SYCL Reference

Sep 04, 2020

CONTENTS

CHAPTER

ONE

INTRODUCTION

CHAPTER

TWO

INTERFACE

For further details on SYCL, see the SYCL Specification.

Tip: If you are unfamiliar with C++ templates and lambda functions, consult a C++ language references to gain a basic understanding before continuing.

2.1 Header File

A single header file must be included:

```
#include "CL/sycl.hpp"
```

2.2 Namespaces

Unless otherwise noted, all symbols should be prefixed with the `sycl` namespace. `buffer` is `sycl::buffer`, and `info::device::name` is `sycl::info::device::name`.

2.3 Common Interface

In this section, we define methods that are common to multiple classes.

2.3.1 By-value Semantics

Types: id, range, item, nd_item, h_item, group and nd_range.

Classes with reference semantics support the following methods.

```
class T {
  T(const T &rhs);
  T(T &&rhs);
  T &operator=(const T &rhs);
  T &operator=(T &&rhs);
  ~T();
  friend bool operator==(const T &lhs, const T &rhs) { /* ... */ }
  friend bool operator!=(const T &lhs, const T &rhs) { /* ... */ }
};
```

2.3.2 Reference Semantics

Classes: device, context, queue, program, kernel, event, buffer, image, sampler, accessor and stream

Classes with reference semantics support the following methods. An instance that is constructed as a copy of another instance must behave as-if it were the same instance.

```
class T {
public:
  T(const T &rhs);
  T(T &&rhs);
  T &operator=(const T &rhs);
  T &operator=(T &&rhs);
  ~T();
  friend bool operator==(const T &lhs, const T &rhs) { /* ... */ }
  friend bool operator!=(const T &lhs, const T &rhs) { /* ... */ }
};
```

2.3.3 property_list

```
class property_list;
```

Member and nonmember functions

property_list

```
template <typename... propertyTN>
property_list(propertyTN... props);
```

2.3.4 param_traits

```
template <typename T, T param>
class param_traits;
```

Namespace

```
info
```

Member types

return_type	

2.4 Runtime Classes

2.4.1 Device selectors

Devices selectors allow the SYCL runtime to choose the device.

A device selector can be passed to *queue*, *platform*, and other constructors to control the selection of a device. A program may use *Built-in Device Selectors* or define its own *device_selector* for full control.

device_selector

```
class device_selector;
```

Abstract class for device selectors.

This is the base class for the *Built-in Device Selectors*. To define a custom device selector, create a derived class that defines the () operator.

Member and nonmember functions

(constructors)

```
device_selector();
device_selector(const device_selector &rhs);
```

Construct a device_selector.

A device selector can be created from another by passing `rhs`.

select_device

```
device select_device() const;
```

Returns the device with the highest score as determined by calling *operator()*.

Exceptions

Throws a runtime error if all devices have a negative score.

operator=

```
device_selector &operator=(const device_selector &rhs);
```

Create a device selector by copying another one.

operator()

```
virtual int operator()(const device &device) const = 0;
```

Scoring function for devices.

All derived device selectors must define this operator. *select_device* calls this operator for every device, and selects the device with highest score. Return a negative score if a device should not be selected.

Built-in Device Selectors

SYCL provides built-in device selectors for convenience. They use *device_selector* as a base class.

default_selector	Selects device according to implementation-defined heuristic or host device if no device can be found.
gpu_selector	Select a GPU
accelerator_selector	Select an accelerator
cpu_selector	Select a CPU device
host_selector	Select the host device

Create a device selector by copying another one.

See also:

SYCL Specification Section 4.6.1.1

Example

```
#include <CL/sycl.hpp>

using namespace sycl;

int main() {
  device d;

  try {
    d = device(gpu_selector());
  } catch (exception const& e) {
    std::cout << "Cannot select a GPU\n" << e.what() << "\n";
    std::cout << "Using a CPU device\n";
    d = device(cpu_selector());
  }

  std::cout << "Using " << d.get_info<sycl::info::device::name>();
}
```

Output on a system without a GPU:

```
Cannot select a GPU
No device of requested type available. Please check https://software.intel.com/en-us/
↪articles/intel-oneapi-dpcpp-compiler-system-requirements-beta -1 (CL_DEVICE_NOT_
↪FOUND)
```

(continues on next page)

(continued from previous page)

```
Using a CPU device
Using Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz
```

2.4.2 Platforms

platform

```
class platform;
```

Abstraction for SYCL platform.

A platform contains 1 or more SYCL devices, or a host device.

See also:

SYCL Specification Section 4.6.2

Member and nonmember functions

Example

Enumerate the platforms and the devices they contain.

```
#include <CL/sycl.hpp>

int main() {
  auto platforms = sycl::platform::get_platforms();

  for (auto &platform : platforms) {
    std::cout << "Platform: "
              << platform.get_info<sycl::info::platform::name>()
              << std::endl;

    auto devices = platform.get_devices();
    for (auto &device : devices ) {
      std::cout << "  Device: "
                << device.get_info<sycl::info::device::name>()
                << std::endl;
    }
  }

  return 0;
}
```

Output:

```
Platform: Intel(R) FPGA Emulation Platform for OpenCL(TM)
  Device: Intel(R) FPGA Emulation Device
Platform: Intel(R) OpenCL
  Device: Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz
Platform: Intel(R) CPU Runtime for OpenCL(TM) Applications
  Device: Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz
Platform: SYCL host platform
  Device: SYCL host device
```

(constructors)

```
platform();
explicit platform(cl_platform_id platformID);
explicit platform(const device_selector &deviceSelector);
```

Construct a SYCL platform instance.

The default constructor creates a host platform. When passed a `cl_platform_id`, an OpenCL|trade| platform is used to construct the platform. The `cl_platform_id` is retained and available via *get*. When passed a *device_selector*, a platform is constructed that includes the preferred device.

get

```
cl_platform_id get() const;
```

Returns the OpenCL device associated with the platform.

Only call this when the platform constructor was passed a `cl_platform_id`.

get_devices

```
vector_class<device> get_devices(
   info::device_type = info::device_type::all) const;
```

Returns vector of SYCL devices associated with the platform and filtered by *device_type*

Example

See *platform-example*.

get_info

```
template< info::platform param >
typename info::param_traits<info::platform, param>::return_type get_info() const;
```

Returns information about the platform as determined by `param`.

See *Platform Info* for details.

Example

See *platform-example*.

has_extension

```
bool has_extension(const string_class &extension) const;
```

Returns True if the platform has `extension`.

is_host

```
bool is_host() const;
```

Returns True if the platform contains a SYCL host device

get_platforms

```
static vector_class<platform> get_platforms();
```

Returns a vector_class containing SYCL platforms bound to the system.

Example

See *platform-example*.

Platform Info

```
enum class platform : unsigned int {
  profile,
  version,
  name,
  vendor,
  extensions
};
```

Namespace

```
info
```

Used as a template parameter for *get_info* to determine the type of information.

Descriptor	Return type	Description
profile	string_class	OpenCL profile
version	string_class	OpenCL software driver version
name	string_class	Device name of the platform
vendor	string_class	Vendor name
extensions	vector_class<string_class>	Extension names supported by the platform

2.4.3 Contexts

context

```
class context;
```

A context encapsulates a single SYCL platform and a collection of SYCL devices associated with the platform.

A context may include a subset of the devices provided by the platform. The same platform may be associated with more than one context, but a device can only be part of a single context.

See also:

SYCL Specification Section 4.6.3

Member and nonmember functions

(constructors)

```
explicit context(const property_list &propList = {});
context(async_handler asyncHandler,
        const property_list &propList = {});
context(const device &dev, const property_list &propList = {});
context(const device &dev, async_handler asyncHandler,
        const property_list &propList = {});
context(const platform &plt, const property_list &propList = {});
context(const platform &plt, async_handler asyncHandler,
        const property_list &propList = {});
context(const vector_class<device> &deviceList,
        const property_list &propList = {});
context(const vector_class<device> &deviceList,
        async_handler asyncHandler, const property_list &propList = {});
context(cl_context clContext, async_handler asyncHandler = {});
```

Construct a context.

The parameters to the constructor allow control of the devices and platforms associated with the context. The constructor uses the *default selector* when no platforms or devices are supplied.

Parameters

propList See *Context Properties*.

asyncHandler Called to report asynchronous SYCL exceptions for this context

dev Constructed context contains device

deviceList Constructed context contains devices

plt Constructed context contains platform

clContext Constructed context contains cl_context

Constructs a context

get

```
cl_context get() const;
```

Returns cl_context that was passed in constructor.

is_host

```
bool is_host() const;
```

Returns True if this context is a host context.

get_platform

```
platform get_platform() const;
```

Return platform associated with this context.

get_devices

```
vector_class<device> get_devices() const;
```

Returns vector of devices associated with this context.

get_info

```
template <info::context param>
typename info::param_traits<info::context, param>::return_type get_info() const;
```

Returns information about the context as determined by `param`. See *Context Info* for details.

has_property

```
template <typename propertyT>
bool has_property() const;
```

Template parameters

propertyT	

Returns True if the property type was passed to the constructor.

get_property

```
template <typename propertyT>
propertyT get_property() const;
```

Template parameters

propertyT	

Returns copy of property of passed to the constructor.

Context Info

```
enum class context : int {
  reference_count,
  platform,
  devices
};
```

Namespace

```
info
```

Used as a template parameter for *get_info* to determine the type of information.

Descriptor	Return type	Description
reference_count	cl_uint	Reference count of the underlying cl_context
platform	platform	SYCL platform for the context
devices	vector_class<device>	SYCL devices associated with this platform

Context Properties

SYCL does not define any properties for *context*.

2.4.4 Devices

device

```
class device;
```

An abstract class representing various models of SYCL devices. A device could be a GPU, CPU, or other type of accelerator. Devices execute kernel functions.

See also:

SYCL Specification Section 4.6.4

Member and nonmember functions

(constructors)

```
device();
explicit device(cl_device_id deviceId);
explicit device(const device_selector &deviceSelector);
```

Construct a device.

The default constructor creates a host device. A device can also be constructed from an OpenCL|tradel device or may be chosen by a *Device selectors*.

Parameters

deviceID	OpenCL device id
deviceSelector	Device selector

get

```
cl_device_id get() const;
```

Return the cl_device_id of the underlying OpenCL platform.

is_host

```
bool is_host() const;
```

Returns True if the device is a host device, False otherwise.

is_cpu

```
bool is_cpu() const;
```

Returns True if the device is a CPU, False otherwise.

is_gpu

```
bool is_gpu() const;
```

Returns True if the device is a GPU, False otherwise.

is_accelerator

```
bool is_accelerator() const;
```

Returns True if the device is an accelerator, False otherwise.

get_platform

```
platform get_platform() const;
```

Returns the platform that contains the device.

get_info

```
template <info::device param>
typename info::param_traits<info::device, param>::return_type
get_info() const;
```

Returns information about the device as determined by `param`. See *Device Info* for details.

Example

See *Example*.

has_extension

```
bool has_extension(const string_class &extension) const;
```

Returns True if device supports the extension.

create_sub_devices

```
Available only when:
 prop == info::partition_property::partition_equally

template <info::partition_property prop>
vector_class<device> create_sub_devices(size_t nbSubDev) const;

Available only when:
 prop == info::partition_property::partition_by_counts

template <info::partition_property prop>
vector_class<device> create_sub_devices(const vector_class<size_t> &counts)␣
↪const;

Available only when:
 prop == info::partition_property::partition_by_affinity_domain
```

```
template <info::partition_property prop>
vector_class<device> create_sub_devices(info::affinity_domain affinityDomain)␣
↪const;
```

Divide into sub-devices, according to the requested partition property.

Template parameters

prop	See *partition_property*

Parameters

nbSubDev	Number of subdevices
counts	Vector of sizes for the subdevices
affinityDomain	See *partition_affinity_domain*

Exceptions

feature_not_supported When device does not support the *partition_property* specified by the `prop` template argument.

get_devices

```
static vector_class<device> get_devices(
    info::device_type deviceType = info::device_type::all);
```

Returns vector of devices filtered by *device_type*.

Device Info

device

```
enum class device : int {
  device_type,
  vendor_id,
  max_compute_units,
  max_work_item_dimensions,
  max_work_item_sizes,
  max_work_group_size,
  preferred_vector_width_char,
  preferred_vector_width_short,
  preferred_vector_width_int,
  preferred_vector_width_long,
  preferred_vector_width_float,
  preferred_vector_width_double,
  preferred_vector_width_half,
  native_vector_width_char,
  native_vector_width_short,
```

(continues on next page)

(continued from previous page)

```
  native_vector_width_int,
  native_vector_width_long,
  native_vector_width_float,
  native_vector_width_double,
  native_vector_width_half,
  max_clock_frequency,
  address_bits,
  max_mem_alloc_size,
  image_support,
  max_read_image_args,
  max_write_image_args,
  image2d_max_height,
  image2d_max_width,
  image3d_max_height,
  image3d_max_width,
  image3d_max_depth,
  image_max_buffer_size,
  image_max_array_size,
  max_samplers,
  max_parameter_size,
  mem_base_addr_align,
  half_fp_config,
  single_fp_config,
  double_fp_config,
  global_mem_cache_type,
  global_mem_cache_line_size,
  global_mem_cache_size,
  global_mem_size,
  max_constant_buffer_size,
  max_constant_args,
  local_mem_type,
  local_mem_size,
  error_correction_support,
  host_unified_memory,
  profiling_timer_resolution,
  is_endian_little,
  is_available,
  is_compiler_available,
  is_linker_available,
  execution_capabilities,
  queue_profiling,
  built_in_kernels,
  platform,
  name,
  vendor,
  driver_version,
  profile,
  version,
  opencl_c_version,
  extensions,
  printf_buffer_size,
  preferred_interop_user_sync,
  parent_device,
  partition_max_sub_devices,
  partition_properties,
  partition_affinity_domains,
  partition_type_property,
```

(continues on next page)

(continued from previous page)

```
  partition_type_affinity_domain,
  reference_count
}
```

Namespace

```
info
```

Used as a template parameter for *get_info* to determine the type of information.

Descriptor	Return type	Description
device_type		
vendor_id		
max_compute_units		
max_work_item_dimensions		
max_work_item_sizes		
max_work_group_size		
preferred_vector_width_char		
preferred_vector_width_short		
preferred_vector_width_int		
preferred_vector_width_long		
preferred_vector_width_float		
preferred_vector_width_double		
preferred_vector_width_half		
native_vector_width_char		
native_vector_width_short		
native_vector_width_int		
native_vector_width_long		
native_vector_width_float		
native_vector_width_double		
native_vector_width_half		
max_clock_frequency		
address_bits		
max_mem_alloc_size		
image_support		
max_read_image_args		
max_write_image_args		
image2d_max_height		
image2d_max_width		
image3d_max_height		
image3d_max_width		
image3d_max_depth		
image_max_buffer_size		
image_max_array_size		
max_samplers		
max_parameter_size		
mem_base_addr_align		
half_fp_config	*fp_config*	
single_fp_config	*fp_config*	

continues on next page

Table 1 – continued from previous page

Descriptor	Return type	Description
double_fp_config	*fp_config*	
global_mem_cache_type	*global_mem_cache_type*	
global_mem_cache_line_size		
global_mem_cache_size		
global_mem_size		
max_constant_buffer_size		
max_constant_args		
local_mem_type	*local_mem_type*	
local_mem_size		
error_correction_support		
host_unified_memory		
profiling_timer_resolution		
is_endian_little		
is_available		
is_compiler_available		
is_linker_available		
execution_capabilities	*execution_capability*	
queue_profiling		
built_in_kernels		
platform		
name		
vendor		
driver_version		
profile		
version		
opencl_c_version		
extensions		
printf_buffer_size		
preferred_interop_user_sync		
parent_device		
partition_max_sub_devices		
partition_properties		
partition_affinity_domains		
partition_type_property		
partition_type_affinity_domain		
reference_count		

device_type

```
enum class device_type : unsigned int {
  cpu,          // Maps to OpenCL CL_DEVICE_TYPE_CPU
  gpu,          // Maps to OpenCL CL_DEVICE_TYPE_GPU
  accelerator,  // Maps to OpenCL CL_DEVICE_TYPE_ACCELERATOR
  custom,       // Maps to OpenCL CL_DEVICE_TYPE_CUSTOM
  automatic,    // Maps to OpenCL CL_DEVICE_TYPE_DEFAULT
  host,
  all           // Maps to OpenCL CL_DEVICE_TYPE_ALL
};
```

See platform *get_devices* and device *get_devices*.

partition_property

```
enum class partition_property : int {
  no_partition,
  partition_equally,
  partition_by_counts,
  partition_by_affinity_domain
};
```

See *create_sub_devices*

partition_affinity_domain

```
enum class partition_affinity_domain : int {
  not_applicable,
  numa,
  L4_cache,
  L3_cache,
  L2_cache,
  L1_cache,
  next_partitionable
};
```

See *create_sub_devices*

local_mem_type

```
enum class local_mem_type : int { none, local, global };
```

See *get_info*

fp_config

```
enum class fp_config : int {
  denorm,
  inf_nan,
  round_to_nearest,
  round_to_zero,
  round_to_inf,
  fma,
  correctly_rounded_divide_sqrt,
  soft_float
};
```

See *get_info*

global_mem_cache_type

```
enum class global_mem_cache_type : int { none, read_only, read_write };
```

See *get_info*

execution_capability

```
enum class execution_capability : unsigned int {
  exec_kernel,
  exec_native_kernel
};
```

See *get_info*

2.4.5 Queues

queue

```
class queue;
```

Queues connect a host program to a single device. Programs submit tasks to a device via the queue and may monitor the queue for completion. A program initiates the task by submitting a *Command group function object* to a queue. The command group defines a kernel function, the prerequisites to execute the kernel function, and an invocation of the kernel function on an index space. After submitting the command group, a program may use the queue to monitor the completion of the task for completion and errors.

See also:

SYCL Specification Section 4.6.5

Member and nonmember functions

(constructors)

```
explicit queue(const property_list &propList = {});
explicit queue(const async_handler &asyncHandler,
               const property_list &propList = {});
explicit queue(const device_selector &deviceSelector,
               const property_list &propList = {});
explicit queue(const device_selector &deviceSelector,
               const async_handler &asyncHandler,
               const property_list &propList = {});
explicit queue(const device &syclDevice, const property_list &propList = {});
explicit queue(const device &syclDevice, const async_handler &asyncHandler,
               const property_list &propList = {});
explicit queue(const context &syclContext,
               const device_selector &deviceSelector,
               const property_list &propList = {});
explicit queue(const context &syclContext,
               const device_selector &deviceSelector,
```

(continues on next page)

(continued from previous page)

```
              const async_handler &asyncHandler,
              const property_list &propList = {});
explicit queue(const context &syclContext,
              const device &syclDevice,
              const property_list &propList = {});
explicit queue(const context &syclContext, const device &syclDevice,
              const async_handler &asyncHandler,
              const property_list &propList = {});
explicit queue(cl_command_queue clQueue, const context& syclContext,
              const async_handler &asyncHandler = {});
```

Construct a queue.

Constructing a queue selects the device attached to the queue. The program may control the device by passing a cl_command_queue, *device*, or a *device_selector*. If none are provided, the constructor uses the *default_selector* to select a device. The constructor implicitly creates the *context*, *platform*, and *device* as needed.

The SYCL runtime executes the tasks asynchronously. Programs may catch asynchronous errors that occur during execution by constructing the queue with an `asyncHandler` and calling *wait_and_throw*.

Parameters

propList	See *Queue Properties*
asyncHandler	Called for asynchronous exceptions, see *async_handler*
deviceSelector	Selects device for queue
syclDevice	Device for queue
syclContext	Associate queue with the context
clQueue	Assocate queue with OpenCLltradel queue

Exceptions

invalid_object_error If `syclContext` does not encapsulate `syclDevice`.

get

```
cl_command_queue get() const;
```

Return OpenCL queue associated with SYCL queue.

get_context

```
context get_context() const;
```

Returns context associated with queue.

get_device

```
device get_device() const;
```

Returns device associated with queue.

is_host

```
bool is_host() const;
```

Returns True if queue executes on host device.

get_info

```
template <info::queue param>
typename info::param_traits<info::queue, param>::return_type get_info() const;
```

Returns information about the queue as determined by `param`. See *queue* for details.

submit

```
template <typename T>
event submit(T cgf);
template <typename T>
event submit(T cgf, const queue &secondaryQueue);
```

Template parameters

T	

Parameters

cgf	Command group function object
secondaryQueue	On error, runtime resubmits command group to the secondary queue.

Submit a command group function object to the queue for asynchronous execution.

Returns an *event*, which may be used for synchronizing enqueued tasks. See *Command group function object* for more information on the `cgf` parameter.

In most cases, the `T` template parameter is not provided because it is inferred from the type of `cgf`.

Exceptions

The runtime resubmits the command group to the secondary queue if an error occurs executing on the primary queue.

wait

```
void wait();
```

Wait for all enqueued tasks to complete.

wait_and_throw

```
void wait_and_throw();
```

Wait for all enqueued tasks and pass asynchronous errors to handler provided in *(constructors)*.

throw_asynchronous

```
void throw_asynchronous();
```

Passes any asynchronous errors to handler provided in *(constructors)*.

memcpy

```
event memcpy(void* dest, const void* src, size_t num_bytes);
```

Set memory allocated with *malloc_device*.

memset

```
event memset(void* ptr, int value, size_t num_bytes);
```

Set memory allocated with *malloc_device*.

fill

```
template <typename T>
event fill(void* ptr, const T& pattern, size_t count);
```

Set memory allocated with *malloc_device*.

Queue Info

```
enum class queue : int {
  context,
  device,
  reference_count,
};
```

Namespace

```
info
```

Used as a template parameter for *get_info* to determine the type of information.

Descriptor	Return type	Description
context	context	SYCL context associated with the queue
device	device	SYCL device associated with the queue
reference_count	cl_uint	Reference count of the queue

Queue Properties

Namespace

```
property::queue
```

Queue properties are specified in the queue constructor.

enable_profiling SYCL runtime captures profiling information for command groups submitted to the queue.

2.4.6 Events

event

```
class event;
```

Events support the explicit control of scheduling of kernels, and querying status of a running kernel. Operations like *submit* that queue a kernel for execution may accept an event to wait on and return an event associated with the queued kernel.

See also:

SYCL Specification Section 4.6.6

Member and nonmember functions

(constructors)

```
event();
event(cl_event clEvent, const context& syclContext);
```

Construct an event.

cl_event_get

```
cl_event get();
```

Returns OpenCLltradel event associated with this event.

is_host

```
bool is_host() const;
```

Returns True if this a host event

get_wait_list

```
vector_class<event> get_wait_list();
```

Returns vector of events that this events waits on.

wait

```
void wait();
```

Wait for the associated command to complete.

wait

```
static void wait(const vector_class<event> &eventList);
```

Wait for vector of events to complete.

wait_and_throw

```
void wait_and_throw();
```

Wait for an event to complete, and pass asynchronous errors to handler associated with the command.

wait_and_throw

```
static void wait_and_throw(const vector_class<event> &eventList);
```

Wait for a vector of events to complete, and pass asynchronous errors to handlers associated with the commands.

get_info

```
template <info::event param>
typename info::param_traits<info::event, param>::return_type get_info() const;
```

Returns information about the queue as determined by `param`. See *Event Info* for details.

get_profiling_info

```
template <info::event_profiling param>
typename info::param_traits<info::event_profiling, param>::return_type get_profiling_
↪info() const;
```

Returns information about the queue as determined by `param`. See *Event profiling info* for details.

Event info

```
enum class event: int {
  command_execution_status,
  reference_count
};
```

Namespace

```
info
```

Used as a template parameter for *get_info* to determine the type of information.

Descriptor	Return type	Description
command_execution_status	info::event_command_status	See *event_command_status*
reference_count	cl_uint	Reference count of the event

event_command_status

```
enum class event_command_status : int {
  submitted,
  running,
  complete
};
```

Event profiling info

```
enum class event_profiling : int {
  command_submit,
  command_start,
  command_end
};
```

Namespace

```
info
```

Used as a template parameter for *get_profiling_info* to determine the type of information.

Descriptor	Return type	Description
command_submit	cl_ulong	Time in nanoseconds when *command_group* was submitted
command_start	cl_ulong	Time in nanoseconds when *command_group* started execution
command_end	cl_ulong	Time in nanoseconds when *command_group* finished execution

2.5 Data access

2.5.1 Buffers

buffer

```
template <typename T, int dimensions = 1,
          typename AllocatorT = cl::sycl::buffer_allocator>
class buffer;
```

Template parameters

T	Type of data in buffer
dimensions	Dimensionality of data: 1, 2, or 3
AllocatorT	Allocator for buffer data

Buffers are containers for data that can be read/written by both kernel and host. Data in a buffer cannot be directly via pointers. Instead, a program creates an *Buffer accessor* that references the buffer. The accessor provides array-like interfaces to read/write actual data. Accessors indicate when they read or write data. When a program creates an

accessor for a buffer, the SYCL runtime copies the data to where it is needed, either the host or the device. If the accessor is part of a device command group, then the runtime delays execution of the kernel until the data movement is complete. If the host creates an accessor, it will pause until the data is available on the host. As a result data and kernels can execute asynchronously and in parallel, only requiring the program to specify the data dependencies.

Initialization

Buffers can be automatically initialized via host data, iterator, or as a slice of another buffer. The constructor determines the initialization method.

Write back

The destructor for a buffer can optionally write the data back to host memory, either by pointer or iterator. *set_final_data* and *set_write_back* control the write back of data.

Memory allocation

The SYCL runtimes uses the default allocator for buffer memory allocation, unless the constructor provides an allocator.

Member types

value_type	type of buffer element
reference	reference type of buffer element
const_reference	const reference type of buffer element
allocator_type	type of allocator for buffer data

See also:

SYCL Specification Section 4.7.2

Member and nonmember functions

(constructors)

```
buffer(const range<dimensions> &bufferRange,
       const property_list &propList = {});
buffer(const range<dimensions> &bufferRange, AllocatorT allocator,
       const property_list &propList = {});
buffer(T hostData, const range<dimensions> &bufferRange,
       const property_list &propList = {});
buffer(T *hostData, const range<dimensions> &bufferRange,
       AllocatorT allocator, const property_list &propList = {});
buffer(const T *hostData, const range<dimensions> &bufferRange,
       const property_list &propList = {});
buffer(const T *hostData, const range<dimensions> &bufferRange,
       AllocatorT allocator, const property_list &propList = {});
buffer(const shared_ptr_class<T> &hostData,
       const range<dimensions> &bufferRange, AllocatorT allocator,
       const property_list &propList = {});
buffer(const shared_ptr_class<T> &hostData,
       const range<dimensions> &bufferRange,
       const property_list &propList = {});
buffer(buffer<T, dimensions, AllocatorT> b, const id<dimensions> &baseIndex,
       const range<dimensions> &subRange);
```

```
*Available only when:
 dimensions == 1

template <class InputIterator>
buffer<T, 1>(InputIterator first, InputIterator last, AllocatorT allocator,
             const property_list &propList = {});
template <class InputIterator>
buffer<T, 1>(InputIterator first, InputIterator last,
             const property_list &propList = {});
buffer(cl_mem clMemObject, const context &syclContext,
       event availableEvent = {});
```

Construct a buffer.

Buffers can be initialized by a host data pointer. While the buffer exists, it *owns* the host data and direct access of the host data pointer during that time is undefined. The SYCL runtime performs a write back of the buffer data back to the host data pointer when the buffer is destroyed. Buffers can also be initialized as a slice of another buffer, by specifying the origin of the data and the dimensions.

A constructor can also accept cl_mem or iterators to initialize a buffer.

Template parameters

InputIterator	type of iterator used to initialize the buffer

Parameters

bufferRange	*range* specifies the dimensions of the buffer
allocator	Allocator for buffer data
propList	See *Buffer properties*
hostData	Pointer to host memory to hold data
first	Iterator to initialize buffer
last	Iterator to initialize buffer
b	Buffer used to initialize this buffer
baseIndx	Origin of sub-buffer
subRange	Dimensions of sub-buffer

get_range

```
range<dimensions> get_range() const;
```

Returns the dimensions of the buffer.

get_count

```
size_t get_count() const;
```

Returns the total number of elements in the buffer.

get_size

```
size_t get_size() const;
```

Returns the size of the buffer storage in bytes.

get_allocator

```
AllocatorT get_allocator() const;
```

Returns the allocator provided to the buffer.

get_access

```
template <access::mode mode, access::target target = access::target::global_buffer>
accessor<T, dimensions, mode, target> get_access(
    handler &commandGroupHandler);
template <access::mode mode>
accessor<T, dimensions, mode, access::target::host_buffer> get_access();
template <access::mode mode, access::target target = access::target::global_buffer>
accessor<T, dimensions, mode, target> get_access(
    handler &commandGroupHandler, range<dimensions> accessRange,
    id<dimensions> accessOffset = {});
template <access::mode mode>
accessor<T, dimensions, mode, access::target::host_buffer> get_access(
  range<dimensions> accessRange, id<dimensions> accessOffset = {});
```

Returns a accessor to the buffer.

Template parameters

mode	See *mode*
target	See *target*

Parameters

commandGroupHandler	Command group that uses the accessor
accessRange	Dimensions of the sub-buffer that is accessed
accessOffset	Origin of the sub-buffer that is accessed

set_final_data

```
template <typename Destination = std::nullptr_t>
void set_final_data(Destination finalData = nullptr);
```

Template parameters

Destination	std::weak_ptr<T> or output iterator

Parameters

finalData	Indicates where data is copied at destruction time

Set the final data location. Final data controls the location for write back when the buffer is destroyed.

set_write_back

```
void set_write_back(bool flag = true);
```

Parameters

flag	True to force write back

Set the write back.

is_sub_buffer

```
bool is_sub_buffer() const;
```

Returns True if this is a sub-buffer.

reinterpret

```
template <typename ReinterpretT, int ReinterpretDim>
buffer<ReinterpretT, ReinterpretDim, AllocatorT>
reinterpret(range<ReinterpretDim> reinterpretRange) const;
```

Template parameters

ReinterpretT	Type of new buffer element
ReinterpretDim	Dimensions of new buffer

Parameters

ReinterpretRange	Dimensionality of new buffer

Creates a new buffer with the requested element type and dimensionality, containing the data of the passed buffer or sub-buffer.

Exceptions

errc::invalid_object_error Size in bytes of new buffer does not match original buffer.

Buffer properties

use_host_ptr

```
class use_host_ptr;
```

Namespace

```
property::buffer
```

Use the provided host pointer and do not allocate new data on the host.

Member and nonmember functions

(constructors)

```
use_host_ptr();
```

use_mutex

```
class use_mutex;
```

Namespace

```
property::buffer
```

Adds the requirement that the memory owned by the SYCL buffer can be shared with the application via a std::mutex provided to the property.

Member and nonmember functions

(constructors)

```
use_mutex();
```

get_mutex_ptr

```
mutex_class *get_mutex_ptr() const;
```

context_bound

```
context_bound;
```

Namespace

```
property::buffer
```

The buffer can only be associated with a single SYCL context provided to the property.

Member and nonmember functions

(constructors)

```
use_mutex();
```

get_context

```
context get_context() const;
```

2.5.2 Images

image

```
template <int dimensions = 1,
          typename AllocatorT = cl::sycl::image_allocator>
class image;
```

Template parameters

dimensions	
AllocatorT	

See also:

SYCL Specification Section 4.7.3

Member and nonmember functions

(constructors)

```
image(image_channel_order order, image_channel_type type,
      const range<dimensions> &range, const property_list &propList = {});
image(image_channel_order order, image_channel_type type,
      const range<dimensions> &range, AllocatorT allocator,
      const property_list &propList = {});
image(void hostPointer, image_channel_order order,
      image_channel_type type, const range<dimensions> &range,
      const property_list &propList = {});
image(void *hostPointer, image_channel_order order,
      image_channel_type type, const range<dimensions> &range,
      AllocatorT allocator, const property_list &propList = {});
image(const void *hostPointer, image_channel_order order,
      image_channel_type type, const range<dimensions> &range,
      const property_list &propList = {});
image(const void *hostPointer, image_channel_order order,
      image_channel_type type, const range<dimensions> &range,
      AllocatorT allocator, const property_list &propList = {});
image(shared_ptr_class<void> &hostPointer, image_channel_order order,
      image_channel_type type, const range<dimensions> &range,
      const property_list &propList = {});
image(shared_ptr_class<void> &hostPointer, image_channel_order order,
      image_channel_type type, const range<dimensions> &range,
      AllocatorT allocator, const property_list &propList = {});
image(cl_mem clMemObject, const context &syclContext,
```

```
      event availableEvent = {});

*Available only when:
 dimensions > 1

image(image_channel_order order, image_channel_type type,
      const range<dimensions> &range, const range<dimensions - 1> &pitch,
      const property_list &propList = {});
image(image_channel_order order, image_channel_type type,
      const range<dimensions> &range, const range<dimensions - 1> &pitch,
      AllocatorT allocator, const property_list &propList = {});
image(void *hostPointer, image_channel_order order,
      image_channel_type type, const range<dimensions> &range,
      range<dimensions - 1> &pitch, const property_list &propList = {});
image(void *hostPointer, image_channel_order order,
      image_channel_type type, const range<dimensions> &range,
      range<dimensions - 1> &pitch, AllocatorT allocator,
      const property_list &propList = {});
image(shared_ptr_class<void> &hostPointer, image_channel_order order,
      image_channel_type type, const range<dimensions> &range,
      const range<dimensions - 1> &pitch, const property_list &propList = {});
image(shared_ptr_class<void> &hostPointer, image_channel_order order,
      image_channel_type type, const range<dimensions> &range,
      const range<dimensions - 1> &pitch, AllocatorT allocator,
      const property_list &propList = {});
```

Parameters

order	
type	
range	
propList	See *Image properties*
allocator	
pitch	
hostPointer	
syclContext	
clMemObject	
availableEvent	

get_range

```
range<dimensions> get_range() const;
```

get_pitch

```
range<dimensions-1> get_pitch() const;
```

Available only when dimensions > 1

get_count

```
size_t get_count() const;
```

get_size

```
size_t get_size() const;
```

get_allocator

```
AllocatorT get_allocator() const;
```

get_access

```
template <typename dataT, access::mode accessMode>
accessor<dataT, dimensions, accessMode, access::target::image>
get_access(handler & commandGroupHandler);
template <typename dataT, access::mode accessMode>
accessor<dataT, dimensions, accessMode, access::target::host_image>
get_access();
```

Template parameters

dataT	
accessMode	

Parameters

commandGroupHandler	

set_final_data

```
template <typename Destination = std::nullptr_t>
void set_final_data(Destination finalData = nullptr);
```

Description

Template parameters

Destination	

Parameters

finalData	

set_write_back

```
void set_write_back(bool flag = true);
```

Parameters

flag	

Image properties

use_host_ptr

```
class use_host_ptr;
```

Namespace

```
property::image
```

Description

Member and nonmember functions

(constructors)

```
use_host_ptr();
```

Description

use_mutex

```
class use_mutex;
```

Namespace

```
property::image
```

Description

Member and nonmember functions

(constructors)

```
use_mutex();
```

Description

get_mutex_ptr

```
mutex_class *get_mutex_ptr() const;
```

Description

context_bound

```
context_bound;
```

Namespace

```
property::image
```

Description

Member and nonmember functions

(constructors)

```
use_mutex();
```

Description

get_context

```
context get_context() const;
```

Description

Image_channel_order

```
enum class image_channel_order : unsigned int {
  a,
  r,
  rx,
  rg,
  rgx,
  ra,
  rgb,
  rgbx,
  rgba,
  argb,
  bgra,
  intensity,
  luminance,
  abgr
}
```

Image_channel_type

```
enum class image_channel_type : unsigned int {
  snorm_int8,
  snorm_int16,
  unorm_int8,
  unorm_int16,
  unorm_short_565,
  unorm_short_555,
  unorm_int_101010,
  signed_int8,
  signed_int16,
  signed_int32,
  unsigned_int8,
  unsigned_int16,
  unsigned_int32,
  fp16,
  fp32
}
```

2.5.3 Accessors

An accessor provides access to the data managed by a buffer or image, or to shared local memory allocated by the runtime.

Buffer accessors

Buffer accessor

```
template <typename dataT, int dimensions, access::mode accessmode,
          access::target accessTarget = access::target::global_buffer,
          access::placeholder isPlaceholder = access::placeholder::false_t>
class accessor;
```

Description

Template parameters

dataT	Type of buffer element
dimensions	Number of buffer dimensions
accessmode	See *mode*
accessTarget	See *target*
isPlaceholder	True if accessor is a placeholder

Member types

value_type	Type of buffer element
reference	Type of reference to buffer element
const_reference	Type of const reference to buffer element

See also:

SYCL Specification Section 4.7.6.9

Member and nonmember functions

(constructors)

```
Available only when:
 ((isPlaceholder == access::placeholder::false_t && accessTarget ==␣
↪access::target::host_buffer)
  || (isPlaceholder == access::placeholder::true_t
      && (accessTarget == access::target::global_buffer
          || accessTarget == access::target::constant_buffer)))
 && dimensions == 0

template <typename AllocatorT>
accessor(buffer<dataT, 1, AllocatorT> &bufferRef,
         const property_list &propList = {});
```

```
Available only when:
 (isPlaceholder == access::placeholder::false_t
  && (accessTarget == access::target::global_buffer
      || accessTarget == access::target::constant_buffer))
 && dimensions == 0

template <typename AllocatorT>
accessor(buffer<dataT, 1, AllocatorT> &bufferRef,
         handler &commandGroupHandlerRef, const property_list &propList = {});

Available only when:
 ((isPlaceholder == access::placeholder::false_t
   && accessTarget == access::target::host_buffer)
  || (isPlaceholder == access::placeholder::true_t
      && (accessTarget == access::target::global_buffer
          || accessTarget == access::target::constant_buffer)))
 && dimensions > 0

template <typename AllocatorT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
         const property_list &propList = {});
template <typename AllocatorT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
         range<dimensions> accessRange, const property_list &propList = {});
template <typename AllocatorT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
         range<dimensions> accessRange, id<dimensions> accessOffset,
         const property_list &propList = {});

Available only when:
 (isPlaceholder == access::placeholder::false_t
  && (accessTarget == access::target::global_buffer
      || accessTarget == access::target::constant_buffer))
  && dimensions > 0

template <typename AllocatorT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
         handler &commandGroupHandlerRef, const property_list &propList = {});
template <typename AllocatorT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
         handler &commandGroupHandlerRef, range<dimensions> accessRange,
         const property_list &propList = {});
template <typename AllocatorT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
         handler &commandGroupHandlerRef, range<dimensions> accessRange,
         id<dimensions> accessOffset, const property_list &propList = {});
```

Construct an accessor for a buffer.

Programs typically find it more convenient to use *get_access* to create an accessor for a buffer.

Template parameters

AllocatorT	Type of allocator for buffer element

Parameters

bufferRef	Associate accessor with this buffer
commandGroupHandlerRef	Associate accessor with this handler
propList	*Buffer accessor properties*
accessRange	Dimensions of data to be accessed
accessOffset	Coordinates of origin of data

is_placeholder

```
constexpr bool is_placeholder() const;
```

Return True if this is a placeholder accessor.

get_size

```
size_t get_size() const;
```

Returns size in bytes of the buffer region that this accesses.

get_count

```
size_t get_count() const;
```

Returns number elements that this accesses.

get_range

```
Available only when:
 dimensions > 0

range<dimensions> get_range() const;
```

Template parameters

dimensions	number of dimensions

Returns dimensions of the asssociated buffer or range that was provided when the accessor was created.

get_offset

```
Available only when:
 dimensions > 0

id<dimensions> get_offset() const;
```

Template parameters

dimensions	number of dimensions

Returns coordinates of the origin of the buffer or offset that was provided when the accessor was created.

operator ()

```
Available only when:
 accessMode == access::mode::write
  || accessMode == access::mode::read_write
  || accessMode == access::mode::discard_write
  || accessMode == access::mode::discard_read_write

operator dataT &() const;

Available only when:
 accessMode == access::mode::read

operator dataT() const;

Available only when:
 accessMode == access::mode::atomic

operator atomic<dataT, access::address_space::global_space> () const;
```

Returns reference or value of element in the associated buffer.

The variants of this operator are only available when *dimensions == 0*, which means that a buffer contains a single element.

operator[]

```
Reference variants
dataT &operator[](size_t index) const;
dataT &operator[](id<dimensions> index) const;

Value variants
dataT operator[](size_t index) const;
dataT operator[](id<dimensions> index) const;

Atomic variants
atomic<dataT, access::address_space::global_space> operator[](
  size_t index) const;
atomic<dataT, access::address_space::global_space> operator[](
  id<dimensions> index) const;

Single dimension in multi-dimensional buffer
__unspecified__ &operator[](size_t index) const;
```

Returns reference or value of element in the associated buffer at the requested index.

One dimensional buffers are indexed by a data of type size_t. Multi-dimensional buffers may be indexed by a data of type `id<dimensions>`, or by a sequence of *[]*, 1 per dimension. For example `a[1][2]`. The operator returns a reference when the accessor allows writes, which requires that accessMode be one of `access::mode::write`, `accessMode == access::mode::read_write`, `accessMode == access::mode::discard_write`, or `accessMode == access::mode::discard_read_write`. The operator returns an atomic if the accessMode is `access::mode::atomic`.

get_pointer

```
Available only when:
 accessTarget == access::target::host_buffer

dataT *get_pointer() const;

Available only when:
 accessTarget == access::target::global_buffer

global_ptr<dataT> get_pointer() const;

Available only when:
 accessTarget == access::target::constant_buffer

constant_ptr<dataT> get_pointer() const;
```

Returns pointer to memory in a host buffer.

Buffer accessor properties

SYCL does not define any properties for the buffer specialization of an accessor.

Local accessor

```
template <typename dataT, int dimensions, access::mode accessmode,
          access::target accessTarget = access::target::global_buffer,
          access::placeholder isPlaceholder = access::placeholder::false_t>
class accessor;
```

Description

Template parameters

dataT	
dimensions	
accessmode	
accessTarget	
isPlaceholder	

Member types

value_type	
reference	
const_reference	

See also:

SYCL Specification Section 4.7.6.11

Member and nonmember functions

(constructors)

```
Available only when:
 dimensions == 0

accessor(handler &commandGroupHandlerRef, const property_list &propList = {});


Available only when:
 dimensions > 0

accessor(range<dimensions> allocationSize, handler &commandGroupHandlerRef,
         const property_list &propList = {});
```

get_size

```
size_t get_size() const;
```

Returns

get_count

```
size_t get_count() const;
```

Returns

get_range

```
range<dimensions> get_range() const;
```

Template parameters

dimensions	

Returns

get_pointer

```
local_ptr<dataT> get_pointer() const;
```

Available only when: accessTarget == access::target::local

operator[]

```
Available only when:
 accessMode == access::mode::read_write && dimensions > 0

dataT &operator[](id<dimensions> index) const;

Available only when:
 accessMode == access::mode::read_write && dimensions == 1

dataT &operator[](size_t index) const

Available only when:
 accessMode == access::mode::atomic && dimensions > 0

atomic<dataT, access::address_space::local_space> operator[](
  id<dimensions> index) const;
```

```
Available only when:
 accessMode == access::mode::atomic && dimensions == 1

atomic<dataT, access::address_space::local_space> operator[](
  size_t index) const;

Available only when:
 dimensions > 1

__unspecified__ &operator[](size_t index) const;
```

operator ()

```
Available only when:
 accessMode == access::mode::read_write && dimensions == 0

operator dataT &() const;

Available only when:
 accessMode == access::mode::atomic && dimensions == 0

operator atomic<dataT,access::address_space::local_space> () const;
```

Image accessor

```
template <typename dataT, int dimensions, access::mode accessmode,
          access::target accessTarget = access::target::global_buffer,
          access::placeholder isPlaceholder = access::placeholder::false_t>
class accessor;
```

Description

Template parameters

dataT	
dimensions	
accessmode	
accessTarget	
isPlaceholder	

Member types

value_type	
reference	
const_reference	

See also:

SYCL Specification Section 4.7.6.12

Member and nonmember functions

(constructors)

Available only when:
accessTarget == access::target::host_image

```
template <typename AllocatorT>
accessor(image<dimensions, AllocatorT> &imageRef,
         const property_list &propList = {});
```

Available only when:
accessTarget == access::target::image

```
template <typename AllocatorT>
accessor(image<dimensions, AllocatorT> &imageRef,
         handler &commandGroupHandlerRef, const property_list &propList = {});
```

Available only when:
accessTarget == access::target::image_array && dimensions < 3

```
template <typename AllocatorT>
accessor(image<dimensions + 1, AllocatorT> &imageRef,
         handler &commandGroupHandlerRef, const property_list &propList = {});
```

get_count

```
size_t get_count() const;
```

get_range

Available only when:
(accessTarget != access::target::image_array)

```
range<dimensions> get_range() const;
```

Available only when:
(accessTarget == access::target::image_array)

```
range<dimensions+1> get_range() const;
```

Template parameters

dimensions	

read

```
Available only when:
 (accessTarget == access::target::image && accessMode == access::mode::read)
 || (accessTarget ==
      access::target::host_image && (accessMode ==
      access::mode::read || accessMode == access::mode::read_write))

template <typename coordT>
dataT read(const coordT &coords) const;

Available only when:
 (accessTarget == access::target::image && accessMode == access::mode::read)
 || (accessTarget ==
      access::target::host_image && (accessMode ==
      access::mode::read || accessMode == access::mode::read_write))

template <typename coordT>
dataT read(const coordT &coords, const sampler &smpl) const;
```

Template parameters

coordT	

operator[]

```
*Available only when:
 accessTarget == access::target::image_array && dimensions < 3*

__image_array_slice__ operator[](size_t index) const;
```

mode

```
enum class mode {
  read = 1024,
  write,
  read_write,
  discard_write,
  discard_read_write,
  atomic
};
```

Namespace

```
access
```

target

```
enum class target {
  global_buffer = 2014,
  constant_buffer,
  local,
  image,
  host_buffer,
  host_image,
  image_array
};
```

Namespace

```
access
```

2.5.4 Multipointer

access::address_space

```
enum class address_space : int {
    global_space,
    local_space,
    constant_space,
    private_space
};
```

See also:

SYCL Specification Section 4.7.7

multi_ptr

```
template <typename ElementType, access::address_space Space> class multi_ptr;

template <access::address_space Space> class multi_ptr<VoidType, Space>;
```

Template parameters

ElementType	
Space	

Member types

element_type	
difference_type	
pointer_t	
const_pointer_t	
reference_t	
const_reference_t	

Nonmember data

address_space	

See also:

SYCL Specification Section 4.7.7.1

Member and nonmember functions

(constructors)

```
multi_ptr();
multi_ptr(const multi_ptr&);
multi_ptr(multi_ptr&&);
multi_ptr(pointer_t);
multi_ptr(ElementType*);
multi_ptr(std::nullptr_t);
```

operator=

```
multi_ptr &operator=(const multi_ptr&);
multi_ptr &operator=(multi_ptr&&);
multi_ptr &operator=(pointer_t);
multi_ptr &operator=(ElementType*);
multi_ptr &operator=(std::nullptr_t);


Available only when:
 Space == global_space

template <int dimensions, access::mode Mode, access::placeholder␣
↪isPlaceholder>
```

```
multi_ptr(accessor<ElementType, dimensions, Mode, access::target::global_
 ↪buffer, isPlaceholder>);

Available only when:
 Space == local_space

template <int dimensions, access::mode Mode, access::placeholder␣
 ↪isPlaceholder>
multi_ptr(accessor<ElementType, dimensions, Mode, access::target::local,␣
 ↪isPlaceholder>);

Available only when:
 Space == constant_space

template <int dimensions, access::mode Mode, access::placeholder␣
 ↪isPlaceholder>
multi_ptr(accessor<ElementType, dimensions, Mode, access::target::constant_
 ↪buffer, isPlaceholder>);
```

Template parameters

dimensions	
Mode	
isPlaceholder	

operator*

```
friend ElementType& operator*(const multi_ptr& mp);
```

operator->

```
ElementType* operator->() const;
```

get

```
pointer_t get() const;
```

Returns

Returns the underlying OpenCL C pointer

(Implicit conversions)

```
Implicit conversion to the underlying pointer type

operator ElementType*() const;

Implicit conversion to a multi_ptr<void>.  Only available
 when ElementType is not const-qualified

operator multi_ptr<void, Space>() const;

Implicit conversion to a multi_ptr<const void>. Only
 available when ElementType is const-qualified

operator multi_ptr<const void, Space>() const;

Implicit conversion to multi_ptr<const ElementType, Space>

operator multi_ptr<const ElementType, Space>() const;
```

(Arithmetic operators)

```
friend multi_ptr& operator++(multi_ptr& mp);
friend multi_ptr operator++(multi_ptr& mp, int);
friend multi_ptr& operator--(multi_ptr& mp);
friend multi_ptr operator--(multi_ptr& mp, int);
friend multi_ptr& operator+=(multi_ptr& lhs, difference_type r);
friend multi_ptr& operator-=(multi_ptr& lhs, difference_type r);
friend multi_ptr operator+(const multi_ptr& lhs, difference_type r);
friend multi_ptr operator-(const multi_ptr& lhs, difference_type r);
```

prefetch

```
void prefetch(size_t numElements) const;
```

(Relational operators)

```
friend bool operator==(const multi_ptr& lhs, const multi_ptr& rhs);
friend bool operator!=(const multi_ptr& lhs, const multi_ptr& rhs);
friend bool operator<(const multi_ptr& lhs, const multi_ptr& rhs);
friend bool operator>(const multi_ptr& lhs, const multi_ptr& rhs);
friend bool operator<=(const multi_ptr& lhs, const multi_ptr& rhs);
friend bool operator>=(const multi_ptr& lhs, const multi_ptr& rhs);

friend bool operator==(const multi_ptr& lhs, std::nullptr_t);
friend bool operator!=(const multi_ptr& lhs, std::nullptr_t);
friend bool operator<(const multi_ptr& lhs, std::nullptr_t);
friend bool operator>(const multi_ptr& lhs, std::nullptr_t);
friend bool operator<=(const multi_ptr& lhs, std::nullptr_t);
friend bool operator>=(const multi_ptr& lhs, std::nullptr_t);

friend bool operator==(std::nullptr_t, const multi_ptr& rhs);
friend bool operator!=(std::nullptr_t, const multi_ptr& rhs);
friend bool operator<(std::nullptr_t, const multi_ptr& rhs);
friend bool operator>(std::nullptr_t, const multi_ptr& rhs);
friend bool operator<=(std::nullptr_t, const multi_ptr& rhs);
friend bool operator>=(std::nullptr_t, const multi_ptr& rhs);
```

2.5.5 private_memory

```
template <typename T, int Dimensions = 1>
class private_memory;
```

See also:

SYCL Specification Section 4.10.7.3

Member and nonmember functions

(constructors)

```
private_memory(const group<Dimensions> &);
```

(operators)

```
T &operator()(const h_item<Dimensions> &id);
```

2.5.6 Samplers

See also:

SYCL Specification Section 4.7.8

address_mode

```
enum class addressing_mode: unsigned int {
    mirrored_repeat,
    repeat,
    clamp_to_edge,
    clamp,
    none
};
```

filtering_mode

```
enum class filtering_mode: unsigned int {
   nearest,
   linear
};
```

coordinate_normalization_mode

```
enum class coordinate_normalization_mode : unsigned int {
   normalized,
   unnormalized
};
```

sampler

```
class sampler;
```

(constructors)

```
sampler(coordinate_normalization_mode normalizationMode,
        addressing_mode addressingMode, filtering_mode filteringMode,
        const property_list &propList = {});

sampler(cl_sampler clSampler, const context &syclContext);
```

get_address_mode

```
addressing_mode get_addressing_mode() const;
```

get_filtering_mode

```
filtering_mode get_filtering_mode() const;
```

get_coordinate_normalization_mode

```
coordinate_normalization_mode get_coordinate_normalization_mode() const;
```

2.6 Unified shared memory (USM)

2.6.1 malloc_device

Since SYCL 2020

```
void* malloc_device(size_t num_bytes,
                    const device& dev,
                    const context& ctxt);
void* aligned_alloc_device(size_t alignment,
                           size_t num_bytes,
                           const device& dev,
                           const context& ctxt);
template <typename T>
T* malloc_device(size_t count,
                 const device& dev,
                 const context& ctxt);
template <typename T>
T* aligned_alloc_device(size_d alignment,
                        size_t count,
                        const device& dev,
                        const context& ctxt);
```

Parameters

alignment	alignment of allocated data
num_bytes	allocation size in bytes
count	number of elements
dev	See *device*
ctxt	See *context*

Returns a pointer to memory that resides on the device.

The host may not directly reference the memory, but can read and write the memory with *queue* member functions (*memset*, *memcpy*, *fill*) or *handler* member functions (*memset*, *memcpy*, and *fill*).

Deallocate with *free*.

See also:

SYCL Specification Section 4.8.5.1

2.6.2 malloc_host

Since SYCL 2020

```
void* malloc_host(size_t num_bytes, const context& ctxt);
template <typename T>
void* aligned_alloc_host(size_t alignment, size_t num_bytes, const context& ctxt);
T* malloc_host(size_t count, const context& ctxt);
T* aligned_alloc_host(size_t alignment, size_t count, const context& ctxt);
```

Parameters

alignment	alignment of allocated data
num_bytes	allocation size in bytes
count	number of elements
dev	See *device*
ctxt	See *context*

Returns a pointer to memory that resides on the host.

Host and device may reference the memory.

Deallocate with *free*.

See also:

SYCL Specification Section 4.8.5.2

2.6.3 malloc_shared

Since SYCL 2020

```
void* malloc_shared(size_t num_bytes,
                    const device& dev,
                    const context& ctxt);
void* aligned_alloc_ahared(size_t alignment,
                           size_t num_bytes,
                           const device& dev,
                           const context& ctxt);
template <typename T>
T* malloc_shared(size_t count,
                 const device& dev,
                 const context& ctxt);
template <typename T>
T* aligned_alloc_ahared(size_t alignment,
                        size_t count,
                        const device& dev,
                        const context& ctxt);
```

Parameters

alignment	alignment of allocated data
num_bytes	allocation size in bytes
count	number of elements
dev	See *device*
ctxt	See *context*

Returns a pointer to memory that may reside on host or device.

The SYCL runtime may migrate the data between host and device to optimize access.

Deallocate with *free*.

See also:

SYCL Specification Section 4.8.5.2

2.6.4 free

Since SYCL 2020

```
void free(void* ptr, context& context);
```

Free memory allocated by *malloc_device*, *malloc_host*, or *malloc_shared*.

See also:

SYCL Specification Section 4.8.5.4

2.6.5 usm_allocator

Since SYCL 2020

```
template <typename T, usm::alloc AllocKind, size_t Alignment = 0>
class usm_allocator;
```

Allocator suitable for use with a C++ standard library container.

A `usm_allocator` enables using USM allocation for standard library containers. It is typically passed as template parameter when declaring standard library containers (e.g. vector).

Template parameters

T	Type of allocated element
AllocKind	Type of allocation, see o
Alignment	Alignment of the allocation

Example

```
#include <vector>

#include <CL/sycl.hpp>

using namespace sycl;

const int size = 10;

int main() {
  queue q;

  // USM allocator for data of type int in shared memory
  typedef usm_allocator<int, usm::alloc::shared> vec_alloc;
  // Create allocator for device associated with q
  vec_alloc myAlloc(q);
  // Create std vectors with the allocator
  std::vector<int, vec_alloc >
    a(size, myAlloc),
    b(size, myAlloc),
    c(size, myAlloc);

  // Get pointer to vector data for access in kernel
  auto A = a.data();
  auto B = b.data();
  auto C = c.data();

  for (int i = 0; i < size; i++) {
    a[i] = i;
    b[i] = i;
    c[i] = i;
  }

  q.submit([&](handler &h) {
      h.parallel_for(range<1>(size),
                     [=](id<1> idx) {
                       C[idx] = A[idx] + B[idx];
                     });
    }).wait();

  for (int i = 0; i < size; i++) std::cout << c[i] << std::endl;
  return 0;
}
```

Member types

value_type	

See also:

SYCL Specification Section 4.8.4

Member and nonmember functions

(constructors)

```
usm_allocator(const context &ctxt, const device &dev) noexcept;
usm_allocator(const queue &q) noexcept;
usm_allocator(const usm_allocator &other) noexcept;
template <class U>
usm_allocator(usm_allocator<U, AllocKind, Alignment> const &) noexcept;
```

allocate

```
T *allocate(size_t Size);
```

Allocates memory

deallocate

```
void deallocate(T *Ptr, size_t size);
```

Deallocates memory

construct

```
template <
    usm::alloc AllocT = AllocKind,
    typename std::enable_if<AllocT != usm::alloc::device, int>::type = 0,
    class U, class... ArgTs>
void construct(U *Ptr, ArgTs &&... Args);
template <
    usm::alloc AllocT = AllocKind,
    typename std::enable_if<AllocT == usm::alloc::device, int>::type = 0,
    class U, class... ArgTs>
void construct(U *Ptr, ArgTs &&... Args);
```

Constructs an object on memory pointed by Ptr.

destroy

```
template <
    usm::alloc AllocT = AllocKind,
    typename std::enable_if<AllocT != usm::alloc::device, int>::type = 0>
void destroy(T *Ptr);

/// Throws an error when trying to destroy a device allocation
/// on the host
template <
    usm::alloc AllocT = AllocKind,
    typename std::enable_if<AllocT == usm::alloc::device, int>::type = 0>
void destroy(T *Ptr);
```

Destroys an object.

(operators)

```
template <class T, usm::alloc AllocKindT, size_t AlignmentT, class U,
          usm::alloc AllocKindU, size_t AlignmentU>
bool operator==(const usm_allocator<T, AllocKindT, AlignmentT> &,
                const usm_allocator<U, AllocKindU, AlignmentU> &) noexcept;
template <class T, class U, usm::alloc AllocKind, size_t Alignment = 0>
bool operator!=(const usm_allocator<T, AllocKind, Alignment> &allocT,
                const usm_allocator<U, AllocKind, Alignment> &allocU) noexcept;
```

Allocators only compare equal if they are of the same USM kind, alignment, context, and device (when kind is not host).

2.6.6 alloc

Since SYCL 2020

```
enum class alloc {
  host,
  device,
  shared,
  unknown
};
```

Namespace

```
usm
```

Identifies type of USM memory in calls to USM-related API.

host Resides on host and also accessible by device

device Resides on device and only accessible by device

shared SYCL runtime may move data between host and device. Accessible by host and device.

See also:

SYCL Specification Section 4.8.3

2.7 Expressing parallelism

2.7.1 range

```
template <int dimensions = 1>
class range;
```

The range is an abstraction that describes the number of elements in each dimension of buffers and index spaces. It can contain 1, 2, or 3 numbers, dependending on the dimensionality of the object it describes.

Template parameters

dimensions	Number of dimensions

See also:

SYCL Specification Section 4.10.1.1

Member and nonmember functions

(constructors)

```
range(size_t dim0);
range(size_t dim0, size_t dim1);
range(size_t dim0, size_t dim1, size_t dim2);
```

Constructs a 1, 2, or 3 dimensional range.

get

```
size_t get(int dimension) const;
```

Returns the range of a single dimension.

operator[]

```
size_t &operator[](int dimension);
size_t operator[](int dimension) const;
```

Returns the range of a single dimension.

size

```
size_t size() const;
```

Returns the size of a range by multiplying the range of the individual dimensions.

For a buffer, it is the number of elements in the buffer.

Arithmetic Operators

*OP is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=*

```
friend range operatorOP(const range &lhs, const range &rhs)
friend range operatorOP(const range &lhs, const size_t &rhs)
friend range operatorOP(const size_t &lhs, const range &rhs)
```

*OP is: +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=*

```
friend range & operatorOP(const range &lhs, const range &rhs)
```

```
friend range & operatorOP(const range &lhs, const size_t &rhs)
```

Arithmetical and relational operations on ranges.

2.7.2 group

```
template <int dimensions = 1>
class group;
```

Template parameters

dimensions	

See also:

SYCL Specification Section 4.10.1.7

Member and nonmember functions

get_id

```
id<dimensions> get_id() const;
size_t get_id(int dimension) const;
```

get_global_range

```
range<dimensions> get_global_range() const;
size_t get_global_range(int dimension) const;
```

get_local_range

```
range<dimensions> get_local_range() const;
size_t get_local_range(int dimension) const;
```

get_group_range

```
range<dimensions> get_group_range() const;
size_t get_group_range(int dimension) const;
```

get_linear_id

```
size_t get_linear_id() const;
```

parallel_for_work_item

```
template<typename workItemFunctionT>
void parallel_for_work_item(workItemFunctionT func) const;
template<typename workItemFunctionT>
void parallel_for_work_item(range<dimensions> logicalRange,
  workItemFunctionT func) const;
```

mem_fence

```
template <access::mode accessMode = access::mode::read_write>
void mem_fence(access::fence_space accessSpace =
  access::fence_space::global_and_local) const;
```

async_work_group_copy

```
template <typename dataT>
device_event async_work_group_copy(local_ptr<dataT> dest,
  global_ptr<dataT> src, size_t numElements) const;
template <typename dataT>
device_event async_work_group_copy(global_ptr<dataT> dest,
  local_ptr<dataT> src, size_t numElements) const;
template <typename dataT>
device_event async_work_group_copy(local_ptr<dataT> dest,
  global_ptr<dataT> src, size_t numElements, size_t srcStride) const;
template <typename dataT>
device_event async_work_group_copy(global_ptr<dataT> dest,
  local_ptr<dataT> src, size_t numElements, size_t destStride) const;
```

wait_for

```
template <typename... eventTN>
void wait_for(eventTN... events) const;
```

operator[]

```
size_t operator[](int dimension) const;
```

2.7.3 id

```
template <int dimensions = 1>
class id;
```

The `id` is an abstraction that describes the location of a point in a *range*. Examples includes use as an index in an *Buffer accessor* and as an argument to a kernel function in a *parallel_for* to identify the work item.

See also:

SYCL Specification Section 4.10.1.3

Member and nonmember functions

(constructors)

```
id();
id(size_t dim0);
id(size_t dim0, size_t dim1);
id(size_t dim0, size_t dim1, size_t dim2);

id(const range<dimensions> &range);
id(const item<dimensions> &item);
```

Construct an `id`.

An `id` can be 0, 1, 2, or 3 dimensions. An `id` constructed from a *range* uses the `range` values. An `id` constructed from an *item* uses the `id` contained in the `item`.

get

```
size_t get(int dimension) const;
```

Returns the value for dimension `dimension`.

(operators)

```
size_t &operator[](int dimension);
size_t operator[](int dimension) const;

*OP is:
 +, -, \*, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=*

friend id operatorOP(const id &lhs, const id &rhs);
friend id operatorOP(const id &lhs, const size_t &rhs);

*OP is:
 +=, -=, \*=, /=, %=, <<=, >>=, &=, |=, ^=*

friend id &operatorOP(id &lhs, const id &rhs);
friend id &operatorOP(id &lhs, const size_t &rhs);

*OP is:
```

(continues on next page)

(continued from previous page)

```
 +, -, \*, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=*

 friend id operatorOP(const size_t &lhs, const id &rhs);
```

Relational, arithmetic, and indexing operators on an `id`.

2.7.4 item

```
template <int dimensions = 1, bool with_offset = true>
class item;
```

Similar to an *id*, the `item` describes the location of a point in a `range`. It can be used as an argument to a kernel function in a *parallel_for* to identify the work item. The `item` carries more information than than *id*, such as the `range` of an index space. The interface does not include a constructor because only the SYCL runtime needs to construct an `item`.

Template parameters

dimensions	Number of dimensions in index space
with_offset	True if item has offset

See also:

SYCL Specification Section 4.10.1.4

Member and nonmember functions

get_id

```
id<dimensions> get_id() const;
size_t get_id(int dimension) const;
```

Returns *id* associated with `item`.

get_range

```
range<dimensions> get_range() const;
size_t get_range(int dimension) const;
```

Returns *range* associated with `item`.

get_offset

```
*Only available when:
 with_offset is true*

id<dimensions> get_offset() const;
```

Returns offset associated with `item`.

get_linear_id

```
size_t get_linear_id() const;
```

Returns the linear id, suitable for mapping the `id` to a 1 dimensional array.

operator[]

```
size_t operator[](int dimension) const;
```

Returns id for dimension `dimension`.

operator()

```
operator item<dimensions, true>() const;
```

Returns item with offset set to 0.

Only available when `with_offset` is False.

2.7.5 h_item

```
template <int dimensions>
class h_item;
```

See also:

SYCL Specification Section 4.10.1.6

Member and nonmember functions

get_global

```
item<dimensions, false> get_global() const;
```

get_local

```
item<dimensions, false> get_local() const;
```

get_logical_local

```
item<dimensions, false> get_logical_local() const;
```

get_physical_local

```
item<dimensions, false> get_physical_local() const;
```

get_global_range

```
range<dimensions> get_global_range() const;
size_t get_global_range(int dimension) const;
```

get_global_id

```
id<dimensions> get_global_id() const;
size_t get_global_id(int dimension) const;
```

get_local_range

```
range<dimensions> get_local_range() const;
size_t get_local_range(int dimension) const;
```

get_local_id

```
id<dimensions> get_local_id() const;
size_t get_local_id(int dimension) const;
```

get_logical_local_range

```
range<dimensions> get_logical_local_range() const;
size_t get_logical_local_range(int dimension) const;
```

get_logical_local_id

```
id<dimensions> get_logical_local_id() const;
size_t get_logical_local_id(int dimension) const;
```

get_physical_local_range

```
range<dimensions> get_physical_local_range() const;
size_t get_physical_local_range(int dimension) const;
```

get_physical_local_id

```
id<dimensions> get_physical_local_id() const;
size_t get_physical_local_id(int dimension) const;
```

2.7.6 nd_item

```
template <int dimensions = 1>
class nd_item;
```

The `nd_item` describes the location of a point in an *nd_range*.

An `nd_item` is typically passed to a kernel function in a *parallel_for*. It addition to containing the `id` of the work item in the work group and global space, the `nd_item` also contains the *nd_range* defining the index space.

See also:

SYCL Specification Section 4.10.1.5

Member and nonmember functions

get_global_id

```
id<dimensions> get_global_id() const;
size_t get_global_id(int dimension) const;
```

Returns global *id* for the requested dimensions.

get_global_linear_id

```
size_t get_global_linear_id() const;
```

Returns global id mapped to a linear space.

get_local_id

```
id<dimensions> get_local_id() const;
size_t get_local_id(int dimension) const;
```

Returns id for the point in the work group.

get_local_linear_id

```
size_t get_local_linear_id() const;
```

Returns linear id for point in the work group.

get_group

```
group<dimensions> get_group() const;
size_t get_group(int dimension) const;
```

Returns *group* associated with the item.

get_group_linear_id

```
size_t get_group_linear_id() const;
```

Returns linear id for group in workspace.

get_group_range

```
range<dimensions> get_group_range() const;
size_t get_group_range(int dimension) const;
```

Returns the number of groups in every dimension.

get_global_range

```
range<dimensions> get_global_range() const;
size_t get_global_range(int dimension) const;
```

Returns the *range* of the index space.

get_local_range

```
range<dimensions> get_local_range() const;
size_t get_local_range(int dimension) const;
```

Returns the position of the work item in the work group.

get_offset

```
id<dimensions> get_offset() const;
```

Returns the offset provided to the *parallel_for*.

get_nd_range

```
nd_range<dimensions> get_nd_range() const;
```

Returns the *nd_range* provided to the *parallel_for*.

barrier

```
void barrier(access::fence_space accessSpace =
  access::fence_space::global_and_local) const;
```

Executes a work group barrier.

mem_fence

```
template <access::mode accessMode = access::mode::read_write>
void mem_fence(access::fence_space accessSpace =
  access::fence_space::global_and_local) const;
```

Executes a work group memory fence.

async_work_group_copy

```
template <typename dataT>
device_event async_work_group_copy(local_ptr<dataT> dest,
  global_ptr<dataT> src, size_t numElements) const;
template <typename dataT>
device_event async_work_group_copy(global_ptr<dataT> dest,
  local_ptr<dataT> src, size_t numElements) const;
template <typename dataT>
device_event async_work_group_copy(local_ptr<dataT> dest,
  global_ptr<dataT> src, size_t numElements, size_t srcStride) const;
template <typename dataT>
device_event async_work_group_copy(global_ptr<dataT> dest,
  local_ptr<dataT> src, size_t numElements, size_t destStride) const;
```

Copies elements from a source local to the destination asynchronously.

Returns an event that indicates when the operation has completed.

wait_for

```
template <typename... eventTN>
void wait_for(eventTN... events) const;
```

Wait for asynchronous events to complete.

2.7.7 nd_range

```
template <int dimensions = 1>
class nd_range;
```

The `nd_range` defines the index space for a work group as well as the global index space. It is passed to *parallel_for* to execute a kernel on a set of work items.

Template parameters

dimensions	Number of dimensions

See also:

SYCL Specification Section 4.10.1.2

Member and nonmember functions

(constructors)

```
nd_range(range<dimensions> globalSize, range<dimensions> localSize,
         id<dimensions> offset = id<dimensions>());
```

Construct an `nd_range`.

Parameters

globalSize	dimensions of the entire index space
localSize	dimensions of the work group
offset	Origin of the index space

get_global_range

```
range<dimensions> get_global_range() const;
```

Returns a *range* defining the index space.

get_local_range

```
range<dimensions> get_local_range() const;
```

Returns a *range* defining the index space of a work group.

get_group_range

```
range<dimensions> get_group_range() const;
```

Returns a *range* defining the number of work groups in every dimension.

get_offset

```
id<dimensions> get_offset() const;
```

Returns a *id* defining the offset.

2.7.8 device_event

```
class device_event;
```

See also:

SYCL Specification Section 4.7.8

Member and nonmember functions

wait

```
void wait();
```

2.7.9 Command groups

Command group function object

command_group

```
class command_group;
```

Member and nonmember functions

(constructors)

```
template <typename functorT>
command_group(queue &primaryQueue, const functorT &lambda);
template <typename functorT>
command_group(queue &primaryQueue, queue &secondaryQueue,
              const functorT &lambda);
```

events

```
event start_event();
event kernel_event();
event complete_event();
```

2.7.10 Invoking kernels

handler

```
class handler;
```

The `handler` defines the interface to invoke kernels by submitting commands to a queue.

A `handler` can only be constructed by the SYCL runtime and is passed as an argument to the command group function. The command group function is an argument to *submit*.

See also:

SYCL Specification Section 4.10.4

Member and nonmember functions

require

```
template <typename dataT, int dimensions, access::mode accessMode,
  access::target accessTarget>
void require(accessor<dataT, dimensions, accessMode, accessTarget,
             access::placeholder::true_t> acc);
```

Adds a requirement before a device may execute a kernel.

set_arg

```
template <typename T>
void set_arg(int argIndex, T && arg);
```

Sets a kernel argument.

set_args

```
template <typename... Ts>
void set_args(Ts &&... args);
```

Sets all kernel arguments.

single_task

```
template <typename KernelName, typename KernelType>
void single_task(KernelType kernelFunc);

void single_task(kernel syclKernel);
```

Defines and invokes a kernel function.

parallel_for

```
template <typename KernelName, typename KernelType, int dimensions>
void parallel_for(range<dimensions> numWorkItems, KernelType kernelFunc);

template <typename KernelName, typename KernelType, int dimensions>
void parallel_for(range<dimensions> numWorkItems,
                  id<dimensions> workItemOffset, KernelType kernelFunc);

template <typename KernelName, typename KernelType, int dimensions>
void parallel_for(nd_range<dimensions> ndRange, KernelType kernelFunc);

template <int dimensions>
void parallel_for(range<dimensions> numWorkItems, kernel syclKernel);

template <int dimensions>
void parallel_for(range<dimensions> numWorkItems,
                  id<dimensions> workItemOffset, kernel syclKernel);

template <int dimensions>
void parallel_for(nd_range<dimensions> ndRange, kernel syclKernel);
```

Invokes a kernel function for a *range* or *nd_range*.

Parameters

numWorkItems	Range for work items
workItemOffset	Offset into range for work items
kernelFunc	Kernel function
syclKernel	See *kernel*
ndRange	See *nd_range*

parallel_for_work_group

```
template <typename KernelName, typename WorkgroupFunctionType, int dimensions>
void parallel_for_work_group(range<dimensions> numWorkGroups,
                             WorkgroupFunctionType kernelFunc);

template <typename KernelName, typename WorkgroupFunctionType, int dimensions>
void parallel_for_work_group(range<dimensions> numWorkGroups,
                             range<dimensions> workGroupSize,
                             WorkgroupFunctionType kernelFunc);
```

Outer invocation in a hierarchical invocation of a kernel.

The kernel function is executed once per work group.

copy

```
template <typename T_src, int dim_src, access::mode mode_src, access::target tgt_src,
          access::placeholder isPlaceholder, typename T_dest>
void copy(accessor<T_src, dim_src, mode_src, tgt_src, isPlaceholder> src,
          shared_ptr_class<T_dest> dest);
template <typename T_src,
          typename T_dest, int dim_dest, access::mode mode_dest, access::target tgt_
↪dest,
          access::placeholder isPlaceholder>
void copy(shared_ptr_class<T_src> src,
          accessor<T_dest, dim_dest, mode_dest, tgt_dest, isPlaceholder> dest);
template <typename T_src, int dim_src, access::mode mode_src,
          access::target tgt_src, access::placeholder isPlaceholder,
          typename T_dest>
void copy(accessor<T_src, dim_src, mode_src, tgt_src, isPlaceholder> src,
          T_dest *dest);
template <typename T_src,
          typename T_dest, int dim_dest, access::mode mode_dest,
          access::target tgt_dest, access::placeholder isPlaceholder>
void copy(const T_src *src,
          accessor<T_dest, dim_dest, mode_dest, tgt_dest, isPlaceholder> dest);
template <typename T_src, int dim_src, access::mode mode_src,
          access::target tgt_src, access::placeholder isPlaceholder_src,
          typename T_dest, int dim_dest, access::mode mode_dest, access::target tgt_
↪dest,
          access::placeholder isPlaceholder_dest>
void copy(accessor<T_src, dim_src, mode_src, tgt_src, isPlaceholder_src> src,
          accessor<T_dest, dim_dest, mode_dest, tgt_dest, isPlaceholder_dest> dest);
```

Copies memory from `src` to `dest`.

`copy` invokes the operation on a *device*. The source, destination, or both source and destination are *Accessors*. Source or destination can be a pointer or a `shared_ptr`.

Template parameters

T_src	Type of source data elements
dim_src	Dimensionality of source accessor data
T_dest	Type of element for destination data
dim_dest	Dimensionality of destination accessor data
mode_src	Mode for source accessor
mode_dest	Mode for destination accessor
tgt_src	Target for source accessor
tgt_dest	Target for destination accessor
isPlaceholder_src	Placeholder value for source accessor
isPlaceholder_dest	Placeholder value for destination accessor

Parameters

src	source of copy
dest	destination of copy

update_host

```
template <typename T, int dim, access::mode mode,
          access::target tgt, access::placeholder isPlaceholder>
void update_host(accessor<T, dim, mode, tgt, isPlaceholder> acc);
```

Template parameters

T	Type of element associated with accessor
dim	Dimensionality of accessor
mode	Access mode for accessor
tgt	Target for accessor
isPlaceholder	Placeholder value for accessor

Updates host copy of data associated with accessor.

fill

```
template <typename T, int dim, access::mode mode,
          access::target tgt, access::placeholder isPlaceholder>
void fill(accessor<T, dim, mode, tgt, isPlaceholder> dest, const T& pattern);
template <typename T>
event fill(void* ptr, const T& pattern, size_t count);
```

Template parameters

T	Type of element associated with accessor
dim	Dimensionality of accessor
mode	Access mode for accessor
tgt	Target for accessor
isPlaceholder	Placeholder value for accessor

Parameters

dest	Destination of fill operation
pattern	Value to fill

Fill the destination with the value in `pattern`. The destination may be memory associated with an accessor or allocated with *malloc_device*.

memcpy

```
void memcpy(void* dest, const void* src, size_t num_bytes);
```

Set memory allocated with *malloc_device*.

memset

```
void memset(void* ptr, int value, size_t num_bytes);
```

Set memory allocated with *malloc_device*.

2.7.11 Kernel

kernel

```
class kernel;
```

Abstraction of a kernel object.

See also:

SYCL Specification Section 4.12

Member and nonmember functions

(constructors)

```
kernel(cl_kernel clKernel, const context& syclContext);
```

Constructs a SYCL kernel instance from an OpenCL kernel.

get

```
cl_kernel get() const;
```

Returns OpenCL kernel associated with the SYCL kernel.

is_host

```
bool is_host() const;
```

Return true if this SYCL kernel is a host kernel.

get_context

```
context get_context() const;
```

Returns context associated with the kernel.

get_program

```
program get_program() const;
```

Returns program that this kernel is part of.

get_info

```
template <info::kernel param>
typename info::param_traits<info::kernel, param>::return_type
get_info() const;
```

Template parameters

param	See *info::kernel*

Returns information about the kernel

get_work_group_info

```
template <info::kernel_work_group param>
typename info::param_traits<info::kernel_work_group, param>::return_type
get_work_group_info(const device &dev) const;
```

Template parameters

param	See *info::kernel_work_group*

Returns information about the work group

info::kernel

```
enum class kernel: int {
    function_name,
    num_args,
    context,
    program,
    reference_count,
    attributes
};
```

info::kernel_work_group

```
enum class kernel_work_group: int {
    global_work_size,
    work_group_size,
    compile_work_group_size,
    preferred_work_group_size_multiple,
    private_mem_size
};
```

2.7.12 Program

info::program

```
enum class program: int {
    context,
    devices,
    reference_count
};
```

program_state

```
enum class program_state {
    none,
    compiled,
    linked
};
```

program

```
class program;
```

(constructors)

```
explicit program(const context &context,
                 const property_list &propList = {});
program(const context &context, vector_class<device> deviceList,
        const property_list &propList = {});
program(vector_class<program> &programList,
        const property_list &propList = {});
program(vector_class<program> &programList, string_class linkOptions,
        const property_list &propList = {});
program(const context &context, cl_program clProgram);
```

get

```
cl_program get() const;
```

is_host

```
bool is_host() const;
```

compile_with_kernel_type

```
template <typename kernelT>
void build_with_kernel_type(string_class buildOptions = "");
```

build_with_source

```
void build_with_source(string_class kernelSource,
                       string_class buildOptions = "");
```

link

```
void link(string_class linkOptions = "");
```

has_kernel

```
template <typename kernelT>
bool has_kernel<kernelT>() const;

bool has_kernel(string_class kernelName) const;
```

get_kernel

```
template <typename kernelT>
kernel get_kernel<kernelT>() const;

kernel get_kernel(string_class kernelName) const;
```

get_info

```
template <info::program param>
typename info::param_traits<info::program, param>::return_type
get_info() const;
```

get_binaries

```
vector_class<vector_class<char>> get_binaries() const;
```

get_context

```
context get_context() const;
```

get_devices

```
vector_class<device> get_devices() const;
```

get_compile_options

```
string_class get_compile_options() const;
```

get_link_options

```
string_class get_link_options() const;
```

get_build_options

```
string_class get_build_options() const;
```

get_state

```
program_state get_state() const;
```

2.8 Error handling

2.8.1 Exceptions

exception

```
class exception;
```

See also:

SYCL Specification Section 4.15.2

Member and nonmember functions

Container for an exception that occurs during execution. Synchronous API's throw exceptions that may be caught with C++ exception handling methods. The SYCL runtime holds exceptions that occur during asynchronous operations until *wait_and_throw* or *throw_asynchronous* is called. They runtime delivers the exception as a list to the *async_handler* associated with the *queue*.

what

```
const char *what() const;
```

Returns string that describes the error that triggered the exception.

has_context

```
bool has_context() const;
```

Returns true if error has an associated *context*.

get_context

```
context get_context() const;
```

Returns *context* associated with this error.

get_cl_code

```
cl_int get_cl_code() const;
```

Returns OpenCL error code if the error is an OpenCL error, otherwise `CL_SUCCESS`.

exception_list

```
class exception_list;
```

An exContainer for a list of asychronous exceptions that occur in the same queue. Re

Member types

value_type	
reference	
const_reference	
size_type	
iterator	
const_iterator	

Member and nonmember functions

size

```
size_type size() const;
```

Returns number of elements in the list.

begin

```
iterator begin() const;
```

Returns an iterator to the beginning of the list of exceptions.

end

```
iterator end() const;
```

Returns an iterator to the beginning of the list of exceptions.

Derived exceptions

runtime_error

```
class runtime_error : public exception;
```

kernel_error

```
class kernel_error : public runtime_error;
```

Error that occured before or while enqueuing the SYCL kernel.

accessor_error

```
class accessor_error : public runtime_error;
```

Error regarding *Accessors*.

nd_range_error

```
class nd_range_error : public runtime_error;
```

Error regarding the *nd_range* for a SYCL kernel.

event_error

```
class event_error : public runtime_error;
```

Error regarding an *event*.

invalid_parameter_error

```
class invalid_parameter_error : public runtime_error;
```

Error regarding parameters to a SYCL kernel, including captured parameters to a lambda.

device_error

```
class device_error : public exception;
```

compile_program_error

```
class compile_program_error : public device_error;
```

Error while compiling a SYCL kernel.

link_program_error

```
class link_program_error : public device_error;
```

Error linking a SYCL kernel to a SYCL device.

invalid_object_error

```
class invalid_object_error : public device_error;
```

Error regarding memory objects used inside a kernel.

memory_allocation_error

```
class memory_allocation_error : public device_error;
```

Error regarding memory allocation on the SYCL device.

platform_error

```
class platform_error : public device_error;
```

Error triggered by the *platform*.

profililng_error

```
class profiling_error : public device_error;
```

Error triggered while profiling is enabled.

featured_non_supported

```
class feature_not_supported : public device_error;
```

Optional feature or extension is not available on the *device*.

async_handler

```
void handler(exception_list e);
```

Parameters

e	List of asynchronous exceptions. See *exception_list*

The SYCL runtime delivers asynchronous exceptions by invoking an `async_handler`. The handler is passed to a *queue* constructor. The SYCL runtime delivers asynchronous exceptions to the handler when *wait_and_throw* or *throw_asynchronous* is called.

2.9 Data types

2.9.1 Scalar types

byte

OpenCL types

2.9.2 Vector types

rounding_mode

```
enum class rounding_mode {
    automatic,
    rte,
    rtz,
    rtp,
    rtn
};
```

elem

```
struct elem {
    static constexpr int x = 0;
    static constexpr int y = 1;
    static constexpr int z = 2;
    static constexpr int w = 3;
    static constexpr int r = 0;
    static constexpr int g = 1;
    static constexpr int b = 2;
    static constexpr int a = 3;
    static constexpr int s0 = 0;
    static constexpr int s1 = 1;
    static constexpr int s2 = 2;
    static constexpr int s3 = 3;
    static constexpr int s4 = 4;
    static constexpr int s5 = 5;
    static constexpr int s6 = 6;
    static constexpr int s7 = 7;
    static constexpr int s8 = 8;
    static constexpr int s9 = 9;
    static constexpr int sA = 10;
    static constexpr int sB = 11;
    static constexpr int sC = 12;
    static constexpr int sD = 13;
    static constexpr int sE = 14;
    static constexpr int sF = 15;
};
```

vec

```
template <typename dataT, int numElements>
class vec;
```

Member types

element_type	
vector_t	

(constructors)

```
vec();
explicit vec(const dataT &arg);
template <typename... argTN>
vec(const argTN&... args);
vec(const vec<dataT, numElements> &rhs);
vec(vector_t openclVector);
```

Conversion functions

```
 operator vector_t() const;
```

Available when:
numElements == 1

```
operator dataT() const;
```

get_count

```
size_t get_count() const;
```

get_size

```
size_t get_size() const;
```

convert

```
template <typename convertT, rounding_mode roundingMode = rounding_mode::automatic>
vec<convertT, numElements> convert() const;
```

as

```
template <typename asT>
asT as() const;
```

swizzle

```
template<int... swizzleIndexes>
__swizzled_vec__ swizzle() const;
```

swizzle access

```
__swizzled_vec__ x() const;
__swizzled_vec__ y() const;
__swizzled_vec__ z() const;

__swizzled_vec__ w() const;
__swizzled_vec__ r() const;
__swizzled_vec__ g() const;
__swizzled_vec__ b() const;
__swizzled_vec__ a() const;

__swizzled_vec__ s0() const;
__swizzled_vec__ s1() const;
__swizzled_vec__ s2() const;
__swizzled_vec__ s3() const;
__swizzled_vec__ s4() const;
__swizzled_vec__ s5() const;
__swizzled_vec__ s6() const;
__swizzled_vec__ s7() const;
__swizzled_vec__ s8() const;
__swizzled_vec__ s9() const;
__swizzled_vec__ sA() const;
__swizzled_vec__ sC() const;
__swizzled_vec__ sD() const;
__swizzled_vec__ sE() const;
__swizzled_vec__ sF() const;

__swizzled_vec__ lo() const;
__swizzled_vec__ hi() const;
__swizzled_vec__ odd() const;
__swizzled_vec__ even() const;
```

load

```
template <access::address_space addressSpace>
void load(size_t offset, multi_ptr<const dataT, addressSpace> ptr);
```

store

```
template <access::address_space addressSpace>
void load(size_t offset, multi_ptr<const dataT, addressSpace> ptr);
```

Arithmetic operators

```
friend vec operator+(const vec &lhs, const vec &rhs);
friend vec operator+(const vec &lhs, const dataT &rhs);
friend vec operator+(const dataT &lhs, const vec &rhs);

friend vec operator-(const vec &lhs, const vec &rhs);
friend vec operator-(const vec &lhs, const dataT &rhs);
friend vec operator-(const dataT &lhs, const vec &rhs);

friend vec operator*(const vec &lhs, const vec &rhs);
friend vec operator*(const vec &lhs, const dataT &rhs);
friend vec operator*(const dataT &lhs, const vec &rhs);

friend vec operator/(const vec &lhs, const vec &rhs);
friend vec operator/(const vec &lhs, const dataT &rhs);
friend vec operator/(const dataT &lhs, const vec &rhs);


friend vec &operator+=(vec &lhs, const vec &rhs);
friend vec &operator+=(vec &lhs, const dataT &rhs);

friend vec &operator-=(vec &lhs, const vec &rhs);
friend vec &operator-=(vec &lhs, const dataT &rhs);

friend vec &operator*=(vec &lhs, const vec &rhs);
friend vec &operator*=(vec &lhs, const dataT &rhs);

friend vec &operator/=(vec &lhs, const vec &rhs);
friend vec &operator/=(vec &lhs, const dataT &rhs);


friend vec &operator++(vec &lhs);
friend vec operator++(vec& lhs, int);

friend vec &operator--(vec &lhs);
friend vec operator--(vec& lhs, int);

friend vec<RET, numElements> operator&&(const vec &lhs, const vec &rhs);
friend vec<RET, numElements> operator&&(const vec& lhs, const dataT &rhs);
```

```
friend vec<RET, numElements> operator||(const vec &lhs, const vec &rhs);
friend vec<RET, numElements> operator||(const vec& lhs, const dataT &rhs);


friend vec<RET, numElements> operator==(const vec &lhs, const vec &rhs);
friend vec<RET, numElements> operator==(const vec &lhs, const dataT &rhs);
friend vec<RET, numElements> operator==(const dataT &lhs, const vec &rhs);

friend vec<RET, numElements> operator!=(const vec &lhs, const vec &rhs);
friend vec<RET, numElements> operator!=(const vec &lhs, const dataT &rhs);
friend vec<RET, numElements> operator!=(const dataT &lhs, const vec &rhs);

friend vec<RET, numElements> operator<(const vec &lhs, const vec &rhs);
friend vec<RET, numElements> operator<(const vec &lhs, const dataT &rhs);
friend vec<RET, numElements> operator<(const dataT &lhs, const vec &rhs);

friend vec<RET, numElements> operator>(const vec &lhs, const vec &rhs);
friend vec<RET, numElements> operator>(const vec &lhs, const dataT &rhs);
friend vec<RET, numElements> operator>(const dataT &lhs, const vec &rhs);

friend vec<RET, numElements> operator<=(const vec &lhs, const vec &rhs);
friend vec<RET, numElements> operator<=(const vec &lhs, const dataT &rhs);
friend vec<RET, numElements> operator<=(const dataT &lhs, const vec &rhs);

friend vec<RET, numElements> operator>=(const vec &lhs, const vec &rhs);
friend vec<RET, numElements> operator>=(const vec &lhs, const dataT &rhs);
friend vec<RET, numElements> operator>=(const dataT &lhs, const vec &rhs);

vec<dataT, numElements> &operator=(const vec<dataT, numElements> &rhs);
vec<dataT, numElements> &operator=(const dataT &rhs);

friend vec<RET, numElements> operator&&(const dataT &lhs, const vec &rhs);

friend vec<RET, numElements> operator||(const dataT &lhs, const vec &rhs);
```

Available only when:
dataT != cl_float && dataT != cl_double && dataT != cl_half

```
friend vec operator<<(const vec &lhs, const vec &rhs);
friend vec operator<<(const vec &lhs, const dataT &rhs);
friend vec operator<<(const dataT &lhs, const vec &rhs);
friend vec operator>>(const vec &lhs, const vec &rhs);
friend vec operator>>(const vec &lhs, const dataT &rhs);
friend vec operator>>(const dataT &lhs, const vec &rhs);
friend vec &operator>>=(vec &lhs, const vec &rhs);
friend vec &operator>>=(vec &lhs, const dataT &rhs);
friend vec &operator<<=(vec &lhs, const vec &rhs);
friend vec &operator<<=(vec &lhs, const dataT &rhs);
friend vec operator&(const vec &lhs, const vec &rhs);
friend vec operator&(const vec &lhs, const dataT &rhs);
friend vec operator|(const vec &lhs, const vec &rhs);
friend vec operator|(const vec &lhs, const dataT &rhs);
friend vec operator^(const vec &lhs, const vec &rhs);
```

```
friend vec operator^(const vec &lhs, const dataT &rhs);
friend vec &operator&=(vec &lhs, const vec &rhs);
friend vec &operator&=(vec &lhs, const dataT &rhs);
friend vec &operator|=(vec &lhs, const vec &rhs);
friend vec &operator|=(vec &lhs, const dataT &rhs);
friend vec &operator^=(vec &lhs, const vec &rhs);
friend vec &operator^=(vec &lhs, const dataT &rhs);
friend vec &operator%=(vec &lhs, const vec &rhs);
friend vec &operator%=(vec &lhs, const dataT &rhs);
friend vec operator%(const vec &lhs, const vec &rhs);
friend vec operator%(const vec &lhs, const dataT &rhs);
friend vec operator%(const dataT &lhs, const vec &rhs);
friend vec operator~(const vec &v);
friend vec<RET, numElements> operator!(const vec &v);
friend vec operator&(const dataT &lhs, const vec &rhs);
friend vec operator|(const dataT &lhs, const vec &rhs);
friend vec operator^(const dataT &lhs, const vec &rhs);
```

2.10 Synchronization and atomics

2.10.1 Synchronization types

access::fence_space

```
enum class fence_space : char {
    local_space,
    global_space,
    global_and_local
};
```

memory_order

```
enum class memory_order : int {
    relaxed
};
```

atomic

```
template <typename T,
          access::address_space addressSpace = access::address_space::global_space
         >
class atomic;
```

(constructors)

```
template <typename pointerT>
atomic(multi_ptr<pointerT, addressSpace> ptr);
```

store

```
void store(T operand, memory_order memoryOrder = memory_order::relaxed);
```

load

```
T load(memory_order memoryOrder = memory_order::relaxed) const;
```

exchange

```
T exchange(T operand, memory_order memoryOrder = memory_order::relaxed);
```

compare_exchange_strong

Available only when:
T != float

```
bool compare_exchange_strong(T &expected, T desired,
                             memory_order successMemoryOrder = memory_
↪order::relaxed,
                             memory_order failMemoryOrder = memory_
↪order::relaxed);
```

fetch_add

Available only when:
T != float

```
T fetch_add(T operand, memory_order memoryOrder = memory_order::relaxed);
```

fetch_sub

Available only when:
T != float

```
T fetch_sub(T operand, memory_order memoryOrder = memory_order::relaxed);
```

fetch_and

Available only when:
T != float

```
T fetch_and(T operand, memory_order memoryOrder = memory_order::relaxed);
```

fetch_or

Available only when:
T != float

```
T fetch_or(T operand, memory_order memoryOrder = memory_order::relaxed);
```

fetch_xor

Available only when:
T != float

```
T fetch_xor(T operand, memory_order memoryOrder = memory_order::relaxed);
```

fetch_min

Available only when:
T != float

```
T fetch_min(T operand, memory_order memoryOrder = memory_order::relaxed);
```

fetch_max

Available only when:
T != float

```
T fetch_max(T operand, memory_order memoryOrder = memory_order::relaxed);
```

2.11 IO

2.11.1 Streams

stream_manipulator

```
enum class stream_manipulator {
    flush,
    dec,
    hex,
    oct,
    noshowbase,
```

(continues on next page)

(continued from previous page)

```
    showbase,
    noshowpos,
    showpos,
    endl,
    fixed,
    scientific,
    hexfloat,
    defaultfloat
 };
```

Stream manipulators

```
const stream_manipulator flush = stream_manipulator::flush;
const stream_manipulator dec = stream_manipulator::dec;
const stream_manipulator hex = stream_manipulator::hex;
const stream_manipulator oct = stream_manipulator::oct;
const stream_manipulator noshowbase = stream_manipulator::noshowbase;
const stream_manipulator showbase = stream_manipulator::showbase;
const stream_manipulator noshowpos = stream_manipulator::noshowpos;
const stream_manipulator showpos = stream_manipulator::showpos;
const stream_manipulator endl = stream_manipulator::endl;
const stream_manipulator fixed = stream_manipulator::fixed;
const stream_manipulator scientific = stream_manipulator::scientific;
const stream_manipulator hexfloat = stream_manipulator::hexfloat;
const stream_manipulator defaultfloat = stream_manipulator::defaultfloat;
__precision_manipulator__ setprecision(int precision);
__width_manipulator__ setw(int width);
```

Stream Class

```
class stream;
```

(constructors)

```
stream(size_t totalBufferSize, size_t workItemBufferSize, handler& cgh);
```

get_size

```
size_t get_size() const;
```

get_work_item_buffer_size

```
size_t get_work_item_buffer_size() const;
```

get_max_statement_size

```
size_t get_max_statement_size() const;
```

get_max_statement_size() has the same functionality as get_work_item_buffer_size(), and is provided for backward compatibility. get_max_statement_size() is a deprecated query.

operator<<

```
template <typename T>
const stream& operator<<(const stream& os, const T &rhs);
```