ]]
```

The index value is an unsigned integer that identifies the vertex input location that is being assigned. The proper syntax is for the attribute qualifier to follow the argument/variable name. The Metal framework API uses this attribute to identify the location of the vertex buffer and describe the vertex data such as the buffer to fetch the per-vertex data from, its data format, and stride.

The example below shows how vertex attributes can be assigned to elements of a vertex input struct passed to a vertex function using the `stage_in` qualifier.

```
#include <metal_stdlib>
using namespace metal;

struct VertexInput {
    float4 position [[ attribute(0) ]];
    float3 normal   [[ attribute(1) ]];
    half4 color     [[ attribute(2) ]];
    half2 texcoord  [[ attribute(3) ]];
};

constexpr constant uint MAX_LIGHTS = 4;

struct LightDesc {
    uint   num_lights;
    float4 light_position[MAX_LIGHTS];
    float4 light_color[MAX_LIGHTS];
    float4 light_attenuation_factors[MAX_LIGHTS];
};

constexpr sampler s = sampler(coord::normalized,
                              address::clamp_to_zero,
                              filter::linear);

vertex VertexOutput
render_vertex(VertexInput v_in [[ stage_in ]],
              constant float4x4& mvp_matrix [[ buffer(1) ]],
              constant LightDesc& lights [[ buffer(2) ]],
              uint v_id [[ vertex_id ]])
{
    VertexOutput v_out;
    ...
    return v_out;
}
```

The example below shows how both buffers and the `stage_in` qualifier can be used to fetch per-vertex inputs in a vertex function.

```
#include <metal_stdlib>
using namespace metal;
```

```
struct VertexInput {
    float4 position [[ attribute(0) ]];
    float3 normal   [[ attribute(1) ]];
};

struct VertexInput2 {
    half4 color;
    half2 texcoord[4];
};

constexpr constant uint MAX_LIGHTS = 4;

struct LightDesc {
    uint   num_lights;
    float4 light_position[MAX_LIGHTS];
    float4 light_color[MAX_LIGHTS];
    float4 light_attenuation_factors[MAX_LIGHTS];
};

constexpr sampler s = sampler(coord::normalized,
                              address::clamp_to_zero,
                              filter::linear);

vertex VertexOutput
render_vertex(VertexInput v_in [[ stage_in ]],
              VertexInput2 v_in2 [[ buffer(0) ]],
              constant float4x4& mvp_matrix [[ buffer(1) ]],
              constant LightDesc& lights [[ buffer(2) ]],
              uint v_id [[ vertex_id ]])
{
    VertexOutput vOut;
    ...
    return vOut;
}
```

Attribute Qualifiers for Built-in Variables

Some graphics operations occur in the fixed-function pipeline stages and need to provide values to or receive values from graphics functions. Built-in input and output variables are used to communicate values between the graphics (vertex and fragment) functions and the fixed-function graphics pipeline stages. Attribute qualifiers are used with arguments and the return type of graphics functions to identify these built-in variables.

Attribute Qualifiers for Vertex Function Input

Table 4-1 (page 45) lists the built-in attribute qualifiers that can be specified for arguments to a vertex function and the corresponding data types with which they can be used.

Table 4-1 Attribute Qualifiers for Vertex Function Input Arguments

Attribute Qualifier	Corresponding Data Types
`[[vertex_id]]`	`ushort` or `uint`
`[[instance_id]]`	`ushort` or `uint`

Attribute Qualifiers for Vertex Function Output

Table 4-2 (page 45) lists the built-in attribute qualifiers that can be specified for a return type of a vertex function or the members of a struct that are returned by a vertex function (and the corresponding data types with which they can be used).

Table 4-2 Attribute Qualifiers for Vertex Function Return Types

Attribute Qualifier	Corresponding Data Types
`[[clip_distance]]`	`float` or `float[n]` `n` must be known at compile time
`[[point_size]]`	`float`
`[[position]]`	`float4`

The example below describes a vertex function called `process_vertex`. The function returns a user-defined struct called `VertexOutput`, which contains a built-in variable that represents the vertex position, so it requires the `[[position]]` qualifier.

```
struct VertexOutput {
     float4 position [[position]];
     float4 color;
     float2 texcoord;
};

vertex VertexOutput
process_vertex(...)
{
    VertexOutput v_out;

    // compute per-vertex output
    ...

    return v_out;
}
```

Attribute Qualifiers for Fragment Function Input

Table 4-3 (page 46) lists the built-in attribute qualifiers that can be specified for arguments of a fragment function (and their corresponding data types).

Note: If the return type of a vertex function is not `void`, it must include the vertex position. If the vertex return type is `float4` this always refers to the vertex position (and the `[[ position ]]` qualifier need not be specified). If the vertex return type is a struct, it must include an element declared with the `[[ position ]]` qualifier.

Table 4-3 Attribute Qualifiers for Fragment Function Input Arguments

Attribute Qualifier	Corresponding Data Types	Description
`[[color(m)]]`	`floatn`, `halfn`, `intn`, `uintn`, `shortn`, or `ushortn` `m` must be known at compile time	The input value read from a color attachment. The index `m` indicates which color attachment to read from.
`[[front_facing]]`	`bool`	This value is `true` if the fragment belongs to a front-facing primitive.
`[[point_coord]]`	`float2`	Two-dimensional coordinates indicating where within a point primitive the current fragment is located. They range from 0.0 to 1.0 across the point.
`[[position]]`	`float4`	Describes the window relative coordinate (x, y, z, 1/w) values for the fragment.
`[[sample_id]]`	`uint`	The sample number of the sample currently being processed.
`[[sample_mask]]`	`uint`	The set of samples covered by the primitive generating the fragment during multisample rasterization.

A variable declared with the `[[ position ]]` attribute as input to a fragment function can only be declared with the `center_no_perspective` sampling and interpolation qualifier.

For [[color(m)]], m is used to specify the color attachment index when accessing (reading or writing) multiple color attachments in a fragment function.

Attribute Qualifiers for Fragment Function Output

The return type of a fragment function describes the per-fragment output. A fragment function can output one or more render-target color values, a depth value, and a coverage mask, which must be identified by using the attribute qualifiers listed in Table 4-4 (page 47). If the depth value is not output by the fragment function, the depth value generated by the rasterizer is output to the depth attachment.

Table 4-4 Attribute Qualifiers for Fragment Function Return Types

Attribute Qualifier	Corresponding Data Types
`[[color(m)]]`	`floatn`, `halfn`, `intn`, `uintn`, `shortn`, or `ushortn` `m` must be known at compile time
`[[depth(depth_qualifier)]]`	`float`
`[[sample_mask]]`	`uint`

The color attachment index m for fragment output is specified in the same way as it is for [[color(m)]] for fragment input (see discussion for Table 4-3 (page 46)).

If there is only a single color attachment in a fragment function, then [[color(m)]] is optional. If [[color(m)]] is not specified, the attachment index will be 0. If multiple color attachments are specified, [[color(m)]] must be specified for all color values. See examples of specifying the color attachment in Per-Fragment Function vs. Per-Sample Function (page 54) and Programmable Blending (page 55).

If a fragment function writes a depth value, the depth_qualifier must be specified with one of the following values: any, greater, or less.

The following example shows how color attachment indices can be specified. Color values written in clr_f write to color attachment index 0, clr_i to color attachment index 1, and clr_ui to color attachment index 2.

```
struct MyFragmentOutput {
    // color attachment 0
    float4 clr_f [[color(0)]];

    // color attachment 1
    int4 clr_i [[color(1)]];

    // color attachment 2
    uint4 clr_ui [[color(2)]];
};

fragment MyFragmentOutput
my_frag_shader( ... )
{
     MyFragmentOutput f;
     ....
     f.clr_f = ...;
     ...
     return f;
}
```

Note: If a color attachment index is used both as an input to and output of a fragment function, the data types associated with the input argument and output declared with this color attachment index must match.

Attribute Qualifiers for Kernel Function Input

When a kernel is submitted for execution, it executes over an N-dimensional grid of threads, where N is one, two or three. An instance of the kernel executes for each point in this grid. A thread is an instance of the kernel that executes for each point in this grid, and `thread_position_in_grid` identifies its position in the grid.

Threads are organized into **threadgroups**. Threads in a threadgroup cooperate by sharing data through `threadgroup` memory and by synchronizing their execution to coordinate memory accesses to both `device` and `threadgroup` memory. The threads in a given threadgroup execute concurrently on a single compute unit on the GPU. (A GPU may have multiple compute units. Multiple threadgroups can execute concurrently across multiple compute units.) Within a compute unit, a threadgroup is partitioned into multiple smaller groups for execution. The execution width of the compute unit, referred to as the `thread_execution_width`, determines the recommended size of this smaller group. For best performance, the total number of threads in the threadgroup should be a multiple of the `thread_execution_width`.

Threadgroups are assigned a unique position within the grid (referred to as `threadgroup_position_in_grid`) with the same dimensionality as the index space used for the threads. Threads are assigned a unique position within a threadgroup (referred to as `thread_position_in_threadgroup`). The unique scalar index of a thread within a threadgroup is given by `thread_index_in_threadgroup`.

Each thread's position in the grid and position in the threadgroup are N-dimensional tuples. Threadgroups are assigned a position using a similar approach to that used for threads. Threads are assigned to a threadgroup and given a position in the threadgroup with components in the range from zero to the size of the threadgroup in that dimension minus one.

When a kernel is submitted for execution, the number of threadgroups and the threadgroup size are specified. For example, consider a kernel submitted for execution that uses a 2-dimensional grid where the number of threadgroups specified are (W_x, W_y) and the threadgroup size is (S_x, S_y). Let (w_x, w_y) be the position of each threadgroup in the grid (i.e., `threadgroup_position_in_grid`), and (l_x, l_y) be the position of each thread in the threadgroup (i.e., `thread_position_in_threadgroup`).

The thread position in the grid (i.e., `thread_position_in_grid`) is:

$$(g_x, g_y) = (w_x * S_x + l_x, w_y * S_y + l_y)$$

The grid size (i.e., `threads_per_grid`) is:

$$(G_x, G_y) = (W_x * S_x , W_y * S_y)$$

The thread index in the threadgroup (i.e., `thread_index_in_threadgroup`) is:

$$l_y * S_x + l_x$$

Table 4-5 (page 49) lists the built-in attribute qualifiers that can be specified for arguments to a compute function and the corresponding data types with which they can be used.

Table 4-5 Attribute Qualifiers for Kernel Function Input Arguments

Attribute Qualifier	Corresponding Data Types
`[[thread_position_in_grid]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`
`[[thread_position_in_threadgroup]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`
`[[thread_index_in_threadgroup]]`	`uint`, `ushort`
`[[threadgroup_position_in_grid]]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`
`[[threads_per_grid]]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`
`[[threads_per_threadgroup]]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`
`[[threadgroups_per_grid]]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`
`[[thread_execution_width]]`	`ushort` or `uint`

Notes on kernel function attribute qualifiers:

- The type used to declare `[[thread_position_in_grid]]`, `[[threads_per_grid]]`, `[[thread_position_in_threadgroup]]`, `[[threads_per_threadgroup]]`, `[[threadgroup_position_in_grid]]` and `[[threadgroups_per_grid]]` must be a scalar type or a vector type. If it is a vector type, the number of components for the vector types used to declare these arguments must match.
- The data types used to declare `[[thread_position_in_grid]` and `[[threads_per_grid]]` must match.
- The data types used to declare `[[thread_position_in_threadgroup]]` and `[[threads_per_threadgroup]]` must match.
- If `thread_position_in_threadgroup` is declared to be of type `uint`, `uint2`, or `uint3`, then `[[thread_index_in_threadgroup]]` must be declared to be of type `uint`.

stage_in Qualifier

The per-fragment inputs to a fragment function are generated using the output from a vertex function and the fragments generated by the rasterizer. The per-fragment inputs are identified using the `[[stage_in]]` attribute qualifier.

A vertex function can read per-vertex inputs by indexing into a buffer(s) passed as arguments to the vertex function using the vertex and instance IDs. In addition, per-vertex inputs can also be passed as arguments to a vertex function by declaring them with the `[[stage_in]]` attribute qualifier.

Only one argument of the fragment or vertex function can be declared with the stage_in qualifier. For a user-defined struct declared with the `stage_in` qualifier, the members of the struct can be:

- a scalar integer or floating-point value or
- a vector of integer or floating-point values.

Note: Packed vectors, matrices, structs, references or pointers to a type, and arrays of scalars, vectors, and matrices are not supported as members of the struct declared with the `stage_in` qualifier.

Vertex function example that uses the stage_in qualifier

The following example shows how to pass per-vertex inputs using the `stage_in` qualifier.

```
#include <metal_stdlib>
using namespace metal;

struct VertexOutput {
    float4 position [[position]];
    float4 color;
    float2 texcoord[4];
};

struct VertexInput {
    float4 position [[ attribute(0) ]];
    float3 normal   [[ attribute(1) ]];
    half4 color     [[ attribute(2) ]];
    half2 texcoord  [[ attribute(3) ]];
};

constexpr constant uint MAX_LIGHTS = 4;

struct LightDesc {
    uint   num_lights;
    float4 light_position[MAX_LIGHTS];
    float4 light_color[MAX_LIGHTS];
    float4 light_attenuation_factors[MAX_LIGHTS];
};

constexpr sampler s = sampler(coord::normalized,
                              address::clamp_to_zero,
                              filter::linear);


vertex VertexOutput
render_vertex(VertexInput v_in [[ stage_in ]],
              constant float4x4& mvp_matrix [[ buffer(1) ]],
              constant LightDesc& lights [[ buffer(2) ]],
              uint v_id [[ vertex_id ]])
{
    VertexOutput v_out;
    v_out.position = v_in.position * mvp_matrix;
    v_out.color = do_lighting(v_in.position,
                              v_in.normal,
                              lights);
    ...
    return v_out;
}
```

Fragment function example that uses the stage_in qualifier

An example in Attribute Qualifiers to Locate Per-Vertex Inputs (page 42) previously introduces the `process_vertex` vertex function, which returns a `VertexOutput` struct per vertex. In the following example, the output from `process_vertex` is pipelined to become input for a fragment function called `render_pixel`, so the first argument of the fragment function uses the `[[stage_in]]` qualifier and the incoming `VertexOutput` type. (In `render_pixel`, the `imgA` and `imgB` 2D textures call the built-in function `sample`, which is introduced in 2D Texture (page 76).)

```
#include <metal_stdlib>
using namespace metal;

struct VertexOutput {
    float4 position [[position]];
    float4 color;
    float2 texcoord;
};

struct VertexInput {
    float4 position;
    float3 normal;
    float2 texcoord;
};

constexpr constant uint MAX_LIGHTS = 4;

struct LightDesc {
    uint   num_lights;
    float4 light_position[MAX_LIGHTS];
    float4 light_color[MAX_LIGHTS];
    float4 light_attenuation_factors[MAX_LIGHTS];
};

constexpr sampler s = sampler(coord::normalized,
                              address::clamp_to_border,
                              filter::linear);

vertex VertexOutput
render_vertex(const device VertexInput* v_in [[ buffer(0) ]],
              constant float4x4& mvp_matrix [[ buffer(1) ]],
              constant LightDesc& lights [[ buffer(2) ]],
              uint v_id [[ vertex_id ]])
{
    VertexOutput v_out;
    v_out.position = v_in[v_id].position * mvp_matrix;
    v_out.color = do_lighting(v_in[v_id].position,
                              v_in[v_id].normal,
                              lights);
    v_out.texcoord = v_in[v_id].texcoord;
    return v_out;
}

fragment float4
render_pixel(VertexOutput input [[stage_in]],
             texture2d<float> imgA [[ texture(0) ]],
             texture2d<float> imgB [[ texture(1) ]])
{
    float4 tex_clr0 = imgA.sample(s, input.texcoord);
    float4 tex_clr1 = imgB.sample(s, input.texcoord);
    // compute color
    float4 clr = compute_color(tex_clr0, tex_clr1, ...);
    return clr;
}
```

Storage Class Specifiers

The Metal shading language supports the `static` and `extern` storage class specifiers. Metal does not support the `thread_local` storage class specifiers.

The `extern` storage-class specifier can only be used for functions and variables declared in program scope or variables declared inside a function. The `static` storage-class specifier is only for variables declared in program scope (see constant Address Space (page 39)) and is not for variables declared inside a graphics or kernel function. In the following example, the `static` specifier is incorrectly used by the variables `b` and `c` declared inside a kernel function.

```
#include <metal_stdlib>
using namespace metal;

extern constant float4 noise_table[256];
static constant float4 color_table[256] = { ... };  // static is okay

extern void my_foo(texture2d<float> img);
extern void my_bar(device float *a);

kernel void my_func(texture2d<float> img [[ texture(0) ]],
                    device float *ptr [[ buffer(0) ]])
{
    extern constant float4 a;
    static constant float4 b;    // static is an error.
    static float c;             // static is an error.
    ...
    my_foo(img);
    ...
    my_bar(ptr);
    ...
}
```

Sampling and Interpolation Qualifiers

Sampling and interpolation qualifiers are used with inputs to fragment functions declared with the `stage_in` qualifier. The qualifier determines what sampling method the fragment function uses and how the interpolation is performed, including whether to use perspective-correct interpolation, linear interpolation, or no interpolation.

The sampling and interpolation qualifier can be specified on any structure member declared with the `stage_in` qualifier. The sampling and interpolation qualifiers supported are:

```
center_perspective
center_no_perspective
centroid_perspective
centroid_no_perspective
sample_perspective
sample_no_perspective
flat
```

`center_perspective` is the default sampling and interpolation qualifier for all attribute qualfiiers for return types of vertex functions and arguments to fragment functions, except for `[[position]]`, which can only be declared with `center_no_perspective`.

The following example is user-defined struct that specifies how data in certain members are interpolated:

```
struct FragmentInput {
        float4 pos [[center_no_perspective]];
        float4 color [[center_perspective]];
        float2 texcoord;
        int index [[flat]];
        float f [[sample_perspective]];
};
```

For integer types, the only valid interpolation qualifier is `flat`.

The sampling qualifier variants (`sample_perspective` and `sample_no_perspective`) interpolate at a sample location rather than at the pixel center. With one of these qualifiers, the fragment function or code blocks in the fragment function that use these variables execute per-sample rather than per-fragment.

Per-Fragment Function vs. Per-Sample Function

The fragment function is typically executed per-fragment. The sampling qualifier identifies if any fragment input is to be interpolated at per-sample vs. per-fragment. Similarly, the `[[sample_id]]` attribute is used to identify the current sample index and the `[[color(m)]]` attribute is used to identify the destination fragment color or sample color (for a multisampled color attachment) value. If any of these qualifiers are used with arguments to a fragment function, the fragment function may execute per-sample instead of per-pixel. (The implementation may decide to only execute the code per-sample that depends on the per-sample values, and the rest of the fragment function may execute per-fragment.)

Only the inputs with sample specified (or declared with the `[[sample_id]]` or `[[color(m)]]` qualifier) differ between invocations per-fragment or per-sample, whereas other inputs still interpolate at the pixel center.

The following example uses the `[[color(m)]]` attribute to specify that this fragment function should be executed on a per-sample basis.

```
#include <metal_stdlib>
using namespace metal;

fragment float4
my_frag_shader(float2 tex_coord [[ stage_in ]],
               texture2d<float> img [[ texture(0) ]],
               sampler s [[ sampler(0) ]],
               float4 framebuffer [[color(0)]])
{
    return c = mix(img.sample(s, tex_coord), framebuffer,
                   mix_factor);
}
```

Programmable Blending

The fragment function can be used to perform per-fragment or per-sample programmable blending. The color attachment index identified by the `[[color(m)]]` attribute qualifier can be specified as an argument to a fragment function.

Below is the programmable blending example from Allan Schaffer's *Advances in OpenGL and OpenGL ES* talk at WWDC 2012 that describes how to paint grayscale onto what is below. (Look for the talk and slide set among the WWDC 2012 talks at https://developer.apple.com/videos/wwdc/2012/)

The GLSL version is:

```
#extension GL_APPLE_shader_framebuffer_fetch : require

void main()
{
    // RGB to grayscale
    mediump float lum = dot(gl_LastFragData[0].rgb,
                            vec3(0.30,0.59,0.11));
    gl_FragColor = vec4(lum, lum, lum, 1.0);
}
```

The equivalent Metal function is:

```
#include <metal_stdlib>
using namespace metal;

fragment half4
paint_grayscale(half4 dst_color [[color(0)]])
{

    // RGB to grayscale
    half lum = dot(dst_color.rgb,
                   half3(0.30h, 0.59h, 0.11h));
    return half4(lum, lum, lum, 1.0h);
}
```

Graphics Function – Signature Matching

A graphics function signature is a list of parameters that are either input to or output from a graphics function.

Vertex – Fragment Signature Matching

The per-instance input to a fragment function is declared with the `[[stage_in]]` qualifier. These are output by an associated vertex function.

Built-in variables are declared with one of the attribute qualifiers defined in Attribute Qualifiers to Locate Per-Vertex Inputs (page 42). These are either generated by a vertex function (such as `[[position]]`, `[[point_size]]`, `[[clip_distance]]`), are generated by the rasterizer (such as `[[point_coord]]`, `[[front_facing]]`, `[[sample_id]]`, `[[sample_mask]]`) or refer to a framebuffer color value (such as `[[color]]`) passed as an input to the fragment function.

The built-in variable `[[position]]` must always be returned. The other built-in variables (`[[point_size]]`, `[[clip_distance]]`) generated by a vertex function, if needed, must be declared in the return type of the vertex function but cannot be accessed by the fragment function.

Built-in variables generated by the rasterizer or refer to a framebuffer color value may also declared as arguments of the fragment function with the appropriate attribute qualifier.

The attribute `[[user(name)]]` syntax can also be used to specify an attribute name for any user-defined variables.

A vertex and fragment function are considered to have matching signatures if:

- There is no input argument with the `[[stage_in]]` qualifier declared in the fragment function.

- For a fragment function argument declared with `[[stage_in]]`, each element in the type associated with this argument can be one of the following: a built-in variable generated by the rasterizer, a framebuffer color value passed as input to the fragment function, or a user-generated output from a vertex function. For built-in variables generated by the rasterizer or framebuffer color values, there is no requirement for a matching type to be associated with elements of the vertex return type. For elements that are user-generated outputs, the following rules apply:
    - If the attribute name given by `[[user(name)]]` is specified for an element, then this attribute name must match with an element in the return type of the vertex function, and their corresponding data types must also match.
    - If the `[[user(name)]]` attribute name is not specified, then the argument name and types must match.

Below is an example of compatible signatures:

```
struct VertexOutput
{
    float4 position [[position]];
    float3 normal;
    float2 texcoord;
};

vertex VertexOutput
my_vertex_shader(...)
{
    VertexOutput v;

    ...
    return v;
}

fragment float4
my_fragment_shader(VertexOutput f [[stage_in]], ...)
{
    float4 clr;

    ...
    return clr;
}

fragment float4
my_fragment_shader2(VertexOutput f [[stage_in]],
                    bool is_front_face [[front_facing]], ...)
{
    float4 clr;

    ...
    return clr;
}
```

`my_vertex_shader` and `my_fragment_shader`, or `my_vertex_shader` and `my_fragment_shader2` can be used together to render a primitive.

Below is another example of compatible signatures:

```
struct VertexOutput
{
    float4 position [[position]];
    float3 vertex_normal [[user(normal)]];
    float2 texcoord [[user(texturecoord)]];
};

struct FragInput
{
    float3 frag_normal [[user(normal)]];
    float4 position [[position]];
    float4 framebuffer_color [[color(0)]];
    bool is_front_face [[front_facing]];
};

vertex VertexOutput
my_vertex_shader(...)
{
    VertexOutput v;

    ...
    return v;
}

fragment float4
my_fragment_shader(FragInput f [[stage_in]], ...)
{
    float4 clr;

    ...
    return clr;
}
```

Below is another example of compatible signatures:

```
struct VertexOutput
{
    float4 position [[position]];
    float3 normal;
    float2 texcoord;
};

struct FragInput
{
    float4 position [[position]];
    float2 texcoord;
};

vertex VertexOutput
my_vertex_shader(...)
{
    VertexOutput v;

    ...
    return v;
}

fragment float4
my_fragment_shader(FragInput f [[stage_in]], ...)
{
    float4 clr;

    ...
    return clr;
}
```

Below is another example of compatible signatures:

```
struct VertexOutput
{
    float4 position [[position]];
    float3 normal;
    float2 texcoord;
};

vertex VertexOutput
my_vertex_shader(...)
{
    VertexOutput v;

    ...
    return v;
}

fragment float4
my_fragment_shader(float4 p [[position]], ...)
{
    float4 clr;

    ...
    return clr;
}
```

Below is an example of **in**compatible signatures:

```
struct VertexOutput
{
    float4 position [[position]];
    float3 normal;
    float2 texcoord;
};

struct FragInput
{
    float4 position [[position]];
    half3 normal;
};

vertex VertexOutput
my_vertex_shader(...)
{
    VertexOutput v;

    ...
    return v;
}

fragment float4
my_fragment_shader(FragInput f [[stage_in]], ...)
{
    float4 clr;

    ...
    return clr;
}
```

Below is another example of incompatible signatures:

```
struct VertexOutput
{
    float4 position [[position]];
    float3 normal [[user(normal)]];
    float2 texcoord [[user(texturecoord)]];
};

struct FragInput
{
    float3 normal [[user(foo)]];
    float4 position [[position]];
};

vertex VertexOutput
my_vertex_shader(...)
{
    VertexOutput v;

    ...
    return v;
}

fragment float4
my_fragment_shader(FragInput f [[stage_in]], ...)
{
    float4 clr;

    ...
    return clr;
}
```

Additional Restrictions

Writes to a buffer or a texture are disallowed from a fragment function.

Writes to a texture are disallowed from a vertex shader.

Writes to a buffer from a vertex function are not guaranteed to be visible to reads from the associated fragment function of a given primitive.

The return type of a vertex or fragment function cannot include an element that is a packed vector type, matrix type, array type, or a reference or a pointer to a type.

A maximum of 128 scalars can be used as inputs to a fragment function declared with the `stage_in` qualifier. (The restriction does not include the built-in variables declared with one of the following attributes: `[[ color(m) ]]`, `[[ front_facing ]]`, `[[ sample_id ]]`, and `[[ sample_mask ]]`.) If an input to a fragment function is a vector, then the input vector counts as *n* scalars where *n* is the number of components in the vector.

Metal Standard Library

This chapter describes the functions supported by the Metal shading language standard library.

Namespace and Header Files

The Metal standard library functions and enums are declared in the `metal` namespace. In addition to the header files described in the Metal standard library functions, the `<metal_stdlib>` header is available and can access all the functions supported by the Metal standard library.

Common Functions

The functions in this section are in the Metal standard library and are defined in the header `<metal_common>`. `T` is one of the scalar or vector floating-point types.

`T clamp(T x, T minval, T maxval)`

Returns `fmin(fmax(x, minval), maxval)`. Results are undefined if `minval` > `maxval`.

`T mix(T x, T y, T 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.

`T saturate(T x)`

Clamp `x` within the range of 0.0 to 1.0.

`T sign(T 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.

`T smoothstep(T edge0, T edge1, T x)`

Returns 0.0 if `x` <= `edge0` and 1.0 if `x` >= `edge1` and performs a smooth Hermite interpolation between 0 and 1 when `edge0` < `x` < `edge1`. This is useful in cases where you want a threshold function with a smooth transition. Results are undefined if `edge0` >= `edge1` or if `x`, `edge0` or `edge1` is a NaN. This is equivalent to:

```
t = clamp((x – edge0)/(edge1 – edge0), 0, 1);
return t * t * (3 – 2 * t);
```

`T step(T edge, T x)`

Returns 0.0 if `x` < `edge`, otherwise it returns 1.0.

For single-precision floating-point, Metal also supports a precise and fast variant of the following common functions: `clamp` and `saturate`. The difference between the fast and precise variants is how NaNs are handled. In the fast variant, the behavior of NaNs is undefined, whereas the precise variants follow the IEEE 754 rules for NaN handling. The `–ffast–math` compiler option (see Math Intrinsics Options (page 93)) is used to select the appropriate variant when compiling the Metal source. In addition, the `metal::precise` and `metal::fast` nested namespaces are available to provide developers a way to explicitly select the fast or precise variant of these common functions, respectively.

Integer Functions

The functions in this section are in the Metal standard library and are defined in the header `<metal_integer>`. `T` is one of the scalar or vector integer types. `Tu` is the corresponding unsigned scalar or vector integer type.

`T abs(T x))`

Returns | x |.

`Tu absdiff(T x, T y)`

Returns | `x – y` | without modulo overflow.

`T clamp(T x, T minval, T maxval)`

Returns `min(max(x, minval), maxval)`. Results are undefined if `minval` > `maxval`.

`T clz(T 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.

`T ctz(T 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.

`T hadd(T x, T y)`

Returns (`x` + `y`) >> 1. The intermediate sum does not modulo overflow.

`T madhi(T a, T b, T c)`

Returns `mulhi(a, b) + c`.

`T madsat(T a, T b, T c)`

Returns `a * b + c` and saturates the result.

`T max(T x, T y)`

Returns `y` if `x` < `y`, otherwise it returns `x`.

`T min(T x, T x)`

Returns `y` if `y` < `x`, otherwise it returns `x`.

`T mulhi(T x, T y)`

Computes `x * y` and returns the high half of the product of `x` and `y`.

`T popcount(T x)`

Returns the number of non-zero bits in `x`.

`T rhadd(T x, T y)`

Returns (`x` + `y` + 1) >> 1. The intermediate sum does not modulo overflow.

`T rotate(T v, T i)`

For each element in `v`, the bits are shifted left by the number of bits given by the corresponding element in `i`. Bits shifted off the left side of the element are shifted back in from the right.

`T subsat(T x, T y)`

Returns `x` - `y` and saturates the result.

Relational Functions

The functions in this section are in the Metal standard library and are defined in the header `<metal_relational>`. `T` is one of the scalar or vector floating-point types. `Ti` is one of the scalar or vector integer or boolean types. `Tb` only refers to the scalar or vector boolean types.

`bool all(Tb x)`

Returns true only if all components of `x` are true.

`bool any(Tb x)`

Returns true only if any component of `x` is true.

`Tb isfinite(T x)`

Tests for finite value.

`Tb isinf(T x)`

Tests for infinity value (positive or negative).

`Tb isnan(T x)`

Tests for a NaN.

`Tb isnormal(T x)`

Tests for a normal value.

`Tb isordered(T x, T y)`

Tests if arguments `x` and `y` are ordered. Returns the result `(x == x) && (y == y)`.

`Tb isunordered(T x, T y)`

Tests if arguments `x` and `y` are unordered. Returns true if `x` or `y` is NaN and false otherwise.

`Tb not(Tb x)`

Returns the component-wise logical complement of `x`.

`T select(T a, T b, Tb c)`

`Ti select(Ti a, Ti b, Tb c)`

For each component of a vector type, `result[i] = c[i] ? b[i] : a[i]`

For a scalar type, `result = c ? b : a`.

`Tb signbit(T x)`

Tests for sign bit. Returns true if the sign bit is set for the floating-point value in `x` and false otherwise.

Math Functions

The functions in this section are in the Metal standard library and are defined in the header `<metal_math>`. `T` is one of the scalar or vector floating-point types. `Ti` refers only to the scalar or vector integer types.

`T abs(T x)`

`T fabs(T x)`

Computes absolute value of `x`.

`T acos(T x)`

Computes arc cosine function of `x`.

`T acosh(T x)`

Computes inverse hyperbolic cosine of `x`.

`T asin(T x)`

Computes arc sine function of `x`.

`T asinh(T x)`

Computes inverse hyperbolic sine of `x`.

`T atan(T y_over_x)`

Computes arc tangent function of `y_over_x`.

`T atan2(T y, T x)`

Computes arc tangent of `y` over `x`.

`T atanh(T x)`

Computes hyperbolic arc tangent of `x`.

`T ceil(T x)`

Rounds `x` to integral value using the round to positive infinity rounding mode.

`T copysign(T x, T y)`

Returns `x` with its sign changed to match the sign of `y`.

`T cos(T x)`

Computes cosine of `x`.

`T cosh(T x)`

Computes hyperbolic cosine of `x`.

`T exp(T x)`

Computes the base-e exponential of `x`.

`T exp2(T x)`

Computes the base-2 exponential of `x`.

`T exp10(T x)`

Computes the base-10 exponential of `x`.

`T fdim(T x, T y)`

Returns `x – y` if `x > y`, +0 if `x` is less than or equal to `y`.

`T floor(T x)`

Rounds `x` to integral value using the round to negative infinity rounding mode.

`T fmod(T x, T y)`

Returns `x – y * trunc(x/y)`.

`T fract(T x)`

Returns the fractional part of `x` which is greater than or equal to 0 or less than 1.

`T frexp(T x, Ti &exp)`

Extracts mantissa and exponent from `x`. For each component the mantissa returned is a float with magnitude in the interval [1/2, 1) or 0. Each component of `x` equals mantissa returned * 2^{exp}.

`Ti ilogb(T x)`

Returns the exponent as an integer value.

`T ldexp(T x, Ti k)`

Multiply `x` by 2 to the power `k`.

`T log(T x)`

Computes natural logarithm of `x`.

`T log2(T x)`

Computes a base-2 logarithm of `x`.

`T log10(T x)`

Computes a base-10 logarithm of `x`.

`T max(T x, T y)`

`T fmax(T x, T 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. (The NaN behavior is only valid for `precise::fmax` and `precise::max`.)

`T min(T x, T y)`

`T fmin(T x, T 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. (The NaN behavior is only valid for `precise::fmin` and `precise::min`.)

`T modf(T x, T &intval)`

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.

Returns the fractional value. The integral value is returned in `intval`.

`T pow(T x, T y)`

Computes `x` to the power `y`.

`T powr(T x, T y)`

Computes `x` to the power `y`, where `x` is >= 0.

`T rint(T x)`

Rounds `x` to integral value using round to nearest even rounding mode in floating-point format.

`T round(T x)`

Returns the integral value nearest to `x` rounding halfway cases away from zero.

`T rsqrt(T x)`

Computes inverse square root of `x`.

`T sin(T x)`

Computes sine of `x`.

`T sincos(T x, T &cosval)`

Computes sine and cosine of `x`. The computed sine is the return value and the computed cosine is returned in `cosval`.

`T sinh(T x)`

Computes hyperbolic sine of `x`.

`T sqrt(T x)`

Computes square root of `x`.

`T tan(T x)`

Computes tangent of `x`.

`T tanh(T x)`

Computes hyperbolic tangent of `x`.

`T trunc(T x)`

Rounds `x` to integral value using the round to zero rounding mode.

For single precision floating-point, the Metal shading language supports two variants of the math functions listed above: the precise and the fast variants. The `-ffast-math` compiler option (refer to Math Intrinsics Options (page 93)) is used to select the appropriate variant when compiling Metal shading language source. In addition, the `metal::precise` and `metal::fast` nested namespaces are also available and provide developers a way to explicitly select the fast or precise variant of these math functions for single precision floating-point.

Examples:

```
#include <metal_stdlib>
using namespace metal;

float x;
float a = sin(x); // use fast or precise version of sin based on
                  // whether -ffast-math is specified as compile option.
float b = fast::sin(x);    // use fast version of sin()
float c = precise::cos(x); // use precise version of cos()
```

Matrix Functions

The functions in this section are in the Metal standard library and are defined in the header `<metal_matrix>`.

`float determinant(floatnxn)`

`half determinant(halfnxn)`

Computes the determinant of `matrix`, which must be a square matrix.

`floatmxn transpose(floatnxm)`

`halfmxn transpose(halfnxm)`

Transpose `matrix`.

Example:

```
float4x4 mA;
float det = determinant(mA);
```

Geometric Functions

The functions in this section are in the Metal standard library and are defined in the header `<metal_geometric>`. `T` is a vector floating-point type (`floatn` or `halfn`). `Ts` refers to the corresponding scalar type (i.e. `float` if `T` is `floatn` and `half` if `T` is `halfn`).

`T cross(T x, T y)`

Returns the cross product of `x` and `y`. `T` must be a 3-component vector type.

`Ts distance(T x, T y)`

Returns the distance between `x` and `y`; i.e., `length(x – y)`

`Ts distance_squared(T x, T y)`

Returns the square of the distance between `x` and `y`.

`Ts dot(T x, T y)`

Returns the dot product of `x` and `y`; i.e., `x[0] * y[0] + x[1] * y[1] + …`

`T faceforward(T N, T I, T Nref)`

If `dot(Nref, I) < 0.0` returns `N`, otherwise returns –`N`.

`Ts length(T x)`

Returns the length of vector `x`; i.e., `sqrt(x[0]^2 + x[1]^2 + …)`

`Ts length_squared(T x)`

Returns the square of the length of vector `x`; i.e., `x[0]^2 + x[1]^2 + …`

`T normalize(T x)`

Returns a vector in the same direction as `x` but with a length of 1.

`T reflect(T I, T N)`

For the incident vector `I` and surface normal `N`, returns the reflection direction: `I – 2 * dot(N, I) * N`

In order to achieve the desired result, `N` must be normalized.

`T refract(T I, T N, Ts eta)`

For the incident vector `I`, the surface normal `N`, and the ratio of indices of refraction `eta`, returns the refraction vector. The input parameters for the incident vector `I` and the surface normal `N` must already be normalized to get the desired results.

For single precision floating-point, Metal also supports a precise and fast variant of the following geometric functions: `distance`, `length`, and `normalize`. The `–ffast–math` compiler option (refer to Math Intrinsics Options (page 93)) is used to select the appropriate variant when compiling the Metal shading language source. In addition, the `metal::precise` and `metal::fast` namespaces are also available and provide developers a way to explicitly select the fast or precise variant of these geometric functions.

Compute Functions

The functions in this section and its subsections can only be called from a `kernel` function and are defined in the header `<metal_compute>`.

threadgroup Synchronization Functions

The threadgroup function described below is supported.

`void threadgroup_barrier(mem_flags flags)`
All threads in a threadgroup executing the kernel must execute this function before any thread is allowed to continue execution beyond the `threadgroup_barrier`.

The `threadgroup_barrier` function acts as an execution and memory barrier. The `threadgroup_barrier` function must be encountered by all threads in a threadgroup executing the kernel.

If `threadgroup_barrier` is inside a conditional statement and if any thread enters the conditional statement and executes the barrier, then all threads in the threadgroup must enter the conditional and execute the barrier.

If `threadgroup_barrier` is inside a loop, for each iteration of the loop, all threads in the threadgroup must execute the `threadgroup_barrier` before any threads are allowed to continue execution beyond the `threadgroup_barrier`.

The `threadgroup_barrier` function can also queue a memory fence (reads and writes) to ensure correct ordering of memory operations to threadgroup or device memory.

The `mem_flags` argument in `threadgroup_barrier` can be one of the following flags, as described in Table 5-1 (page 72).

Table 5-1 `mem_flags` Enum Values for `threadgroup_barrier`

mem_flags	Description
`mem_none`	In this case, no memory fence is applied, and `threadgroup_barrier` acts only as an execution barrier.
`mem_device`	Ensure correct ordering of memory operations to device memory.
`mem_threadgroup`	Ensure correct ordering of memory operations to threadgroup memory for threads in a threadgroup.

mem_flags	Description
`mem_device_and_threadgroup`	Ensure correct ordering of memory operations to device and threadgroup memory for threads in a threadgroup.

The enumeration types used by `mem_flags` are specified as follows:

```
enum class mem_flags {mem_none,
                      mem_device,
                      mem_threadgroup,
                      mem_device_and_threadgroup };
```

Graphics Functions

This section and its subsections list the set of graphics functions that can be called by fragment functions. These are defined in the header `<metal_graphics>`.

Fragment Functions

The functions in these subsections can only be called inside a fragment function (a function declared with the `fragment` qualifier) or inside a function called from a fragment function. Otherwise the behavior is undefined and may result in a compile-time error.

Fragment Functions – Derivatives

Metal includes the following functions to compute derivatives. `T` is one of `float`, `float2`, `float3`, `float4`, `half`, `half2`, `half3` or `half4`.

Note: Derivatives are undefined within non-uniform control flow.

`T dfdx(T p)`

Returns a high precision partial derivative of the specified value with respect to the screen space x coordinate.

`T dfdy(T p)`

Returns a high precision partial derivative of the specified value with respect to the screen space y coordinate.

`T fwidth(T p)`

Returns the sum of the absolute derivatives in x and y using local differencing for `p`; i.e. `fabs(dfdx(p)) + fabs(dfdy(p))`.

Fragment Functions – Samples

Metal includes the following per-sample functions.

`uint get_num_samples()`

Returns the number of samples for the multisampled color attachment.

`float2 get_sample_position(uint indx)`

Returns the normalized sample offset (x, y) for a given sample index `indx`. Values of x and y are in [-1.0, 1.0).

`get_num_samples` and `get_sample_position` return the number of samples for the color attachment and the sample offsets for a given sample index. For example, for transparency super-sampling, this can be used to shade per-fragment, but do the alpha test per-sample.

Fragment Functions – Flow Control

The following Metal function is used to terminate a fragment.

`void discard_fragment(void)`

Marks the current fragment as being terminated, and the output of the fragment function for this fragment is discarded.

Texture Functions

The texture functions are categorized into: sample from a texture, sample compare from a texture, read (sampler-less read) from a texture, gather from a texture, gather compare from a texture, write to a texture, and texture query functions.

These are defined in the header `<metal_texture>`.

The texture `sample`, `sample_compare`, `gather`, and `gather_compare` functions take an `offset` argument for a 2D texture, 2D texture array, 3D texture, and cube texture. The `offset` is an integer value that is applied to the texture coordinate before looking up each pixel. This integer value can be in the range -8 to +7. The default value is 0.

Overloaded variants of texture `sample` and `sample_compare` functions for a 2D texture, 2D texture array, 3D texture, and cube texture are available and allow the texture to be sampled using a bias that is applied to a mip-level before sampling or with user-provided gradients in the x and y direction.

Note: The texture `sample`, `sample_compare`, `gather`, and `gather_compare` functions require that the texture is declared with the `sample` access qualifier.

The texture `sample_compare` and `gather_compare` functions are only available for depth texture types.

The texture read functions require that the texture is declared with the `sample` or `read` access qualifier.

The texture write functions require that the texture is declared with the `write` access qualifier.

In this section, `Tv` is a 4-component vector type based on the templated type <T> used to declare the texture type. If `T` is `float`, `Tv` is `float4`. If `T` is `half`, `Tv` is `half4`. If `T` is `int`, `Tv` is `int4`. If `T` is `uint`, `Tv` is `uint4`. If `T` is `short`, `Tv` is `short4`. If `T` is `ushort`, `Tv` is `ushort4`.

1D Texture

The following built-in function samples from a 1D texture.

```
Tv sample(sampler s, float coord) const
```

The following built-in function performs sampler-less reads from a 1D texture.

```
Tv read(uint coord, uint lod = 0) const
```

The following built-in function writes to a specific mip-level of a 1D texture.

```
void write(Tv color, uint coord, uint lod = 0)
```

The following built-in 1D texture query function is provided.

```
uint get_width(uint lod = 0) const
uint get_num_mip_levels() const
```

1D Texture Array

The following built-in function samples from a 1D texture array.

```
Tv  sample(sampler s,
           float coord,
           uint array) const
```

The following built-in function performs sampler-less reads from a 1D texture array.

```
Tv  read(uint coord,
         uint array,
         uint lod = 0) const
```

The following built-in function writes to a specific mip-level of a 1D texture array.

```
void write(Tv  color,
           uint coord,
           uint array,
           uint lod = 0)
```

The following built-in 1D texture array query functions are provided.

```
uint get_width(uint lod = 0) const
uint get_array_size() const
uint get_num_mip_levels() const
```

2D Texture

The following data types and corresponding constructor functions are available to specify various sampling options.

```
bias(float value)
```

```
level(float lod)
gradient2d(float2 dPdx, float2 dPdy)
```

The following built-in functions sample from a 2D texture.

```
Tv sample(sampler s,
          float2 coord,
          int2 offset = int2(0)) const

Tv sample(sampler s,
          float2 coord,
          lod_options options,
          int2 offset = int2(0)) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradient2d`.

The following built-in function performs sampler-less reads from a 2D texture.

```
Tv read(uint2 coord, uint lod = 0) const
```

The following built-in function writes to a 2D texture.

```
void write(Tv color, uint2 coord, uint lod = 0)
```

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a 2D texture.

```
enum class component { x, y, z, w };
Tv gather(sampler s,
          float2 coord,
          int2 offset = int2(0),
          component c = component::x) const
```

The following built-in 2D texture query functions are provided.

```
uint get_width(uint lod = 0) const
uint get_height(uint lod = 0) const
uint get_num_mip_levels() const
```

2D Texture Sampling Example

The following code shows several uses of the 2D texture sample function, depending upon its arguments.

```
#include <metal_stdlib>
using namespace metal;

texture2d<float> tex;
sampler s;
float2 coord;
int2 offset;
float lod;

// no optional arguments
float4 clr = tex.sample(s, coord);

// sample using a mip-level
clr = tex.sample(s, coord, level(lod));

// sample with an offset
clr = tex.sample(s, coord, offset);

// sample using a mip-level and an offset
clr = tex.sample(s, coord, level(lod), offset);
```

2D Texture Array

The following built-in functions sample from a 2D texture array.

```
Tv  sample(sampler s,
           float2 coord,
           uint array,
           int2 offset = int2(0)) const

Tv  sample(sampler s,
           float2 coord,
           uint array,
           lod_options options,
           int2 offset = int2(0)) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradient2d`.

The following built-in function performs sampler-less reads from a 2D texture array.

```
Tv read(uint2 coord, uint array, uint lod = 0) const
```

The following built-in function writes to a 2D texture array.

```
void write(Tv color,
           uint2 coord,
           uint array,
           uint lod = 0)
```

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a 2D texture array.

```
Tv gather(sampler s,
          float2 coord,
          uint array,
          int2 offset = int2(0),
          component c = component::x) const
```

The following built-in 2D texture array query functions are provided.

```
uint get_width(uint lod = 0) const
uint get_height(uint lod = 0) const
uint get_array_size() const
uint get_num_mip_levels() const
```

3D Texture

The following data types and corresponding constructor functions are available to specify various sampling options.

```
bias(float value)
level(float lod)
gradient3d(float3 dPdx, float3 dPdy)
```

The following built-in functions sample from a 3D texture.

```
Tv  sample(sampler s,
           float3 coord,
           int3 offset = int3(0)) const

Tv  sample(sampler s,
           float3 coord,
           lod_options options,
           int3 offset = int3(0)) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradient3d`.

The following built-in function performs sampler-less reads from a 3D texture.

```
Tv read(uint3 coord, uint lod = 0) const
```

The following built-in function writes to a 3D texture.

```
void write(Tv color, uint3 coord, uint lod = 0)
```

The following built-in 3D texture query functions are provided.

```
uint get_width(uint lod = 0) const
uint get_height(uint lod = 0) const
uint get_depth(uint lod = 0) const
uint get_num_mip_levels() const
```

Cube Texture

The following data types and corresponding constructor functions are available to specify various sampling options.

```
bias(float value)
level(float lod)
gradientcube(float3 dPdx, float3 dPdy)
```

The following built-in functions sample from a cube texture.

```
Tv sample(sampler s,
          float3 coord) const
Tv sample(sampler s,
          float3 coord,
          lod_options options) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradientcube`.

The following built-in function writes to a cube texture.

Note: Table 5-2 (page 81) describes the cube texture face and the number used to identify the face.

Table 5-2 Cube Texture Face Number

Face number	Cube texture face
0	Positive X
1	Negative X
2	Positive Y
3	Negative Y
4	Positive Z
5	Negative Z

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a cube texture.

```
Tv gather(sampler s,
          float3 coord,
          component c = component::x) const
```

The following built-in cube texture query functions are provided.

```
uint get_width(uint lod = 0) const
uint get_height(uint lod = 0) const
uint get_num_mip_levels() const
```

2D Multisampled Texture

The following built-in function performs a sampler-less read from a 2D multisampled texture.

```
Tv read(uint2 coord, uint sample) const
```

The following built-in 2D multisampled texture query functions are provided.

```
uint get_width(uint lod = 0) const
uint get_height(uint lod = 0) const
uint get_num_samples() const
```

2D Depth Texture

The following data types and corresponding constructor functions are available to specify various sampling options.

```
bias(float value)
level(float lod)
gradient2d(float2 dPdx, float2 dPdy)
```

The following built-in functions sample from a 2D depth texture.

```
T sample(sampler s,
         float2 coord,
         int2 offset = int2(0)) const

T sample(sampler s,
         float2 coord,
         lod_options options,
         int2 offset = int2(0)) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradient2d`.

The following built-in functions sample from a 2D depth texture and compare a single component against the specified comparison value.

```
T sample_compare(sampler s,
                 float2 coord,
                 float compare_value,
                 int2 offset = int2(0)) const

T sample_compare(sampler s,
                 float2 coord,
                 float compare_value,
                 lod_options options,
                 int2 offset = int2(0)) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradient2d`. `T` must be a `float` type.

Note: `sample_compare` performs a comparison of `compare_value` against the pixel value (1.0 if the comparison passes and 0.0 if it fails). These comparison result values per-pixel are then blended together as in normal texture filtering and the resulting value between 0.0 and 1.0 is returned.

The following built-in function performs a sampler-less read from a 2D depth texture.

```
T read(uint2 coord, uint lod = 0) const
```

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a 2D depth texture.

```
Tv gather(sampler s,
          float2 coord,
          int2 offset = int2(0)) const
```

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a 2D depth texture and comparing these samples with a specified comparison value (1.0 if the comparison passes and 0.0 if it fails).

```
Tv gather_compare(sampler s,
                  float2 coord,
                  float compare_value,
                  int2 offset = int2(0)) const
```

`T` must be a `float` type.

The following built-in 2D depth texture query functions are provided.

```
uint get_width(uint lod = 0) const
uint get_height(uint lod = 0) const
uint get_num_mip_levels() const
```

2D Depth Texture Array

The following built-in functions sample from a 2D depth texture array.

```
T sample(sampler s,
         float2 coord,
         uint array,
         int2 offset = int2(0)) const

T sample(sampler s,
         float2 coord,
         uint array,
         lod_options options,
         int2 offset = int2(0)) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradient2d`.

The following built-in functions sample from a 2D depth texture array and compare a single component against the specified comparison value.

```
T sample_compare(sampler s,
                 float2 coord,
                 uint array,
                 float compare_value,
                 int2 offset = int2(0)) const

T sample_compare(sampler s,
                 float2 coord,
                 uint array,
                 float compare_value,
                 lod_options options,
                 int2 offset = int2(0)) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradient2d`. `T` must be a `float` type.

The following built-in function performs a sampler-less read from a 2D depth texture array.

```
T read(uint2 coord, uint array, uint lod = 0) const
```

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a 2D depth texture array.

```
Tv gather(sampler s,
          float2 coord,
          uint array,
          int2 offset = int2(0)) const
```

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a 2D depth texture array and comparing these samples with a specified comparison value.

```
Tv gather_compare(sampler s,
                  float2 coord,
                  uint array,
                  float compare_value,
                  int2 offset = int2(0)) const
```

T must be a `float` type.

The following built-in 2D depth texture array query functions are provided.

```
uint get_width(uint lod = 0) const
uint get_height(uint lod = 0) const
uint get_array_size() const
uint get_num_mip_levels() const
```

Cube Depth Texture

The following data types and corresponding constructor functions are available to specify various sampling options.

```
bias(float value)
level(float lod)
gradientcube(float3 dPdx, float3 dPdy)
```

The following built-in functions sample from a cube depth texture.

```
T sample(sampler s,
         float3 coord) const

T sample(sampler s,
         float3 coord,
          lod_options options) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradientcube`.

The following built-in functions sample from a cube depth texture and compare a single component against the specified comparison value.

```
T sample_compare(sampler s,
                 float3 coord,
                 float compare_value) const

T sample_compare(sampler s,
                 float3 coord,
                 float compare_value,
                 lod_options options) const
```

`lod_options` must be one of the following types: `bias`, `level`, or `gradientcube`. `T` must be a `float` type.

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a cube depth texture.

```
Tv gather(sampler s, float3 coord) const
```

The following built-in function does a gather of four samples that may be used for bilinear interpolation when sampling a cube texture and comparing these samples with a specified comparison value.

```
Tv gather_compare(sampler s,

                  float3 coord,

                  float compare_value) const
```

`T` must be a `float` type.

The following built-in cube depth texture query functions are provided.

```
uint get_width(uint lod = 0) const

uint get_height(uint lod = 0) const

uint get_num_mip_levels() const
```

2D Multisampled Depth Texture

The following built-in function performs a sampler-less read from a 2D multisampled depth texture.

```
T read(uint2 coord, uint sample) const
```

The following built-in 2D multisampled depth texture query functions are provided.

```
uint get_width() const
uint get_height() const
uint get_num_samples() const
```

Pack and Unpack Functions

This section lists the Metal functions for converting a vector floating-point data to and from a packed integer value. The functions are defined in the header `<metal_pack>`. Refer to Texture Addressing and Conversion Rules (page 100) for details on how to convert from a 8-bit, 10-bit or 16-bit signed or unsigned integer value to a normalized single- or half-precision floating-point value and vice-versa.

Unpack Integer(s); Convert to a Floating-Point Vector

The following functions unpack multiple values from a single unsigned integer and then convert them into floating-point values that are stored in a vector.

```
float4 unpack_unorm4x8_to_float(uint x)
float4 unpack_snorm4x8_to_float(uint x)
half4 unpack_unorm4x8_to_half(uint x)
half4 unpack_snorm4x8_to_half(uint x)
```

Unpack a 32-bit unsigned integer into four 8-bit signed or unsigned integers and then convert each 8-bit signed or unsigned integer value to a normalized single- or half-precision floating-point value to generate a 4-component vector.

```
float4 unpack_unorm4x8_srgb_to_float(uint x)
half4 unpack_unorm4x8_srgb_to_half(uint x)
```

Unpack a 32-bit unsigned integer into four 8-bit signed or unsigned integers and then convert each 8-bit signed or unsigned integer value to a normalized single- or half-precision floating-point value to generate a 4-component vector. The r, g, and b color values are converted from sRGB to linear RGB.

`float2 unpack_unorm2x16_to_float(uint x)`

`float2 unpack_snorm2x16_to_float(uint x)`

`half2 unpack_unorm2x16_to_half(uint x)`

`half2 unpack_snorm2x16_to_half(uint x)`

Unpack a 32-bit unsigned integer into two 16-bit signed or unsigned integers and then convert each 16-bit signed or unsigned integer value to a normalized single- or half-precision floating-point value to generate a 2-component vector. (The `unpack_unorm2x16_to_half` and `unpack_snorm2x16_to_half` functions may result in precision loss when converting from a 16-bit unorm or snorm value to a half-precision floating point.)

`float4 unpack_unorm10a2_to_float(uint x)`

`float3 unpack_unorm565_to_float(ushort x)`

`half4 unpack_unorm10a2_to_half(uint x)`

`half3 unpack_unorm565_to_half(ushort x)`

Convert a 1010102 (10a2), or 565 color value to the corresponding normalized single- or half-precision floating-point vector.

Convert Floating-Point Vector to Integers, then Pack the Integers

The following functions start with a floating-point vector, convert the components into integer values, and then pack the multiple values into a single unsigned integer.

`uint pack_float_to_unorm4x8(float4 x)`

`uint pack_float_to_snorm4x8(float4 x)`

`uint pack_half_to_unorm4x8(half4 x)`

`uint pack_half_to_snorm4x8(half4 x)`

Convert a 4-component vector normalized single- or half-precision floating-point value to four 8-bit integer values and pack these 8-bit integer values into a 32-bit unsigned integer.

`uint pack_float_to_srgb_unorm4x8(float4 x)`

`uint pack_half_to_srgb_unorm4x8(half4 x)`

Convert a 4-component vector normalized single- or half-precision floating-point value to four 8-bit integer values and pack these 8-bit integer values into a 32-bit unsigned integer. The color values are converted from linear RGB to sRGB.

`uint pack_float_to_unorm2x16(float2 x)`

`uint pack_float_to_snorm2x16(float2 x)`

`uint pack_half_to_unorm2x16(half2 x)`

`uint pack_half_to_snorm2x16(half2 x)`

Convert a 2-component vector of normalized single- or half-precision floating-point values to two 16-bit integer values and pack these 16-bit integer values into a 32-bit unsigned integer.

`uint pack_float_to_unorm10a2(float4)`

`ushort pack_float_to_unorm565(float3)`

`uint pack_half_to_unorm10a2(half4)`

`ushort pack_half_to_unorm565(half3)`

Convert a 4- or 3-component vector of normalized single- or half-precision floating-point values to a packed, 1010102 or 565 color integer value.

Atomic Functions

The Metal shading language implements a subset of the C++11 atomics and synchronization operations. For atomic operations, only a `memory_order` of `memory_order_relaxed` is supported. If the atomic operation is to threadgroup memory, the memory scope of these atomic operations is a threadgroup. If the atomic operation is to device memory, then the memory scope is the GPU device.

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

These are defined in the header `<metal_atomic>`.

Note: Atomic operations to device and threadgroup memory can only be performed inside a kernel function (a function declared with the `kernel` qualifier) or inside a function called from a kernel function.

The `memory_order` enum is defined as follows.

```
enum memory_order {
        memory_order_relaxed
};
```

Atomic Store Functions

These functions atomically replace the value pointed to by `obj` (`*obj`) with `desired`.

```
void atomic_store_explicit(volatile device atomic_int* obj,
                           int desired,
                           memory_order order)
void atomic_store_explicit(volatile device atomic_uint* obj,
                           uint desired,
                           memory_order order)
void atomic_store_explicit(volatile threadgroup atomic_int* obj,
                           int desired,
                           memory_order order)
void atomic_store_explicit(volatile threadgroup atomic_uint* obj,
                           uint desired,
                           memory_order order)
```

Atomic Load Functions

These functions atomically obtain the value pointed to by `obj`.

```
int atomic_load_explicit(volatile device atomic_int* obj,
                          memory_order order)
uint atomic_load_explicit(volatile device atomic_uint* obj,
                          memory_order order)
int atomic_load_explicit(volatile threadgroup atomic_int* obj,
                         memory_order order)
uint atomic_load_explicit(volatile threadgroup atomic_uint* obj,
                          memory_order order)
```

Atomic Exchange Functions

These functions atomically replace the value pointed to by `obj` with `desired` and return the value `obj` previously held.

```
int atomic_exchange_explicit(volatile device atomic_int *obj,
                             int desired,
                             memory_order order)
uint atomic_exchange_explicit(volatile device atomic_uint *obj,
                              uint desired,
                              memory_order order)
int atomic_exchange_explicit(volatile threadgroup atomic_int *obj,
                             int desired,
                             memory_order order)
uint atomic_exchange_explicit(volatile threadgroup atomic_uint *obj,
                              uint desired,
                              memory_order order)
```

Atomic Compare and Exchange Functions

These functions atomically compare the value pointed to by `obj` with the value in expected. If those values are equal, the function replaces `*obj` with `desired` (by performing a read-modify-write operation). The function returns the value `obj` previously held.

```
bool atomic_compare_exchange_weak_explicit(
                             volatile device atomic_int *obj,
                             int *expected,
                             int desired,
                             memory_order succ,
                             memory_order fail)
bool atomic_compare_exchange_weak_explicit(
                             volatile device atomic_uint *obj,
                             uint *expected,
                             uint desired,
                             memory_order succ,
                             memory_order fail)
bool atomic_compare_exchange_weak_explicit(
                             volatile threadgroup atomic_int *obj,
                             int *expected,
                             int desired,
                             memory_order succ,
                             memory_order fail)
bool atomic_compare_exchange_weak_explicit(
                             volatile threadgroup atomic_uint *obj,
                             uint *expected,
                             uint desired,
                             memory_order succ,
                             memory_order fail)
```

Atomic Fetch and Modify Functions

The following operations perform arithmetic and bitwise computations. All these operations are applicable to an object of any atomic type. The key, operator, and computation correspondence is given in Table 5-3 (page 91).

Table 5-3 Atomic Operation Function

key	operator	computation
`add`	+	addition
`and`	&	bitwise and
`max`	`max`	compute max
`min`	`min`	compute min

key	operator	computation
`or`	|	bitwise inclusive or
`sub`	–	subtraction
`xor`	^	bitwise exclusive or

Atomically replaces the value pointed to by `obj` with the result of the computation of the value specified by `key` and `arg`. These operations are atomic read-modify-write operations. For signed integer types, arithmetic is defined to use two's complement representation with silent wrap-around on overflow. There are no undefined results. It returns the value `obj` held previously.

```
int atomic_fetch_key_explicit(volatile device atomic_int *obj,
                              int arg,
                              memory_order order)
uint atomic_fetch_key_explicit(volatile device atomic_uint *obj,
                               uint arg,
                               memory_order order)
int atomic_fetch_key_explicit(volatile threadgroup atomic_int *obj,
                              int arg,
                              memory_order order)
uint atomic_fetch_key_explicit(volatile threadgroup atomic_uint *obj,
                               uint arg,
                               memory_order order)
```

Compiler Options

The Metal compiler can be used online (i.e. using the appropriate APIs to compile Metal shading language sources) or offline. Metal shading language source code that is compiled offline can be loaded as binaries, using the appropriate Metal framework API.

This chapter explains the compiler options supported by the Metal shading language compiler, which are categorized as pre-processor options, options for math intrinsics, options that control optimization and miscellaneous options. The online and offline Metal compiler support these options.

Pre-Processor Options

These options control the Metal preprocessor that 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 in a #define directive. This option may receive multiple options, which are processed in the order in which they appear. This option allows developers to compile Metal code to change which features are enabled or disabled.

`-I` *dir*

Add the directory *dir* to the list of directories to be searched for header files. This option is only available for the offline compiler.

Math Intrinsics Options

These options control compiler behavior regarding floating-point arithmetic. These options trade off between speed and correctness.

`-ffast-math (default)`

This option enables the optimizations for floating-point arithmetic that may violate the IEEE 754 standard. It also enables the high-precision variant of math functions for single-precision floating-point scalar and vector types. `-ffast-math` is the default.

`-fno-fast-math`

This option is the opposite of `-ffast-math`. It disables the optimizations for floating-point arithmetic that may violate the IEEE 754 standard. It also disables the high-precision variant of math functions for single-precision floating-point scalar and vector types.

Options to Request or Suppress Warnings

The following options are available.

`-Werror`

Make all warnings into errors.

`-W`

Inhibit all warning messages.

Numerical Compliance

This chapter covers how the Metal shading language represents floating-point numbers with regard to accuracy in mathematical operations. Metal is compliant to a subset of the IEEE 754 standard.

INF, NaN and Denormalized Numbers

INF must be supported for single precision floating-point numbers and are optional for half precision floating-point numbers.

NaNs must be supported for single precision floating-point numbers (with fast math disabled) and are optional for half precision floating-point numbers. If fast math is enabled the behavior of handling NaN (as inputs or outputs) is undefined.

Denormalized single or half precision floating-point numbers passed as input or produced as the output of single or half precision floating-point operations may be flushed to zero.

Rounding Mode

Either round to zero or round to nearest rounding mode may be supported for single precision and half precision floating-point operations.

Floating-Point Exceptions

Floating-point exceptions are disabled in Metal.

Relative Error as ULPs

Table 7-1 (page 96) describes the minimum accuracy of single-precision floating-point basic arithmetic operations and math functions given as ULP values. The reference value used to compute the ULP value of an arithmetic operation is the infinitely precise result.

Table 7-1 Minimum Accuracy of Floating-Point Operations and Functions

Math Function	Min Accuracy - ULP values
x + y	Correctly rounded
x - y	Correctly rounded
x * y	Correctly rounded
1.0 / x	<= 2.5 ulp for single precision
x / y	<= 2.5 ulp for single precision
acos(x)	<= 4 ulp
acosh(x)	<= 4 ulp
asin(x)	<= 4 ulp
asinh(x)	<= 4 ulp
atan(x)	<= 5 ulp
atan2(y, x)	<= 6 ulp
atanh(x)	<= 5 ulp
ceil(x)	Correctly rounded
copysign(x)	0 ulp
cos(x)	<= 4 ulp
cosh(x)	<= 4 ulp
exp(x)	<= 4 ulp
exp2(x)	<= 4 ulp
exp10(x)	<= 4 ulp
fabs(x)	0 ulp
fdim(x, y)	Correctly rounded
floor(x)	Correctly rounded
fma(x, y, z)	Correctly rounded — see note below the table

Math Function	Min Accuracy - ULP values
fmax(x, y)	0 ulp
fmin(x, y)	0 ulp
fmod(x, y)	0 ulp
fract(x)	Correctly rounded
frexp(x, y)	0 ulp
ilogb(x)	0 ulp
ldexp(x, y)	Correctly rounded
log(x)	<= 4 ulp
log2(x)	<= 4 ulp
log10(x)	<= 4 ulp
modf(x, i)	0 ulp
pow(x, y)	<= 16 ulp
powr(x, y)	<= 16 ulp
rint(x)	Correctly rounded
round(x)	Correctly rounded
rsqrt(x)	<= 2 ulp
sin(x)	<= 4 ulp
sincos(x, c)	<= 4 ulp
sinh(x)	<= 4 ulp
sqrt(x)	<= 3 ulp for single precision
tan(x)	<= 6 ulp
tanh(x)	<= 5 ulp
trunc(x)	Correctly rounded

Note: The `metal_math` header does not declare a `fma()` function that developers can directly call. However, the Metal compiler contracts floating-point expressions (i.e., a * b + c) to a fused multiply-add instruction.

Table 7-2 (page 98) describes the minimum accuracy of single-precision floating-point arithmetic operations given as ULP values with fast math enabled (which is the default unless -ffast-math-disable is specified as a compiler option).

Table 7-2 Minimum Accuracy of Operations and Functions with Fast Math Enabled

Math Function	Min Accuracy - ULP values
x + y	Correctly rounded
x - y	Correctly rounded
x * y	Correctly rounded
1.0 / x	<= 2.5 ulp for x in the domain of 2^{-126} to 2^{126}
x / y	<= 2.5 ulp for x in the domain of 2^{-126} to 2^{126}
acos(x)	<= 5 ulp
acosh(x)	Implemented as log(x + sqrt(x * x – 1.0))
asin(x)	<= 5 ulp for |x| >= 2^{-125}
asinh(x)	Implemented as log(x + sqrt(x * x + 1.0))
atan(x)	<= 5 ulp
atanh(x)	Implemented as 0.5 * (log(1.0 + x) / log(1.0 – x))
atan2(y, x)	Implemented as atan(y / x)
cos(x)	For x in the domain [-pi, pi], the maximum absolute error is <= 2^{-13}. Outside that domain, the error is larger.
cosh(x)	Implemented as 0.5 * (exp(x) + exp(-x))
exp(x)	<= 3 + floor(fabs(2 * x)) ulp
exp2(x)	<= 3 + floor(fabs(2 * x)) ulp

Math Function	Min Accuracy - ULP values
exp10(x)	Implemented as exp2(x * log2(10))
fma(x, y, z)	Implemented either as a correctly rounded fma or as a multiply and an add, both of which are correctly rounded.
log(x)	For x in the domain [0.5, 2], the maximum absolute error is <= 2^{-21}; otherwise the maximum error is <= 3 ulp
log2(x)	For x in the domain [0.5, 2], the maximum absolute error is <= 2^{-21}; otherwise the maximum error is <= 3 ulp
log10(x)	Implemented as log2(x) * log10(2)
pow(x, y)	Implemented as exp2(y * log2(x))
powr(x, y)	Implemented as exp2(y * log2(x))
round(x)	Returns a value equal to the nearest integer to x. The fraction 0.5 is rounded in a direction chosen by the implementation.
sin(x)	For x in the domain [-pi, pi], the maximum absolute error is <= 2^{-13}. Outside that domain, the error is larger.
sinh(x)	Implemented as 0.5 * (exp(x) – exp(-x))
sincos(x)	ulp values as defined for sin(x) and cos(x)
sqrt(x)	Implemented as 1.0 / rsqrt(x)
tan(x)	Implemented as sin(x) * (1.0 / cos(x))
tanh(x)	Implemented as (t – 1.0)/(t + 1.0) where t = exp(2.0 * x)

Edge Case Behavior in Flush To Zero Mode

If denormals are flushed to zero, then a function may return one of these two results:

- Any conforming result for non-flush-to-zero mode. If the result is a subnormal before rounding, it may be flushed to zero.
- Any non-flushed conforming result for the function if one or more of its subnormal operands are flushed to zero. If the result is a subnormal before rounding, it may be flushed to zero

In each of the cases, if an operand or result is flushed to zero, the sign of the zero is undefined.

Texture Addressing and Conversion Rules

The texture coordinates specified to the `sample`, `sample_compare`, `gather`, `gather_compare`, `read`, and `write` functions cannot be INF or NaN. In addition, the texture coordinate must refer to a region inside the texture for the texture `read` and `write` functions.

The following sections discuss conversion rules that are applied when reading and writing textures in a graphics or kernel function.

Conversion rules for normalized integer pixel data types

This section discusses converting normalized integer pixel data types to floating-point values and vice-versa.

Converting normalized integer pixel data types to floating-point values

For textures that have 8-bit, 10-bit or 16-bit normalized unsigned integer pixel values, the texture sample and read functions convert the pixel values from an 8-bit or 16-bit unsigned integer to a normalized single or half-precision floating-point value in the range [0.0, 1.0].

For textures that have 8-bit or 16-bit normalized signed integer pixel values, the texture sample and read functions convert the pixel values from an 8-bit or 16-bit signed integer to a normalized single or half-precision floating-point value in the range [-1.0, 1.0].

These conversions are performed as listed in the second column of Table 7-3 (page 100). The precision of the conversion rules are guaranteed to be <= 1.5 ulp except for the cases described in the third column.

Table 7-3 Rules for Conversion from a Normalized Integer to a Normalized Floating-Point Value

Convert from	Conversion Rule to Normalized Float	Corner Cases
8-bit normalized unsigned integer	`float(c) / 255.0`	0 must convert to 0.0 255 must convert to 1.0
10-bit normalized unsigned integer	`float(c) / 1023.0`	0 must convert to 0.0 1023 must convert to 1.0
16-bit normalized unsigned integer	`float(c) / 65535.0`	0 must convert to 0.0 65535 must convert to 1.0

Convert from	Conversion Rule to Normalized Float	Corner Cases
8-bit normalized signed integer	`max(-1.0, float(c)/127.0)`	-128 and -127 must convert to -1.0 0 must convert to 0.0 127 must convert to 1.0
16-bit normalized signed integer	`max(-1.0, float(c)/32767.0)`	-32768 and -32767 must convert to -1.0 0 must convert to 0.0 32767 must convert to 1.0

Converting floating-point values to normalized integer pixel data types

For textures that have 8-bit, 10-bit or 16-bit normalized unsigned integer pixel values, the texture write functions convert the single or half-precision floating-point pixel value to an 8-bit or 16-bit unsigned integer.

For textures that have 8-bit or 16-bit normalized signed integer pixel values, the texture write functions convert the single or half-precision floating-point pixel value to an 8-bit or 16-bit signed integer.

The preferred methods to perform conversions from floating-point values to normalized integer values are listed in Table 7-4 (page 101).

Table 7-4 Rules for Conversion from a Floating-Point to a Normalized Integer Value

Convert to	Conversion Rule to Normalized Integer
8-bit normalized unsigned integer	`float(c) / 255.0`
10-bit normalized unsigned integer	`float(c) / 1023.0`
16-bit normalized unsigned integer	`float(c) / 65535.0`
8-bit normalized signed integer	`max(-1.0, float(c)/127.0)`
16-bit normalized signed integer	`max(-1.0, float(c)/32767.0)`

The GPU may choose to approximate the rounding mode used in the conversions described in Table 7-4 (page 101). If a rounding mode other than round to nearest even is used, the absolute error of the implementation dependent rounding mode vs. the result produced by the round to nearest even rounding mode must be <= 0.6.

Conversion rules for half precision floating-point pixel data type

For textures that have half-precision floating-point pixel color values, the conversions from `half` to `float` are lossless. 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 may be converted to an appropriate NaN or be flushed to zero in the `half` type. A `float` INF may be converted to an appropriate INF or be flushed to zero in the `half` type.

Conversion rules for floating-point channel data type

The following rules apply for reading and writing textures that have single-precision floating-point pixel color values.

- NaNs may be converted to a NaN value(s) or be flushed to zero.
- INFs may be converted to a INF value(s) or be flushed to zero.
- Denorms may be flushed to zero.
- All other values must be preserved.

Conversion rules for signed and unsigned integer pixel data types

For textures that have 8-bit or 16-bit signed or unsigned integer pixel values, the texture sample and read functions return a signed or unsigned 32-bit integer pixel value. The conversions described in this section must be correctly saturated.

Writes to these integer textures perform one of the conversions listed in Table 7-5 (page 102).

Table 7-5 Rules for Conversion between Integer Pixel Data Types

Convert from	Convert to	Conversion Rule
32-bit signed integer	8-bit signed integer	`result = convert_char_saturate(val)`
32-bit signed integer	16-bit signed integer	`result = convert_short_saturate(val)`
32-bit unsigned integer	8-bit unsigned integer	`result = convert_uchar_saturate(val)`
32-bit unsigned integer	16-bit unsigned integer	`result = convert_ushort_saturate(val)`

Conversion rules for sRGBA and sBGRA Textures

Conversion from sRGB space to linear space is automatically done when sampling from an sRGB texture. The conversion from sRGB to linear RGB is performed before the filter specified in the sampler specified when sampling the texture is applied. If the texture has an alpha channel, the alpha data is stored in linear color space.

Conversion from linear to sRGB space is automatically done when writing to an sRGB texture. If the texture has an alpha channel, the alpha data is stored in linear color space.

The following is the conversion rule for converting a normalized 8-bit unsigned integer sRGB color value to a floating-point linear RGB color value (call it `c`) as per rules described in Converting normalized integer pixel data types to floating-point values (page 100).

```
if (c <= 0.04045),
    result = c / 12.92;
else
    result = powr((c + 0.055) / 1.055, 2.4);
```

The resulting floating point value, if converted back to an sRGB value without rounding to a 8-bit unsigned integer value, must be within 0.5 ulp of the original sRGB value.

The following are the conversion rules for converting a linear RGB floating-point color value (call it `c`) to a normalized 8-bit unsigned integer sRGB value.

```
if (isnan(c))
    c = 0.0;
if (c > 1.0)
    c = 1.0;
else if (c < 0.0)
    c = 0.0;
else if (c < 0.0031308)
    c = 12.92 * c;
else
    c = 1.055 * powr(c, 1.0/2.4) - 0.055;

convert to integer scale i.e. c = c * 255.0
convert to integer:
    c = c + 0.5
     drop the decimal fraction, and the remaining
     floating-point(integral) value is converted
        directly to an integer.
```

The precision of the above conversion should be such that:

```
fabs(reference result – integer result) <= 0.6
```

Document Revision History

This table describes the changes to *Metal Shading Language Guide* .

Date	Notes
2014-09-17	New document that describes the programming language used to create graphics and compute functions to use with the Metal framework.