



# SYCL<sup>TM</sup> 2020 Specification (revision 10)

The Khronos<sup>®</sup> SYCL<sup>TM</sup> Working Group

2025-04-07 12:39:14Z: commit 8324599656726d5501d61e6e0cd8f8ccd3781f85

# Table of Contents

1. Acknowledgements .....	14
2. Introduction .....	17
3. SYCL architecture .....	18
3.1. Overview .....	18
3.2. Anatomy of a SYCL application .....	18
3.3. Normative references .....	20
3.4. Non-normative notes and examples .....	20
3.5. The SYCL platform model .....	20
3.6. The SYCL backend model .....	21
3.6.1. Platform mixed version support .....	22
3.7. SYCL execution model .....	22
3.7.1. SYCL application execution model .....	22
3.7.1.1. Backend resources managed by the SYCL application .....	23
3.7.1.2. SYCL command groups and execution order .....	23
3.7.1.3. Controlling execution order with events .....	25
3.7.2. SYCL kernel execution model .....	25
3.7.2.1. Basic kernels .....	25
3.7.2.2. ND-range kernels .....	25
3.7.2.3. Backend-specific kernels .....	26
3.8. Memory model .....	26
3.8.1. SYCL application memory model .....	26
3.8.2. SYCL device memory model .....	29
3.8.2.1. Access to memory .....	30
3.8.3. SYCL memory consistency model .....	30
3.8.3.1. Memory ordering .....	31
3.8.3.2. Memory scope .....	32
3.8.3.3. Atomic operations .....	33
3.8.3.4. Forward progress .....	33
3.9. The SYCL programming model .....	33
3.9.1. Minimum version of C++ .....	33
3.9.2. Alignment with future versions of C++ .....	34
3.9.3. Basic data parallel kernels .....	34
3.9.4. Work-group data parallel kernels .....	34
3.9.5. Hierarchical data parallel kernels (deprecated) .....	35
3.9.6. Kernels that are not launched over parallel instances .....	35
3.9.7. Pre-defined kernels .....	35
3.9.8. Coordination and synchronization .....	35
3.9.8.1. Host-device coordination .....	35
3.9.8.2. Work-item coordination .....	36
3.9.9. Error handling .....	36
3.9.10. Fallback mechanism .....	37

3.9.11. Scheduling of kernels and data movement . . . . .	37
3.9.12. Managing object lifetimes . . . . .	37
3.9.13. Device discovery and selection . . . . .	38
3.9.14. Interfacing with the SYCL backend API . . . . .	38
3.10. Memory objects . . . . .	38
3.11. Multi-dimensional objects and linearization . . . . .	39
3.11.1. Linearization . . . . .	39
3.11.2. Multi-dimensional subscript operators . . . . .	40
3.12. Implementation options . . . . .	40
3.12.1. Single source multiple compiler passes . . . . .	40
3.12.2. Single source single compiler pass . . . . .	41
3.12.3. Library-only implementation . . . . .	41
3.13. Language restrictions in kernels . . . . .	41
3.13.1. Device copyable . . . . .	41
3.14. Endianness support . . . . .	42
3.15. Example SYCL application . . . . .	43
4. SYCL programming interface . . . . .	45
4.1. Backends . . . . .	45
4.1.1. Backend macros . . . . .	45
4.2. Generic vs non-generic SYCL . . . . .	45
4.3. Header files and namespaces . . . . .	46
4.4. Class availability . . . . .	46
4.5. Common interface . . . . .	46
4.5.1. Backend interoperability . . . . .	47
4.5.1.1. Type traits <code>backend_traits</code> . . . . .	47
4.5.1.2. Template function <code>get_native</code> . . . . .	48
4.5.1.3. Template functions <code>make_*</code> . . . . .	48
4.5.2. Common reference semantics . . . . .	50
4.5.3. Common by-value semantics . . . . .	52
4.5.4. Properties . . . . .	54
4.5.4.1. Properties interface . . . . .	54
4.6. SYCL runtime classes . . . . .	56
4.6.1. Device selection . . . . .	56
4.6.1.1. Device selector . . . . .	56
4.6.2. Platform class . . . . .	60
4.6.2.1. Constructors . . . . .	61
4.6.2.2. Member functions . . . . .	61
4.6.2.3. Static member functions . . . . .	62
4.6.2.4. Information descriptors . . . . .	62
4.6.3. Context class . . . . .	64
4.6.3.1. Constructors . . . . .	65
4.6.3.2. Member functions . . . . .	65
4.6.3.3. Information descriptors . . . . .	66

4.6.3.4. Properties .....	68
4.6.4. Device class .....	68
4.6.4.1. Constructors .....	69
4.6.4.2. Member functions .....	70
4.6.4.3. Static member functions .....	73
4.6.4.4. Information descriptors .....	73
4.6.4.5. Aspects .....	94
4.6.4.6. Aspect traits .....	97
4.6.4.7. Other enumerations .....	99
4.6.4.7.1. Device type .....	99
4.6.4.7.2. Partition property .....	99
4.6.4.7.3. Partition affinity domain .....	99
4.6.4.7.4. Floating point configuration .....	99
4.6.4.7.5. Local memory type .....	100
4.6.4.7.6. Global memory cache type .....	101
4.6.4.7.7. Execution capability .....	101
4.6.5. Queue class .....	101
4.6.5.1. Constructors .....	105
4.6.5.2. Member functions .....	108
4.6.5.3. Shortcut member functions .....	110
4.6.5.4. Information descriptors .....	115
4.6.5.5. Properties .....	116
4.6.5.6. Error handling .....	117
4.6.6. Event class .....	117
4.6.6.1. Event information and profiling descriptors .....	120
4.7. Data access and storage in SYCL .....	122
4.7.1. Host allocation .....	122
4.7.1.1. Default allocators .....	122
4.7.2. Buffers .....	123
4.7.2.1. Buffer interface .....	124
4.7.2.2. Buffer properties .....	142
4.7.2.3. Buffer destruction rules .....	143
4.7.3. Images .....	145
4.7.3.1. Unsampled image interface .....	146
4.7.3.2. Sampled image interface .....	159
4.7.3.3. Image properties .....	165
4.7.3.4. Image destruction rules .....	167
4.7.4. Sharing host memory with the SYCL data management classes .....	167
4.7.4.1. Default behavior .....	167
4.7.4.2. SYCL ownership of the host memory .....	167
4.7.4.3. Shared SYCL ownership of the host memory .....	168
4.7.5. Synchronization primitives .....	168
4.7.6. Accessors .....	169

4.7.6.1. Data type .....	170
4.7.6.2. Access modes .....	170
4.7.6.3. Deduction tags .....	171
4.7.6.4. Properties .....	171
4.7.6.5. Read only accessors .....	173
4.7.6.6. Accessing elements of an accessor .....	173
4.7.6.7. Container interface .....	173
4.7.6.8. Ranged accessors .....	174
4.7.6.9. Buffer accessor for commands .....	174
4.7.6.9.1. Interface for buffer command accessors .....	175
4.7.6.9.2. Deduction tags for buffer command accessors .....	186
4.7.6.9.3. Read only buffer command accessors and implicit conversions .....	186
4.7.6.9.4. Deprecated features of the <code>accessor</code> class .....	186
4.7.6.9.4.1. Aliased names .....	186
4.7.6.9.4.2. Discard access modes .....	187
4.7.6.9.4.3. Placeholder template parameter .....	187
4.7.6.9.4.4. Additional member functions for <code>target::device</code> specialization .....	187
4.7.6.9.4.5. Accessor specialization with <code>target::constant_buffer</code> .....	187
4.7.6.9.4.6. Accessor specialization with <code>target::host_buffer</code> .....	192
4.7.6.9.4.7. Accessor specialization with <code>target::local</code> .....	195
4.7.6.9.4.8. Common members for deprecated accessors .....	198
4.7.6.9.4.9. Accessor specialization with <code>access_mode::atomic</code> .....	201
4.7.6.10. Buffer accessor for host code .....	202
4.7.6.10.1. Interface for buffer host accessors .....	202
4.7.6.10.2. Deduction tags for buffer host accessors .....	208
4.7.6.10.3. Read only buffer host accessors and implicit conversions .....	208
4.7.6.11. Local accessor .....	208
4.7.6.11.1. Interface for local accessors .....	208
4.7.6.11.2. Read only local accessors and implicit conversions .....	212
4.7.6.12. Common members for buffer and local accessors .....	212
4.7.6.13. Unsamplerd image accessors .....	218
4.7.6.13.1. Interface for unsamplerd image accessors .....	219
4.7.6.13.2. Read only unsamplerd image accessors and implicit conversions .....	222
4.7.6.14. Samplerd image accessors .....	222
4.7.6.14.1. Interface for samplerd image accessors .....	222
4.7.6.14.2. Read only samplerd image accessors and implicit conversions .....	225
4.7.7. Address space classes .....	225
4.7.7.1. Multi-pointer class .....	225
4.7.7.2. Explicit pointer aliases .....	246
4.7.8. Image samplers .....	247
4.8. Unified shared memory (USM) .....	249
4.8.1. Unified addressing .....	250
4.8.2. Kinds of unified shared memory .....	250

4.8.3. USM allocations .....	253
4.8.3.1. C++ allocator interface .....	253
4.8.3.2. Device allocation functions .....	255
4.8.3.3. Host allocation functions .....	257
4.8.3.4. Shared allocation functions .....	259
4.8.3.5. Parameterized allocation functions .....	261
4.8.3.6. Memory deallocation functions .....	263
4.8.4. Unified shared memory pointer queries .....	264
4.9. Expressing parallelism through kernels .....	264
4.9.1. Ranges and index space identifiers .....	264
4.9.1.1. <code>range</code> class .....	265
4.9.1.2. <code>nd_range</code> class .....	269
4.9.1.3. <code>id</code> class .....	270
4.9.1.4. <code>item</code> class .....	274
4.9.1.5. <code>nd_item</code> class .....	276
4.9.1.6. <code>h_item</code> class (deprecated) .....	281
4.9.1.7. <code>group</code> class .....	284
4.9.1.8. <code>sub_group</code> class .....	291
4.9.2. Reduction variables .....	293
4.9.2.1. <code>reduction</code> interface .....	296
4.9.2.2. Reduction properties .....	298
4.9.2.3. <code>reducer</code> class .....	299
4.9.3. Command group scope .....	302
4.9.4. Command group <code>handler</code> class .....	303
4.9.4.1. SYCL functions for adding requirements .....	305
4.9.4.2. SYCL functions for invoking kernels .....	306
4.9.4.2.1. <code>single_task</code> invoke .....	313
4.9.4.2.2. <code>parallel_for</code> invoke .....	314
4.9.4.2.3. Parallel for hierarchical invoke (deprecated) .....	317
4.9.4.3. SYCL functions for explicit memory operations .....	320
4.9.4.4. Functions for using a kernel bundle .....	324
4.9.5. Specialization constants .....	325
4.9.5.1. Declaring a specialization constant .....	325
4.9.5.1.1. Constructors .....	327
4.9.5.1.2. Special member functions .....	327
4.9.5.2. Setting and getting the value of a specialization constant .....	328
4.9.5.3. Reading the value of a specialization constant from device code .....	328
4.9.5.3.1. Member functions .....	329
4.9.5.4. Example usage .....	329
4.10. Host tasks .....	330
4.10.1. Overview .....	330
4.10.2. Class <code>interop_handle</code> .....	332
4.10.2.1. Constructors .....	332

4.10.2.2. Member functions .....	332
4.10.2.3. Template member functions <code>get_native_*</code> .....	332
4.10.3. Additions to the <code>handler</code> class .....	334
4.11. Kernel bundles .....	334
4.11.1. Overview .....	335
4.11.2. Synopsis .....	336
4.11.3. Fixed-function built-in kernels .....	338
4.11.4. Bundle states .....	338
4.11.5. Kernel identifiers .....	339
4.11.6. Obtaining a kernel identifier .....	340
4.11.7. Obtaining a kernel bundle .....	340
4.11.8. Querying if a kernel bundle exists .....	342
4.11.9. Querying if a kernel is compatible with a device .....	344
4.11.10. Joining kernel bundles .....	344
4.11.11. Online compiling and linking .....	345
4.11.12. The <code>kernel_bundle</code> class .....	347
4.11.12.1. Queries .....	348
4.11.12.2. Specialization constant support .....	349
4.11.12.3. Device image support .....	351
4.11.13. The <code>kernel</code> class .....	351
4.11.13.1. Queries .....	351
4.11.13.2. Kernel information descriptors .....	352
4.11.14. The <code>device_image</code> class .....	354
4.11.15. Example usage .....	355
4.11.15.1. Controlling the timing of online compilation .....	355
4.11.15.2. Specialization constants .....	355
4.11.15.3. Kernel introspection .....	356
4.11.15.4. Invoking a device built-in kernel .....	357
4.12. Defining kernels .....	358
4.12.1. Defining kernels as named function objects .....	358
4.12.2. Defining kernels as lambda expressions .....	359
4.12.3. <code>is_device_copyable</code> type trait .....	361
4.12.4. Rules for parameter passing to kernels .....	361
4.13. Error handling .....	362
4.13.1. Error handling rules .....	362
4.13.1.1. Asynchronous error handler .....	362
4.13.1.2. Behavior without an async handler .....	363
4.13.1.3. Priorities of async handlers .....	363
4.13.1.4. Asynchronous errors with a secondary queue .....	363
4.13.2. Exception class interface .....	364
4.14. Data types .....	369
4.14.1. Scalar data types .....	369
4.14.2. Vector types .....	370

4.14.2.1. Vec interface .....	370
4.14.2.2. Aliases .....	392
4.14.2.3. Swizzles .....	392
4.14.2.4. The swizzled vector classes .....	393
4.14.2.4.1. Member type aliases for the swizzled vector class templates .....	394
4.14.2.4.2. Constructors for the swizzled vector class templates .....	394
4.14.2.4.3. Destructors for the swizzled vector class templates .....	395
4.14.2.4.4. Member functions for the swizzled vector class templates .....	395
4.14.2.4.5. Hidden friend functions of the swizzled vector class templates .....	399
4.14.2.5. Rounding modes .....	403
4.14.2.6. Memory layout and alignment .....	404
4.14.2.7. Performance note .....	404
4.14.3. Math array types .....	404
4.14.3.1. Math array interface .....	404
4.14.3.2. Aliases .....	416
4.14.3.3. Memory layout and alignment .....	416
4.15. Synchronization and atomics .....	416
4.15.1. Barriers and fences .....	417
4.15.2. <code>device_event</code> class .....	417
4.15.3. Atomic references .....	418
4.15.4. Atomic types (deprecated) .....	429
4.15.5. Interaction with host code .....	438
4.16. Stream class .....	439
4.16.1. Stream class interface .....	439
4.16.2. Output .....	443
4.16.3. Implicit flush .....	444
4.16.4. Performance note .....	444
4.17. SYCL built-in functions for SYCL host and device .....	444
4.17.1. Function objects .....	444
4.17.2. Group functions .....	446
4.17.2.1. Group type trait .....	447
4.17.2.2. <code>group_broadcast</code> .....	447
4.17.2.3. <code>group_barrier</code> .....	448
4.17.3. Group algorithms library .....	448
4.17.3.1. <code>any_of</code> , <code>all_of</code> and <code>none_of</code> .....	449
4.17.3.2. <code>shift_left</code> and <code>shift_right</code> .....	452
4.17.3.3. <code>permute</code> .....	453
4.17.3.4. <code>select</code> .....	453
4.17.3.5. <code>reduce</code> .....	454
4.17.3.6. <code>exclusive_scan</code> and <code>inclusive_scan</code> .....	456
4.17.4. Math functions .....	459
4.17.5. Native precision math functions .....	497
4.17.6. Half precision math functions .....	503



4.17.7. Integer functions . . . . .	510
4.17.8. Common functions . . . . .	521
4.17.9. Geometric functions . . . . .	527
4.17.10. Relational functions . . . . .	530
5. SYCL Device Compiler . . . . .	546
5.1. Offline compilation of SYCL source files . . . . .	546
5.2. Naming of kernels . . . . .	546
5.3. Compilation of functions . . . . .	547
5.4. Language restrictions for device functions . . . . .	547
5.5. Built-in scalar data types . . . . .	548
5.6. Preprocessor directives and macros . . . . .	550
5.7. Optional kernel features . . . . .	550
5.8. Attributes for device code . . . . .	551
5.8.1. Kernel attributes . . . . .	552
5.8.2. Device function attributes . . . . .	557
5.9. Address-space deduction . . . . .	558
5.9.1. Address space assignment . . . . .	559
5.9.2. Common address space deduction rules . . . . .	559
5.9.3. Generic as default address space . . . . .	560
5.9.4. Inferred address space . . . . .	560
5.10. SYCL offline linking . . . . .	560
5.10.1. SYCL functions and member functions linkage . . . . .	560
6. SYCL Extensions . . . . .	562
6.1. Definition of an extension . . . . .	562
6.2. Requirements for an extension . . . . .	562
6.3. Guidelines for portable extensions . . . . .	563
6.3.1. Extension namespace . . . . .	563
6.3.2. Names for extensions to existing classes or enumerations . . . . .	563
6.3.3. Feature test macros . . . . .	563
6.3.4. Attribute namespace . . . . .	564
6.3.5. Include file paths . . . . .	564
6.3.6. Optional kernel features . . . . .	564
6.3.7. Adding a backend . . . . .	564
Appendix A: Information descriptors . . . . .	565
A.1. Platform information descriptors . . . . .	565
A.2. Context information descriptors . . . . .	565
A.3. Device information descriptors . . . . .	565
A.4. Queue information descriptors . . . . .	568
A.5. Kernel information descriptors . . . . .	568
A.6. Event information descriptors . . . . .	569
Appendix B: Feature sets . . . . .	570
B.1. Full feature set . . . . .	570
B.2. Reduced feature set . . . . .	570

B.3. Compatibility .....	570
B.4. Conformance .....	570
Appendix C: OpenCL backend specification .....	571
C.1. SYCL application interoperability native backend objects .....	571
C.2. Kernel function interoperability native backend objects .....	571
C.3. Destruction of interop constructed objects with reference semantics .....	572
C.4. SYCL for OpenCL framework .....	572
C.5. Mapping of SYCL programming model on top of OpenCL .....	573
C.5.1. Backend specific information descriptors .....	573
C.5.2. OpenCL memory model .....	573
C.5.3. OpenCL interface for buffer command accessors .....	574
C.5.4. OpenCL resources managed by SYCL application .....	574
C.6. Interoperability with the OpenCL API .....	574
C.7. Programming interface .....	577
C.7.1. Construct SYCL objects from OpenCL ones .....	577
C.7.2. Extension query .....	579
C.7.3. Reference counting .....	580
C.7.4. Errors and limitations .....	580
C.7.5. Interoperability with kernel bundles .....	580
C.7.6. Interoperability with kernels .....	581
C.7.7. OpenCL kernel conventions and SYCL .....	582
C.7.8. Data types .....	583
C.8. Preprocessor directives and macros .....	584
C.8.1. Offline linking with OpenCL C libraries .....	584
C.9. SYCL support of non-core OpenCL features .....	584
C.9.1. Half precision floating-point .....	585
C.9.2. Writing to 3D image memory objects .....	585
C.9.3. Interoperability with OpenGL .....	585
C.10. Correspondence of some OpenCL features to SYCL .....	585
C.10.1. Work-item functions .....	585
C.10.2. Vector data load and store functions .....	585
C.10.3. Synchronization functions .....	586
C.10.4. <code>printf</code> function .....	586
C.11. Precision of built-in math functions .....	586
Appendix D: What has changed from previous versions .....	587
D.1. What has changed from SYCL 1.2.1 to SYCL 2020 .....	587
Appendix E: References .....	593
Appendix F: Optional extensions .....	594
F.1. <code>sycl_khr_default_context</code> .....	594
F.1.1. Dependencies .....	594
F.1.2. Feature test macro .....	594
F.1.3. Extensions to the platform class .....	594
F.1.3.1. Member functions .....	594

---

Glossary ..... 595

Copyright 2011-2025 The Khronos Group Inc.

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

Khronos grants a conditional copyright license to use and reproduce the unmodified Specification for any purpose, without fee or royalty, EXCEPT no licenses to any patent, trademark or other intellectual property rights are granted under these terms.

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

This Specification has been created under the Khronos Intellectual Property Rights Policy, which is Attachment A of the Khronos Group Membership Agreement available at [https://www.khronos.org/files/member\\_agreement.pdf](https://www.khronos.org/files/member_agreement.pdf). Parties desiring to implement the Specification and make use of Khronos trademarks in relation to that implementation, and receive reciprocal patent license protection under the Khronos Intellectual Property Rights Policy must become Adopters and confirm the implementation as conformant under the process defined by Khronos for this Specification; see <https://www.khronos.org/adopters>.

The Khronos Intellectual Property Rights Policy defines the terms 'Scope', 'Compliant Portion', and 'Necessary Patent Claims'.

Some parts of this Specification are purely informative and so are EXCLUDED from the Scope of this Specification. [Section 3.4](#) defines how these parts of the Specification are identified.

Where this Specification uses technical terminology, defined in the Glossary or otherwise, that refer to enabling technologies that are not expressly set forth in this Specification, those enabling technologies are EXCLUDED from the Scope of this Specification. For clarity, enabling technologies not disclosed with particularity in this Specification (e.g. semiconductor manufacturing technology, hardware architecture, processor architecture or microarchitecture, memory architecture, compiler technology, object oriented technology, basic operating system technology, compression technology, algorithms, and so on) are NOT to be considered expressly set forth; only those application program interfaces and data structures disclosed with particularity are included in the Scope of this Specification.

For purposes of the Khronos Intellectual Property Rights Policy as it relates to the definition of Necessary Patent Claims, all recommended or optional features, behaviors and functionality set forth in this Specification, if implemented, are considered to be included as Compliant Portions.

Where this Specification identifies specific sections of external references, only those specifically identified sections define normative functionality. The Khronos Intellectual Property Rights Policy excludes external references to materials and associated enabling technology not created by Khronos from the Scope of this Specification, and any licenses that may be required to implement such referenced materials and associated technologies must be obtained separately and may involve royalty payments.

Khronos® and Vulkan® are registered trademarks, and 3D Commerce™, ANARI™, Kamaros™, KTX™, glTF™, NNEF™, OpenVG™, OpenVX™, SPIR™, SPIR-V™, SYCL™, Vulkan SC™, and WebGL™ are trademarks of The Khronos Group Inc. OpenXR™ is a trademark owned by The Khronos Group Inc. and is registered as a trademark in China, the European Union, Japan and the United Kingdom. OpenCL™ is a trademark of Apple Inc. used under license by Khronos. OpenGL® is a registered trademark and the OpenGL ES™ and OpenGL SC™ logos are trademarks of Hewlett Packard Enterprise used under license by Khronos.

ASTC is a trademark of ARM Holdings PLC. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.

# Chapter 1. Acknowledgements

## Editors

- Maria Rovatsou, Codeplay
- Lee Howes, Qualcomm
- Ronan Keryell, AMD
- Greg Lueck, Intel (current)

## Contributors

- Eric Berdahl, Adobe
- Shivani Gupta, Adobe
- David Neto, Altera
- Carlo Bertolli, AMD
- Andrew Gozillon, AMD
- Gauthier Harnisch, AMD
- Ronan Keryell, AMD
- Yiannis Papadopoulos, AMD
- Brian Sumner, AMD
- Lin-Ya Yu, AMD
- Thomas Applencourt, Argonne National Laboratory
- Hal Finkel, Argonne National Laboratory
- Kevin Harms, Argonne National Laboratory
- Nevin Liber, Argonne National Laboratory
- Anastasia Stulova, ARM
- Balázs Keszthelyi, Broadcom
- Alexandra Crabb, Caster Communications
- Stuart Adams, Codeplay
- Verena Beckham, Codeplay
- Aidan Belton, Codeplay
- Gordon Brown, Codeplay
- Hugh Delaney, Codeplay
- Atharva Dubey, Codeplay
- Morris Hafner, Codeplay
- Alexander Johnston, Codeplay
- Marios Katsigiannis, Codeplay
- Paul Keir, Codeplay
- Steffen Larsen, Codeplay
- Victor Lomüller, Codeplay
- Tomas Matheson, Codeplay
- Duncan McBain, Codeplay
- Nicolas Miller, Codeplay

- Georgi Mirazchiyski, Codeplay
- Mahmoud Moadeli, Codeplay
- Ralph Potter, Codeplay
- Ruyman Reyes, Codeplay
- Andrew Richards, Codeplay
- Maria Rovatsou, Codeplay
- Panagiotis Stratis, Codeplay
- Erik Tomusk, Codeplay
- Michael Wong, Codeplay
- Peter Žužek, Codeplay
- Matt Newport, EA
- Rasool Maghareh, Huawei Technologies Co. Ltd.
- Guansong Zhang, Huawei Technologies Co. Ltd.
- Ruslan Arutyunyan, Intel
- Alexey Bader, Intel
- James Brodman, Intel
- Ilya Burylov, Intel
- Jessica Davies, Intel
- Andrei Elovikov, Intel
- Felipe de Azevedo Piovezan, Intel
- Allen Hux, Intel
- Michael Kinsner, Intel
- Nikita Kornev, Intel
- Greg Lueck, Intel
- John Pennycook, Intel
- Roland Schulz, Intel
- Sergey Semenov, Intel
- Jason Sewall, Intel
- James O’Riordon, Khronos
- Jon Leech, Luna Princeps LLC
- Kathleen Mattson, Miller & Mattson, LLC
- Dave Miller, Miller & Mattson, LLC
- Stéphanie Even, Mercedes-Benz Research and Development NA
- Chris Gearing, Mobileye
- Seiji Nishimura, NSITEXE, Inc.
- Neil Trevett, NVIDIA
- Lee Howes, Qualcomm
- Chu-Cheow Lim, Qualcomm
- Jack Liu, Qualcomm
- Hongqiang Wang, Qualcomm
- Ruihao Zhang, Qualcomm

- Dave Airlie, Red Hat
- Hyesun Hong, Samsung Electronics
- Aksel Alpay, Self
- Dániel Berényi, Self
- Nuno Nobre, STFC Hartree Centre
- Máté Nagy-Egri, Stream HPC
- Bálint Soproni, Stream HPC
- Tom Deakin, University of Bristol
- Philip Salzmann, University of Innsbruck
- Peter Thoman, University of Innsbruck
- Biagio Cosenza, University of Salerno
- Paul Preney, University of Windsor



## Chapter 2. Introduction

SYCL (pronounced “sickle”) is a royalty-free, cross-platform API for heterogeneous computing in C++.

SYCL enables developers to write standard C++ code that executes on a wide range of devices, using modern techniques such as inheritance, templates, and lambda expressions. All computational kernels to be executed on a device can be written inside C++ source files as normal C++ code, alongside any code intended to be run on a system’s host processor. This concept, known as “single-source” programming, reduces the complexity of heterogeneous programming for developers and gives compilers greater opportunities to analyze/optimize across the host-device boundary.

SYCL is designed to be as close to standard C++ as possible, and some implementations of SYCL may be able to use a standard C++ compiler to target CPU devices. However, to ensure portability of device code across a wide range of devices, SYCL imposes some restrictions on the set of C++ features that SYCL implementations are required to support within device code. These restrictions may not be applicable to all devices and can therefore be relaxed by specific Khronos extensions or vendor extensions.

SYCL was originally based on OpenCL, and retains an execution model, runtime feature set, and device capability set inspired by the OpenCL standard. However, there is no requirement that SYCL implementations must use OpenCL; SYCL implementations are free to support devices via any low-level API (or “backend”) they choose.

Some of the key features of SYCL are:

- Common parallel patterns, such as [reductions](#) and [group algorithms](#), are exposed via high-level abstractions.
- Interoperability with the lower-level capabilities of specific [backends](#) guarantees access to platform-specific optimizations.
- [Buffers](#) and [accessors](#) provide a simple way to build task-graphs without manually managing dependencies.
- [Unified Shared Memory](#) (USM) provides an explicit, pointer-based, mechanism for managing and sharing data.

SYCL has been designed to enable implementations on a wide variety of platforms, permitting easy integration with other platform-specific technologies. Both users and implementers are encouraged to build upon SYCL as an open platform for system-wide heterogeneous programming.

# Chapter 3. SYCL architecture

This chapter describes the structure of a SYCL application, and how the SYCL generic programming model lays out on top of a number of [SYCL backends](#).

## 3.1. Overview

SYCL is an open industry standard for programming a heterogeneous system. The design of SYCL allows standard C++ source code to be written such that it can run on either an heterogeneous device or on the [host](#).

The terminology used for SYCL inherits historically from OpenCL with some SYCL-specific additions. However SYCL is a generic C++ programming model that can be laid out on top of other APIs apart from OpenCL. SYCL implementations can provide [SYCL backends](#) for various APIs, implementing the SYCL general specification on top of them. We refer to this API as the [SYCL backend API](#). The SYCL general specification defines the behavior that all SYCL implementations must expose to SYCL users for a SYCL application to behave as expected.

A function object that can execute on a [device](#) exposed by a [SYCL backend API](#) is called a [SYCL kernel function](#).

To ensure maximum interoperability with different [SYCL backend APIs](#), software developers can access the [SYCL backend API](#) alongside the SYCL general API whenever they include the [SYCL backend](#) interoperability headers. However, interoperability is a [SYCL backend](#)-specific feature. An application that uses interoperability does not conform to the SYCL general application model, since it is not portable across backends.

The target users of SYCL are C++ programmers who want all the performance and portability features of a standard like OpenCL, but with the flexibility to use higher-level C++ abstractions across the host/device code boundary. Developers can use most of the abstraction features of C++, such as templates, classes and operator overloading.

However, some C++ language features are not permitted inside kernels, due to the limitations imposed by the capabilities of the underlying heterogeneous platforms. These features include virtual functions, virtual inheritance, throwing/catching exceptions, and run-time type-information. These features are available outside kernels as normal. Within these constraints, developers can use abstractions defined by SYCL, or they can develop their own on top. These capabilities make SYCL ideal for library developers, middleware providers and application developers who want to separate low-level highly-tuned algorithms or data structures that work on heterogeneous systems from higher-level software development. Software developers can produce templated algorithms that are easily usable by developers in other fields.

## 3.2. Anatomy of a SYCL application

Below is an example of a typical [SYCL application](#) which schedules a job to run in parallel on any device available.

```
1 #include <iostream>
2 #include <sycl/sycl.hpp>
3 using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names
4
5 int main() {
6     int data[1024]; // Allocate data to be worked on
7
8     // Create a default queue to enqueue work to the default device
```

```

9  queue myQueue;
10
11 // By wrapping all the SYCL work in a {} block, we ensure
12 // all SYCL tasks must complete before exiting the block,
13 // because the destructor of resultBuf will wait
14 {
15     // Wrap our data variable in a buffer
16     buffer<int, 1> resultBuf{data, range<1>{1024}};
17
18     // Create a command group to issue commands to the queue
19     myQueue.submit([&](handler& cgh) {
20         // Request write access to the buffer without initialization
21         accessor writeResult{resultBuf, cgh, write_only, no_init};
22
23         // Enqueue a parallel_for task with 1024 work-items
24         cgh.parallel_for(1024, [=](id<1> idx) {
25             // Initialize each buffer element with its own rank number starting at 0
26             writeResult[idx] = idx;
27         }); // End of the kernel function
28     }); // End of our commands for this queue
29 } // End of scope, so we wait for work producing resultBuf to complete
30
31 // Print result
32 for (int i = 0; i < 1024; i++)
33     std::cout << "data[" << i << "] = " << data[i] << std::endl;
34
35 return 0;
36 }

```

At line 1, we `#include` the SYCL header files, which provide all of the SYCL features that will be used.

A SYCL application runs on a [SYCL Platform](#). The application is structured in three scopes which specify the different sections; [application scope](#), [command group scope](#) and [kernel scope](#). The [kernel scope](#) specifies a single kernel function that will be, or has been, compiled by a [device compiler](#) and executed on a [device](#). In this example [kernel scope](#) is defined by lines 25 to 26. The [command group scope](#) specifies a unit of work which is comprised of a [SYCL kernel function](#) and [accessors](#). In this example [command group scope](#) is defined by lines 20 to 28. The [application scope](#) specifies all other code outside of a [command group scope](#). These three scopes are used to control the application flow and the construction and lifetimes of the various objects used within SYCL, as explained in [Section 3.9.12](#).

A [SYCL kernel function](#) is the scoped block of code that will be compiled using a device compiler. This code may be defined by the body of a lambda expression or by the `operator()` function of a function object. Each instance of the [SYCL kernel function](#) will be executed as a single, though not necessarily entirely independent, flow of execution and has to adhere to restrictions on what operations may be allowed to enable device compilers to safely compile it to a range of underlying devices.

The `parallel_for` member function can be templated with a class. This class is used to manually name the kernel when desired, such as to avoid a compiler-generated name when debugging a kernel defined through a lambda, to provide a known name with which to apply build options to a kernel, or to ensure compatibility with multiple compiler-pass implementations.

The `parallel_for` member function creates an instance of a [kernel](#), which is the entity that will be enqueued within a command group. In the case of `parallel_for` the [SYCL kernel function](#) will be executed over the given range from 0 to 1023. The different member functions to execute kernels can be found in [Section 4.9.4.2](#).

A [command group scope](#) is the syntactic scope wrapped by the construction of a [command group function object](#) as seen on line 19. The [command group function object](#) may invoke only a single [SYCL kernel function](#), and it takes a parameter of type [command group handler](#), which is constructed by the runtime.

All the requirements for a kernel to execute are defined in this [command group scope](#), as described in [Section 3.7.1](#). In this case the constructor used for `myQueue` on line 9 is the default constructor, which allows the queue to select the best underlying device to execute on, leaving the decision up to the runtime.

In SYCL, data that is required within a [SYCL kernel function](#) must be contained within a [buffer](#), [image](#), or [USM](#) allocation, as described in [Section 3.8](#). We construct a buffer on line 16. Access to the [buffer](#) is controlled via an [accessor](#) which is constructed on line 21. The [buffer](#) is used to keep track of access to the data and the [accessor](#) is used to request access to the data on a queue, as well as to track the dependencies between [SYCL kernel function](#). In this example the [accessor](#) is used to write to the data buffer on line 26.

### 3.3. Normative references

The documents in the following list are referred to within this SYCL specification, and their content is a requirement for this document.

1. **C++17: ISO/IEC 14882:2017 Clauses 1-19**, referred to in this specification as the C++ core language. The SYCL specification refers to language in the following C++ defect reports and assumes a compiler that implements them: [DR2325](#).
2. **C++20: ISO/IEC 14882:2020 Programming languages — C++**, referred to in this specification as the next C++ specification.

### 3.4. Non-normative notes and examples

Unless stated otherwise, text within this SYCL specification is normative and defines the required behavior of a SYCL implementation. Non-normative / informational notes are included within this specification using either of two formats. One format for non-normative notes is the “note” callout of this form:



Information within a note callout, such as this text, is for informational purposes and does not impose requirements on or specify behavior of a SYCL implementation.

The other format for a non-normative note is like this:

[*Note*: This is also a non-normative note. — *end note*]

Source code examples within the specification are provided to aid with understanding, and are non-normative.

In case of any conflict between a non-normative note or source example, and normative text within the specification, the normative text must be taken to be correct.

### 3.5. The SYCL platform model

The SYCL platform model consists of a host connected to one or more devices, called [devices](#). [Devices](#) are grouped together into one or multiple [platforms](#). An implementation may also expose empty [platforms](#) that do not contain any [devices](#).

A SYCL [context](#) is constructed, either directly by the user or implicitly when creating a [queue](#), to hold all the runtime information required by the SYCL runtime and the [SYCL backend](#) to operate on a device, or group of devices. When a group of devices can be grouped together on the same context, they have some visibility of each other’s memory objects. The SYCL runtime can assume that memory is visible across all

devices in the same [context](#). Not all devices exposed from the same [platform](#) can be grouped together in the same [context](#).

A SYCL application executes on the host as a standard C++ program. [Devices](#) are exposed through different [SYCL backends](#) to the SYCL application. The SYCL application submits [command group function objects](#) to [queues](#). Each [queue](#) enables execution on a given device.

The [SYCL runtime](#) then extracts operations from the [command group function object](#), e.g. an explicit copy operation or a [SYCL kernel function](#). When the operation is a [SYCL kernel function](#), the [SYCL runtime](#) uses a [SYCL backend](#)-specific mechanism to extract the device binary from the SYCL application and pass it to the [SYCL backend API](#) for execution on the [device](#).

A SYCL [device](#) is divided into one or more compute units (CUs) which are each divided into one or more processing elements (PEs). Computations on a device occur within the processing elements. How computation is mapped to PEs is [SYCL backend](#) and [device](#) specific. Two devices exposed via two different backends can map computations differently to the same device.

When a SYCL application contains [SYCL kernel function](#) objects, the SYCL implementation must provide an offline compilation mechanism that enables the integration of the device binaries into the SYCL application. The output of the offline compiler can be an intermediate representation, such as SPIR-V, that will be finalized during execution or a final device ISA.

A device may expose special purpose functionality as a *built-in* function. The SYCL API exposes functions to query and dispatch said *built-in* functions. Some [SYCL backends](#) and [devices](#) may not support programmable kernels, and only support *built-in* functions.

## 3.6. The SYCL backend model

SYCL is a generic programming model for the C++ language that can target multiple APIs, such as OpenCL.

SYCL implementations enable these target APIs by implementing [SYCL backends](#). For a SYCL implementation to be conformant on said [SYCL backend](#), it must execute the SYCL generic programming model on the backend. All SYCL implementations must provide at least one backend.

The present document covers the SYCL generic interface available to all [SYCL backends](#). How the SYCL generic interface maps to a particular [SYCL backend](#) is defined either by a separate [SYCL backend](#) specification document, provided by the Khronos SYCL group, or by the SYCL implementation documentation. Whenever there is a [SYCL backend](#) specification document, this takes precedence over SYCL implementation documentation.

When a SYCL user builds their SYCL application, she decides which of the [SYCL backends](#) will be used to build the SYCL application. This is called the set of *active backends*. Implementations must ensure that the active backends selected by the user can be used simultaneously by the SYCL implementation at runtime. If two backends are available at compile time but will produce an invalid SYCL application at runtime, the SYCL implementation must emit a compilation error.

A SYCL application built with a number of active backends does not necessarily guarantee that said backends can be executed at runtime. The subset of active backends available at runtime is called *available backends*. A backend is said to be *available* if the host platform where the SYCL application is executed exposes support for the API required for the [SYCL backend](#).

It is implementation dependent whether certain backends require third-party libraries to be available in the system. Failure to have all dependencies required for all active backends at runtime will cause the SYCL application to not run.

Once the application is running, users can query what SYCL platforms are available. SYCL implementations will expose the devices provided by each backend grouped into platforms. A backend must expose



at least one platform.

Under the [SYCL backend](#) model, SYCL objects can contain one or multiple references to a certain [SYCL backend](#) native type. Not all SYCL objects will map directly to a [SYCL backend](#) native type. The mapping of SYCL objects to [SYCL backend](#) native types is defined by the [SYCL backend](#) specification document when available, or by the SYCL implementation otherwise.

To guarantee that multiple [SYCL backend](#) objects can interoperate with each other, SYCL memory objects are not bound to a particular [SYCL backend](#). SYCL memory objects can be accessed from any device exposed by an *available* backend. SYCL Implementations can potentially map SYCL memory objects to multiple native types in different [SYCL backends](#).

Since SYCL memory objects are independent of any particular [SYCL backend](#), SYCL [command groups](#) can request access to memory objects allocated by any [SYCL backend](#), and execute it on the backend associated with the [queue](#). This requires the SYCL implementation to be able to transfer memory objects across [SYCL backends](#).

USM allocations are subject to the limitations described in [Section 4.8](#).

When a SYCL application runs on any number of [SYCL backends](#) without relying on any [SYCL backend](#)-specific behavior or interoperability, it is said to be a SYCL general application, and it is expected to run in any SYCL-conformant implementation that supports the required features for the application.

### 3.6.1. Platform mixed version support

The SYCL generic programming model exposes a number of [platforms](#), each of them either empty or exposing a number of [devices](#). Each [platform](#) is bound to a certain [SYCL backend](#). SYCL [devices](#) associated with said [platform](#) are associated with that [SYCL backend](#).

Although the APIs in the SYCL generic programming model are defined according to this specification and their version is indicated by the macro `SYCL_LANGUAGE_VERSION`, this does not apply to APIs exposed by the [SYCL backends](#). Each [SYCL backend](#) provides its own document that defines its APIs, and that document tells how to query for the device and platform versions.

## 3.7. SYCL execution model

As described in [Section 3.2](#), a [SYCL application](#) is comprised of three scopes: [application scope](#), [command group scope](#), and [kernel scope](#). Code in the [application scope](#) and [command group scope](#) runs on the host and is governed by the *SYCL application execution model*. Code in the kernel scope runs on a device and is governed by the *SYCL kernel execution model*.



A SYCL device does not necessarily correspond to a physical accelerator. A SYCL implementation may choose to expose some or all of the host's resources as a SYCL device; such an implementation would execute code in [kernel scope](#) on the host, but that code would still be governed by the *SYCL kernel execution model*.

### 3.7.1. SYCL application execution model

The SYCL application defines the execution order of the kernels by grouping each kernel with its requirements into a [command group function object](#). [Command group function objects](#) are submitted for execution via a [queue](#) object, which defines the device where the kernel will run. This specification sometimes refers to this as “submitting the kernel to a device”. The same [command group](#) object can be submitted to different queues. When a [command group](#) is submitted to a SYCL [queue](#), the requirements of the kernel execution are captured. The implementation can start executing a kernel as soon as its requirements have been satisfied.

### 3.7.1.1. Backend resources managed by the SYCL application

The SYCL runtime integrated with the SYCL application will manage the resources required by the [SYCL backend API](#) to manage the devices it is providing access to. This includes, but is not limited to, resource handlers, memory pools, dispatch queues and other temporary handler objects.

The SYCL programming interface represents the lifetime of the resources managed by the SYCL application using RAII rules. Construction of a SYCL object will typically entail the creation of multiple [SYCL backend](#) objects, which will be properly released on destruction of said SYCL object. The overall rules for construction and destruction are detailed in [Chapter 4](#). Those [SYCL backends](#) with a [SYCL backend](#) document will detail how the resource management from SYCL objects map down to the [SYCL backend](#) objects.

In SYCL, the minimum required object for submitting work to devices is the [queue](#), which contains references to a [platform](#), [device](#) and a [context](#) internally.

The resources managed by SYCL are:

1. **Platforms:** all features of [SYCL backend APIs](#) are implemented by platforms. A platform can be viewed as a given vendor's runtime and the devices accessible through it. Some devices will only be accessible to one vendor's runtime and hence multiple platforms may be present. SYCL manages the different platforms for the user which are accessible through a `sycl::platform` object. In some cases, an implementation might also choose to expose empty `sycl::platform` objects, for example if a vendor's runtime is available, but no devices supported by that runtime are available in the system.
2. **Contexts:** any [SYCL backend](#) resource that is acquired by the user is attached to a context. A context contains a collection of devices that the host can use and manages memory objects that can be shared between the devices. Devices belonging to the same [context](#) must be able to access each other's global memory using some implementation-specific mechanism. A given context can only wrap devices owned by a single platform. A context is exposed to the user with a `sycl::context` object.
3. **Devices:** platforms may provide devices for executing SYCL kernels. In SYCL, a device is accessible through a `sycl::device` object.
4. **Kernels:** the SYCL functions that run on SYCL devices are defined as C++ function objects (a named function object type or a lambda expression). A kernel can be introspected through a `sycl::kernel` object.

Note that some [SYCL backends](#) may expose non-programmable functionality as pre-defined kernels.

5. **Kernel bundles:** Kernels are stored internally in the SYCL application as device images, and these device images can be grouped into a `sycl::kernel_bundle` object. These objects provide a way for the application to control the online compilation of kernels for devices.
6. **Queues:** SYCL kernels execute in command queues. The user must create a `sycl::queue` object, which references an associated context, platform and device. The context, platform and device may be chosen automatically, or specified by the user. SYCL queues execute [kernels](#) on a particular device of a particular context, but can have dependencies from any device on any available [SYCL backend](#).

The SYCL implementation guarantees the correct initialization and destruction of any resource handled by the underlying [SYCL backend API](#), except for those the user has obtained manually via the SYCL interoperability API.

### 3.7.1.2. SYCL command groups and execution order

By default, SYCL queues execute kernel functions in an out-of-order fashion based on dependency information. Developers only need to specify what data is required to execute a particular kernel. The SYCL runtime will guarantee that kernels are executed in an order that guarantees correctness. By specifying access modes and types of memory, a directed acyclic dependency graph (DAG) of kernels is built at runtime. This is achieved via the usage of [command group](#) objects. A SYCL [command group](#) object defines a

set of requisites ( $R$ ) and a kernel function ( $k$ ). A **command group** is *submitted* to a queue when using the `sycl::queue::submit` member function.

A **requisite** ( $r_i$ ) is a requirement that must be fulfilled for a kernel-function ( $k$ ) to be executed on a particular device. For example, a requirement may be that certain data is available on a device, or that another command group has finished execution. An implementation may evaluate the requirements of a command group at any point after it has been submitted. The *processing of a command group* is the process by which a SYCL runtime evaluates all the requirements in a given  $R$ . The SYCL runtime will execute  $k$  only when all  $r_i$  are satisfied (i.e., when all requirements are satisfied). To simplify the notation, in the specification we refer to the set of requirements of a command group named *foo* as  $CG_{foo} = r_1, \dots, r_n$ .

The *evaluation of a requisite* ( $\text{Satisfied}(r_i)$ ) returns the status of the requisite, which can be *True* or *False*. A *satisfied* requisite implies the requirement is met.  $\text{Satisfied}(r_i)$  never alters the requisite, only observes the current status. The implementation may not block to check the requisite, and the same check can be performed multiple times.

An **action** ( $a_i$ ) is a collection of implementation-defined operations that must be performed in order to satisfy a requisite. The set of actions for a given **command group**  $A$  is permitted to be empty if no operation is required to satisfy the requirement. The notation  $a_i$  represents the action required to satisfy  $r_i$ . Actions of different requisites can be satisfied in any order with respect to each other without side effects (i.e., given two requirements  $r_j$  and  $r_k$ ,  $(r_j, r_k) \equiv (r_k, r_j)$ ). The intersection of two actions is not necessarily empty. **Actions** can include (but are not limited to): memory copy operations, memory mapping operations, coordination with the host, or implementation-specific behavior.

Finally, *Performing an action* ( $\text{Perform}(a_i)$ ) executes the action operations required to satisfy the requisite  $r_j$ . Note that, after  $\text{Perform}(a_i)$ , the evaluation  $\text{Satisfied}(r_j)$  will return *True* until the kernel is executed. After the kernel execution, it is not defined whether a different **command group** with the same requirements needs to perform the action again, where actions of different requisites inside the same **command group** object can be satisfied in any order with respect to each other without side effects: Given two requirements  $r_j$  and  $r_k$ ,  $\text{Perform}(a_j)$  followed by  $\text{Perform}(a_k)$  is equivalent to  $\text{Perform}(a_k)$  followed by  $\text{Perform}(a_j)$ .

The requirements of different **command groups** submitted to the same or different queues are evaluated in the relative order of submission. **command group** objects whose intersection of requirement sets is not empty are said to depend on each other. They are executed in order of submission to the queue. If **command groups** are submitted to different queues or by multiple threads, the order of execution is determined by the SYCL runtime. Note that independent **command group** objects can be submitted simultaneously without affecting dependencies.

Table 1 illustrates the execution order of three **command group** objects ( $CG_a, CG_b, CG_c$ ) with certain requirements submitted to the same queue. Both  $CG_a$  and  $CG_b$  only have one requirement,  $r_1$  and  $r_2$  respectively.  $CG_c$  requires both  $r_1$  and  $r_2$ . This enables the SYCL runtime to potentially execute  $CG_a$  and  $CG_b$  simultaneously, whereas  $CG_c$  cannot be executed until both  $CG_a$  and  $CG_b$  have been completed. The SYCL runtime evaluates the **requisites** and performs the **actions** required (if any) for the  $CG_a$  and  $CG_b$ . When evaluating the **requisites** of  $CG_c$ , they will be satisfied once the  $CG_a$  and  $CG_b$  have finished.

Table 1. Execution order of three command groups submitted to the same queue

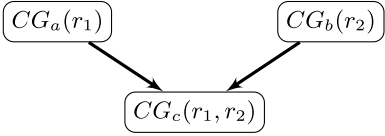
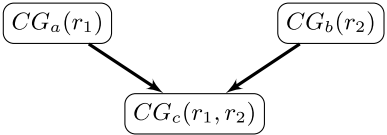
SYCL Application Enqueue Order	SYCL Kernel Execution Order
<pre>sycl::queue syclQueue; syclQueue.submit(CG_a(r_1)); syclQueue.submit(CG_b(r_2)); syclQueue.submit(CG_c(r_1, r_2));</pre>	 <pre> graph TD     CGa["CG_a(r_1)"] --&gt; CGc["CG_c(r_1, r_2)"]     CGb["CG_b(r_2)"] --&gt; CGc   </pre>

Table 2 uses three separate SYCL queue objects to submit the same **command group** objects as before.



Regardless of using three different queues, the execution order of the different **command group** objects is the same. When different threads enqueue to different queues, the execution order of the command group will be the order in which the submit member functions are executed. In this case, since the different **command group** objects execute on different devices, the **actions** required to satisfy the **requirements** may be different (e.g, the SYCL runtime may need to copy data to a different device in a separate context).

Table 2. Execution order of three command groups submitted to the different queues

SYCL Application Enqueue Order	SYCL Kernel Execution Order
<pre> sycl::queue syclQueue1; sycl::queue syclQueue2; sycl::queue syclQueue3; syclQueue1.submit(CG<sub>a</sub>(r<sub>1</sub>)); syclQueue2.submit(CG<sub>b</sub>(r<sub>2</sub>)); syclQueue3.submit(CG<sub>c</sub>(r<sub>1</sub>, r<sub>2</sub>)); </pre>	 <pre> graph TD     CGa["CG<sub>a</sub>(r<sub>1</sub>)"] --&gt; CGc["CG<sub>c</sub>(r<sub>1</sub>, r<sub>2</sub>)"]     CGb["CG<sub>b</sub>(r<sub>2</sub>)"] --&gt; CGc </pre>

### 3.7.1.3. Controlling execution order with events

Submitting an action for execution returns an **event** object. Programmers may use these events to explicitly coordinate host and device execution. Host code can wait for an event to complete, which will block execution on the host until the action(s) represented by the event have completed. The **event** class is described in greater detail in [Section 4.6.6](#).

Events may also be used to explicitly order the execution of kernels. Host code may wait for the completion of specific event, which blocks execution on the host until that event's action has completed. Events may also define requisites between **command groups**. Using events in this manner informs the runtime that one or more **command groups** must complete before another **command group** may begin executing. See [Section 4.9.4.1](#) for greater detail.

## 3.7.2. SYCL kernel execution model

When a kernel is submitted for execution, an index space is defined. An instance of the kernel body executes for each point in this index space. This kernel instance is called a **work-item** and is identified by its point in the index space, which provides a **global id** for the work-item. Each work-item executes the same code but the specific execution pathway through the code and the data operated upon can vary by using the work-item global id to specialize the computation.

An index space of size zero is allowed. All aspects of kernel execution proceed as normal with the exception that the kernel function itself is not executed. Note this means the command queue will still schedule this kernel after satisfying the requirements and this satisfies requirements of any dependent enqueued kernels.

### 3.7.2.1. Basic kernels

SYCL allows a simple execution model in which a kernel is invoked over an  $N$ -dimensional index space defined by **range<N>**, where  $N$  is one, two or three. Each work-item in such a kernel executes independently.

Each work-item is identified by a value of type **item<N>**. The type **item<N>** encapsulates a work-item identifier of type **id<N>** and a **range<N>** representing the number of work-items executing the kernel.

### 3.7.2.2. ND-range kernels

Work-items can be organized into **work-groups**, providing a more coarse-grained decomposition of the index space. Each work-group is assigned a unique **work-group id** with the same dimensionality as the

index space used for the work-items. Work-items are each assigned a [local id](#), unique within the work-group, so that a single work-item can be uniquely identified by its global id or by a combination of its local id and work-group id. The work-items in a given work-group execute on the processing elements of a single compute unit.

When work-groups are used in SYCL, the index space is called an [nd-range](#). An ND-range is an  $N$ -dimensional index space, where  $N$  is one, two or three. In SYCL, the ND-range is represented via the [nd\\_range<N>](#) class. An [nd\\_range<N>](#) is made up of a global range and a local range, each represented via values of type [range<N>](#). Additionally, there can be a global offset, represented via a value of type [id<N>](#); this is deprecated in SYCL 2020. The types [range<N>](#) and [id<N>](#) are each  $N$ -element arrays of integers. The iteration space defined via an [nd\\_range<N>](#) is an  $N$ -dimensional index space starting at the ND-range's global offset whose size is its global range, split into work-groups of the size of its local range.

Each work-item in the ND-range is identified by a value of type [nd\\_item<N>](#). The type [nd\\_item<N>](#) encapsulates a global id, local id and work-group id, all of type [id<N>](#) (the iteration space offset also of type [id<N>](#), but this is deprecated in SYCL 2020), as well as global and local ranges and coordination mechanisms necessary to make work-groups useful. Work-groups are assigned ids using a similar approach to that used for work-item global ids. Work-items are assigned to a work-group and given a local id with components in the range from zero to the size of the work-group in that dimension minus one. Hence, the combination of a work-group id and the local id within a work-group uniquely defines a work-item.

### 3.7.2.3. Backend-specific kernels

SYCL allows a [SYCL backend](#) to expose fixed functionality as non-programmable built-in kernels. The availability and behavior of these built-in kernels are [SYCL backend](#)-specific, and are not required to follow the SYCL execution and memory models. Furthermore the interface exposed utilize these built-in kernels is also [SYCL backend](#)-specific. See the relevant backend specification for details.

## 3.8. Memory model

Since SYCL is a single-source programming model, the memory model affects both the application and the device kernel parts of a program. On the SYCL application, the SYCL runtime will make sure data is available for execution of the kernels. On the SYCL device kernel, the [SYCL backend](#) rules describing how the memory behaves on a specific device are mapped to SYCL C++ constructs. Thus it is possible to program kernels efficiently in pure C++.

### 3.8.1. SYCL application memory model

The application running on the host uses SYCL [buffer](#) objects using instances of the [sycl::buffer](#) class or [USM](#) allocation functions to allocate memory in the global address space, or can allocate specialized image memory using the [sycl::unsampled\\_image](#) and [sycl::sampled\\_image](#) classes.

In the SYCL application, memory objects are bound to all devices in which they are used, regardless of the SYCL context where they reside. SYCL memory objects (namely, [buffer](#) and [image](#) objects) can encapsulate multiple underlying [SYCL backend](#) memory objects together with multiple host memory allocations to enable the same object to be shared between devices in different contexts, platforms or backends. [USM](#) allocations uniquely identify a memory allocation and are bound to a SYCL context. They are only valid on the backend used by the context.

The order of execution of [command group](#) objects ensures a sequentially consistent access to the memory from the different devices to the memory objects. Accessing a USM allocation does not alter the order of execution. Users must explicitly inform the SYCL runtime of any requirements necessary for a legal execution.

To access a memory object, the user must create an [accessor](#) object which parameterizes the type of access to the memory object that a kernel or the host requires. The [accessor](#) object defines a require-

ment to access a memory object, and this requirement is defined by construction of an accessor, regardless of whether there are any uses in a kernel or by the host. An accessor object specifies whether the access is via global memory, constant memory or image samplers and their associated access functions. The `accessor` also specifies whether the access is read-only (RO), write-only (WO) or read-write (RW). An optional `no_init` property can be added to an accessor to tell the system to discard any previous contents of the data the accessor refers to, so there are two additional requirement types: no-init-write-only (NWO) and no-init-read-write (NRW). For simplicity, when a **requisite** represents an accessor object in a certain access mode, we represent it as `MemoryObjectAccessMode`. For example, an accessor that accesses memory object `buf1` in **RW** mode is represented as `buf1RW`. A **command group** object that uses such an accessor is represented as `CG(buf1RW)`. The **action** required to satisfy a requisite and the location of the latest copy of a memory object will vary depending on the implementation.

Table 3 illustrates an example where **command group** objects are enqueued to two separate SYCL queues executing in devices in different contexts. The **requisites** for the **command group** execution are the same, but the **actions** to satisfy them are different. For example, if the data is on the host before execution, `A(b1RW)` and `A(b2RW)` can potentially be implemented as copy operations from the host memory to `context1` or `context2` respectively. After `CGa` and `CGb` are executed, `A'(b1RW)` will likely be an empty operation, since the result of the kernel can stay on the device. On the other hand, the results of `CGb` are now on a different context than `CGc` is executing, therefore `A'(b2RW)` will need to copy data across two separate contexts using an implementation specific mechanism.

Table 3. Actions performed when three command groups are submitted to two distinct queues

SYCL Application Enqueue Order	SYCL Kernel Execution Order
<pre> sycl::queue q1(context1); sycl::queue q2(context2); q1.submit(CG<sub>a</sub>(b1<sub>RW</sub>)); q2.submit(CG<sub>b</sub>(b2<sub>RW</sub>)); q1.submit(CG<sub>c</sub>(b1<sub>RW</sub>, b2<sub>RW</sub>)); </pre>	
Possible implementation by a SYCL Runtime	

Table 3 shows actions performed when three command groups are submitted to two distinct queues, and potential implementation in an OpenCL **SYCL backend** by a SYCL runtime. Note that in this example, each SYCL buffer (`b2`, `b2`) is implemented as separate `cl_mem` objects per context.

Note that the order of the definition of the accessors within the **command group** is irrelevant to the requirements they define. All accessors always apply to the entire **command group** object where they are defined.

When multiple **accessors** in the same **command group** define different requisites to the same memory object these requisites must be resolved.

Firstly, any requisites with different access modes but the same access target are resolved into a single requisite with the union of the different access modes according to Table 4. The atomic access mode acts

as if it was read-write (RW) when determining the combined requirement. The rules in Table 4 are commutative and associative.

Table 4. Combined requirement from two different accessor access modes within the same *command group*. The rules are commutative and associative

One access mode	Other access mode	Combined requirement
read (RO)	write (WO)	read-write (RW)
read (RO)	read-write (RW)	read-write (RW)
write (WO)	read-write (RW)	read-write (RW)
no-init-write (NWO)	no-init-read-write (NRW)	no-init-read-write (NRW)
no-init-write (NWO)	write (WO)	write (WO)
no-init-write (NWO)	read (RO)	read-write (RW)
no-init-write (NWO)	read-write (RW)	read-write (RW)
no-init-read-write (NRW)	write (WO)	read-write (RW)
no-init-read-write (NRW)	read (RO)	read-write (RW)
no-init-read-write (NRW)	read-write (RW)	read-write (RW)

The result of this should be that there should not be any requisites with the same access target.

Secondly, the remaining requisites must adhere to the following rule. Only one of the requisites may have write access (*W* or *RW*), otherwise the SYCL runtime must throw an exception. All requisites create a requirement for the data they represent to be made available in the specified access target, however only the requisite with write access determines the side effects of the *command group*, i.e. only the data which that requisite represents will be updated.

For example:

- $CG(b1_{RW}^G, b1_R^H)$  is permitted.
- $CG(b1_{RW}^G, b1_{RW}^H)$  is **not** permitted.
- $CG(b1_W^G, b1_{RW}^C)$  is **not** permitted.

Where *G* and *C* correspond to a `target::device` and `target::constant_buffer` accessor and *H* corresponds to a host accessor.

A buffer created from a range of an existing buffer is called a sub-buffer. A buffer may be overlaid with any number of sub-buffers. Accessors can be created to operate on these sub-buffers. Refer to Section 4.7.2 for details on sub-buffer creation and restrictions. A requirement to access a sub-buffer is represented by specifying its range, e.g.  $CG(b1_{RW,[0,5)})$  represents the requirement of accessing the range  $[0,5)$  buffer *b1* in read write mode.

If two accessors are constructed to access the same buffer, but both are to non-overlapping sub-buffers of the buffer, then the two accessors are said to not overlap, otherwise the accessors do overlap. Overlapping is the test that is used to determine the scheduling order of command groups. Command-groups with non-overlapping requirements may execute concurrently.

Table 5. Requirements on overlapping vs non-overlapping sub-buffer

SYCL Application Enqueue Order	SYCL Kernel Execution Order
<pre> sycl::queue q1(context1); q1.submit(CG<sub>a</sub>(b1<sub>RW</sub>, [0, 10])); q1.submit(CG<sub>b</sub>(b1<sub>RW</sub>, [10, 20])); q1.submit(CG<sub>c</sub>(b1<sub>RW</sub>, [5, 15])); </pre>	<pre> graph TD     CGa["CG<sub>a</sub>(b1<sub>RW</sub>, [0, 10])"]     CGb["CG<sub>b</sub>(b1<sub>RW</sub>, [10, 20])"]     CGc["CG<sub>c</sub>(b1<sub>RW</sub>, [5, 15])"]     CGa --&gt; CGc     CGb --&gt; CGc </pre>

It is permissible for command groups that only read data to not copy that data back to the host or other devices after reading and for the runtime to maintain multiple read-only copies of the data on multiple devices.

A special case of requirement is the one defined by a **host accessor**. Host accessors are represented with  $H(\text{MemoryObject}_{\text{AccessMode}})$ , e.g.,  $H(b1_{RW})$  represents a host accessor to  $b1$  in read-write mode. Host accessors are a special type of accessor constructed from a memory object outside a command group, and require that the data associated with the given memory object is available on the host in the given pointer. This causes the runtime to block on construction of this object until the requirement has been satisfied. **Host accessor** objects are effectively barriers on all accesses to a certain memory object. Table 6 shows an example of multiple command groups enqueued to the same queue. Once the host accessor  $H(b1_{RW})$  is reached, the execution cannot proceed until  $CG_a$  is finished. However,  $CG_b$  does not have any requirements on  $b1$ , therefore, it can execute concurrently with the barrier. Finally,  $CG_c$  will be enqueued after  $H(b1_{RW})$  is finished, but still has to wait for  $CG_b$  to conclude for all its requirements to be satisfied. See Section 3.9.8 for details on host-device coordination.

Table 6. Execution of command groups when using host accessors

SYCL Application Enqueue Order	SYCL Kernel Execution Order
<pre> sycl::queue q1; q1.submit(CG<sub>a</sub>(b1<sub>RW</sub>)); q1.submit(CG<sub>b</sub>(b2<sub>RW</sub>));  H(b1<sub>RW</sub>);  q1.submit(CG<sub>c</sub>(b1<sub>RW</sub>, b2<sub>RW</sub>)); </pre>	<pre> graph TD     CGa["CG<sub>a</sub>(b1<sub>RW</sub>)"]     CGb["CG<sub>b</sub>(b2<sub>RW</sub>)"]     H["H(b1<sub>RW</sub>)"]     CGc["CG<sub>c</sub>(b1<sub>RW</sub>, b2<sub>RW</sub>)"]     CGa --&gt; H     CGb --&gt; CGc     H --&gt; CGc </pre>

### 3.8.2. SYCL device memory model

The memory model for SYCL devices is based on the OpenCL memory model. Work-items executing in a kernel have access to three distinct address spaces (memory regions) and a virtual address space overlapping some concrete address spaces:

- **Global-memory** is accessible to all work-items in all work-groups. Work-items can read from or write to any element of a global memory object. Reads and writes to global memory may be cached depending on the capabilities of the device. Global memory is persistent across kernel invocations. Concurrent access to a location in an USM allocation by two or more executing kernels where at least one kernel modifies that location is a data race; there is no guarantee of correct results unless **memory fence** and atomic operations are used.
- **Local-memory** is accessible to all work-items in a single work-group. Attempting to access local memory in one work-group from another work-group results in undefined behavior. This memory region can be used to allocate variables that are shared by all work-items in a work-group. Work-group-level visibility allows local memory to be implemented as dedicated regions of the device memory where this is appropriate.
- **Private-memory** is a region of memory private to a work-item. Attempting to access private memory



in one work-item from another work-item results in undefined behavior.

- **Generic-memory** is a virtual address space which overlaps the global, local and private address spaces. Therefore, an object that resides in the global, local, or private address space can also be accessed through the generic address space.

### 3.8.2.1. Access to memory

Accessors in the device kernels provide access to the memory objects, acting as pointers to the corresponding address space.

Pointers can be passed directly as kernel arguments if an implementation supports **USM**. See [Section 4.8](#) for information on when it is legal to dereference pointers passed from the host inside kernels.

To allocate local memory within a kernel, the user can either pass a `sycl::local_accessor` object as a argument to an ND-range kernel (that has a user-defined work-group size), or can define a variable in work-group scope inside `sycl::parallel_for_work_group`.

Any variable defined inside a `sycl::parallel_for` scope or `sycl::parallel_for_work_item` scope will be allocated in private memory. Any variable defined inside a `sycl::parallel_for_work_group` scope will be allocated in local memory.

Users can create accessors that reference sub-buffers as well as entire buffers.

Within kernels, the underlying C++ pointer types can be obtained from an accessor. The pointer types will contain a compile-time deduced address space. So, for example, if a C++ pointer is obtained from an accessor to global memory, the C++ pointer type will have a global address space attribute attached to it. The address space attribute will be compile-time propagated to other pointer values when one pointer is initialized to another pointer value using a defined algorithm.

When developers need to explicitly state the address space of a pointer value, one of the explicit pointer classes can be used. There is a different explicit pointer class for each address space: `sycl::raw_local_ptr`, `sycl::raw_global_ptr`, `sycl::raw_private_ptr`, `sycl::raw_generic_ptr`, `sycl::decorated_local_ptr`, `sycl::decorated_global_ptr`, `sycl::decorated_private_ptr`, or `sycl::decorated_generic_ptr`.

The classes with the **decorated** prefix expose pointers that use an implementation-defined address space decoration, while the classes with the **raw** prefix do not. Buffer accessors with an access target `target::device` or `target::constant_buffer` and local accessors can be converted into explicit pointer classes (`multi_ptr`).

For templates that need to adapt to different address spaces, a `sycl::multi_ptr` class is defined which is templated via a compile-time constant enumerator value to specify the address space.

## 3.8.3. SYCL memory consistency model

The SYCL memory consistency model is based upon the memory consistency model of the C++ core language. Where SYCL offers extensions to classes and functions that may affect memory consistency, the default behavior when these extensions are not used always matches the behavior of standard C++.

A SYCL implementation must guarantee that the same memory consistency model is used across host and device code. Every **device compiler** must support the memory model defined by the minimum version of C++ described in [Section 3.9.1](#); SYCL implementations supporting additional versions of C++ must also support the corresponding memory models.

Within a work-item, operations are ordered according to the *sequenced before* relation defined by the C++ core language.

Ensuring memory consistency across different work-items requires careful usage of **group barrier** operations, **mem-fence** operations and atomic operations. The ordering of operations across different work-

items is determined by the *happens before* relation defined by the C++ core language, with a single relation governing all address spaces (memory regions).

On any SYCL device, local and global memory may be made consistent across work-items in a single [group](#) through use of a [group barrier](#) operation. On SYCL devices supporting acquire-release or sequentially consistent memory orderings, all memory visible to a set of work-items may be made consistent across the work-items in that set through the use of [mem-fence](#) and atomic operations.

Memory consistency between the host and SYCL device(s), or different SYCL devices in the same context, can be guaranteed through library calls in the host application, as defined in [Section 3.9.8](#). On SYCL devices supporting concurrent atomic accesses to USM allocations and acquire-release or sequentially consistent memory orderings, cross-device memory consistency can be enforced through the use of [mem-fence](#) and atomic operations.

### 3.8.3.1. Memory ordering

```

1 namespace sycl {
2
3 enum class memory_order : /* unspecified */ {
4     relaxed,
5     acquire,
6     release,
7     acq_rel,
8     seq_cst
9 };
10
11 inline constexpr auto memory_order_relaxed = memory_order::relaxed;
12 inline constexpr auto memory_order_acquire = memory_order::acquire;
13 inline constexpr auto memory_order_release = memory_order::release;
14 inline constexpr auto memory_order_acq_rel = memory_order::acq_rel;
15 inline constexpr auto memory_order_seq_cst = memory_order::seq_cst;
16
17 } // namespace sycl

```

The memory synchronization order of a given atomic operation is controlled by a `sycl::memory_order` parameter, which can take one of the following values:

- `sycl::memory_order::relaxed`;
- `sycl::memory_order::acquire`;
- `sycl::memory_order::release`;
- `sycl::memory_order::acq_rel`;
- `sycl::memory_order::seq_cst`.

The meanings of these values are identical to those defined in the C++ core language.

These memory orders are listed above from weakest (`memory_order::relaxed`) to strongest (`memory_order::seq_cst`).

The complete set of memory orders is not guaranteed to be supported by every device, nor across all combinations of devices within a platform. The set of supported memory orders can be queried via the information descriptors for the `sycl::device` and `sycl::context` classes.



SYCL implementations are not required to support a memory order equivalent to `std::memory_order::consume`, and using this ordering within a SYCL device kernel results in undefined behavior. Developers are encouraged to use `sycl::memory_order::acquire`

instead.

### 3.8.3.2. Memory scope

```

1 namespace sycl {
2
3 enum class memory_scope : /* unspecified */ {
4     work_item,
5     sub_group,
6     work_group,
7     device,
8     system
9 };
10
11 inline constexpr auto memory_scope_work_item = memory_scope::work_item;
12 inline constexpr auto memory_scope_sub_group = memory_scope::sub_group;
13 inline constexpr auto memory_scope_work_group = memory_scope::work_group;
14 inline constexpr auto memory_scope_device = memory_scope::device;
15 inline constexpr auto memory_scope_system = memory_scope::system;
16
17 } // namespace sycl

```

The set of [work-items](#) and devices to which the memory ordering constraints of a given atomic operation apply is controlled by a `sycl::memory_scope` parameter, which can take one of the following values:

- `sycl::memory_scope::work_item` The ordering constraint applies only to the calling work-item;
- `sycl::memory_scope::sub_group` The ordering constraint applies only to work-items in the same [sub-group](#) as the calling work-item;
- `sycl::memory_scope::work_group` The ordering constraint applies only to work-items in the same [work-group](#) as the calling work-item;
- `sycl::memory_scope::device` The ordering constraint applies only to work-items executing on the same device as the calling work-item;
- `sycl::memory_scope::system` The ordering constraint applies to any work-item or host thread in the system that is currently permitted to access the memory allocation containing the referenced object, as defined by the capabilities of [buffers](#) and [USM](#).

The memory scopes are listed above from narrowest (`memory_scope::work_item`) to widest (`memory_scope::system`).

The complete set of memory scopes is not guaranteed to be supported by every device. The set of supported memory scopes can be queried via the information descriptors for the `sycl::device` and `sycl::context` classes.

The widest scope that can be applied to an atomic operation corresponds to the set of work-items which can access the associated memory location. For example, the widest scope that can be applied to atomic operations in work-group local memory is `sycl::memory_scope::work_group`. If a wider scope is supplied, the behavior is as-if the narrowest scope containing all work-items which can access the associated memory location was supplied.



The addition of memory scopes to the C++ memory model modifies the definition of some concepts from the C++ core language. For example: data races, the synchronizes-with relationship and sequential consistency must be defined in a way that accounts for atomic operations with differing (but compatible) scopes, in a manner similar to the [OpenCL 2.0 specification](#). Efforts to formalize the memory model of SYCL are ongoing,



and a formal memory model will be included in a future version of the SYCL specification.

### 3.8.3.3. Atomic operations

Atomic operations can be performed on memory in buffers and USM. The `sycl::atomic_ref` class must be used to provide safe atomic access to the buffer or USM allocation from device code.

### 3.8.3.4. Forward progress

This section, and any subsequent section referring to progress guarantees, uses the following terms as defined in the C++ core language: thread of execution; weakly parallel forward progress guarantees; parallel forward progress guarantees; concurrent forward progress guarantees; and block with forward progress guarantee delegation.

Each work-item in SYCL is a separate thread of execution, providing at least weakly parallel forward progress guarantees. Whether work-items provide stronger forward progress guarantees is implementation-defined.

All implementations must additionally ensure that a work-item arriving at a [group barrier](#) does not prevent other work-items in the same group from making progress. When a work-item arrives at a group barrier acting on group *G*, implementations must eventually select and potentially strengthen another work-item in group *G* that has not yet arrived at the barrier.

When a host thread blocks on the completion of a command previously submitted to a SYCL queue (for example, via the `sycl::queue::wait` function), it blocks with forward progress guarantee delegation.



SYCL commands submitted to a queue are not guaranteed to begin executing until a host thread blocks on their completion. In the absence of multiple host threads, there is no guarantee that host and device code will execute concurrently.

## 3.9. The SYCL programming model

A SYCL program is written in standard C++. Host code and device code is written in the same C++ source file, enabling instantiation of templated kernels from host code and also enabling kernel source code to be shared between host and device. The device kernels are encapsulated C++ callable types (a function object with `operator()` or a lambda expression), which have been designated to be compiled as SYCL kernels.

SYCL programs target heterogeneous systems. The kernels may be compiled and optimized for multiple different processor architectures with very different binary representations.

### 3.9.1. Minimum version of C++

The C++ features used in SYCL are based on a specific version of C++. Implementations of SYCL must support this minimum C++ version, which defines the C++ constructs that can consequently be used by SYCL feature definitions (for example, lambdas).

The minimum C++ version of this SYCL specification is determined by the normative C++ core language defined in [Section 3.3](#). All implementations of this specification must support at least this core language, and features within this specification are defined using features of the core language. Note that not all core language constructs are supported within [SYCL kernel functions](#) or code invoked by a [SYCL kernel function](#), as detailed by [Section 5.4](#).

Implementations may support newer C++ versions than the minimum required by SYCL. Code written using newer features than the SYCL requirement, though, may not be portable to other implementations that don't support the same C++ version.

### 3.9.2. Alignment with future versions of C++

Some features of SYCL are aligned with the next C++ specification, as defined in [Section 3.3](#).

The following features are pre-adopted by SYCL 2020 and made available in the `sycl::` namespace: `std::span`, `std::dynamic_extent`, `std::bit_cast`. The implementations of pre-adopted features are compliant with the next C++ specification, and are expected to forward directly to standard C++ features in a future version of SYCL.

The following features of SYCL 2020 use syntax based on the next C++ specification: `sycl::atomic_ref`. These features behave as described in the next C++ specification, barring modifications to ensure compatibility with other SYCL 2020 features and heterogeneous programming. Any such modifications are documented in the corresponding sections of this specification.

### 3.9.3. Basic data parallel kernels

Data-parallel [kernels](#) that execute as multiple [work-items](#) and where no work-group-local coordination is required are enqueued with the `sycl::parallel_for` function parameterized by a `sycl::range` parameter. These kernels will execute the kernel function body once for each work-item in the specified [range](#).

Functionality tied to [groups](#) of work-items, including [group barriers](#) and [local memory](#), must not be used within these kernels.

Variables with [reduction](#) semantics can be added to basic data parallel kernels using the features described in [Section 4.9.2](#).

### 3.9.4. Work-group data parallel kernels

Data parallel [kernels](#) can also execute in a mode where the set of [work-items](#) is divided into [work-groups](#) of user-defined dimensions. The user specifies the global [range](#) and local work-group size as parameters to the `sycl::parallel_for` function with a `sycl::nd_range` parameter. In this mode of execution, kernels execute over the [nd-range](#) in work-groups of the specified size. It is possible to share data among work-items within the same work-group in [local](#) or [global memory](#), and the `group_barrier` function can be used to block a work-item until all work-items in the same work-group arrive at the barrier. All work-groups in a given `parallel_for` will be the same size, and the global size defined in the nd-range must either be a multiple of the work-group size in each dimension, or the global size must be zero. When the global size is zero, the kernel function is not executed, the local size is ignored, and any dependencies are satisfied.

Work-groups may be further subdivided into [sub-groups](#). The work-items that compose a sub-group are selected in an implementation-defined way, and therefore the size and number of sub-groups may differ for each kernel. Moreover, different devices may make different guarantees with respect to how sub-groups within a work-group are scheduled. The maximum number of work-items in any sub-group in a kernel is based on a combination of the kernel and its dispatch dimensions. The size of any sub-group in the dispatch is between 1 and this maximum sub-group size, and the size of an individual sub-group is invariant for the duration of a kernel's execution. Similarly to work-groups, the `group_barrier` function can be used to block a work-item until all work-items in the same sub-group arrive at the barrier.

Portable device code must not assume that work-items within a sub-group execute in any particular order, that work-groups are subdivided into sub-groups in a specific way, nor that the work-items within a sub-group provide specific forward progress guarantees.

Variables with [reduction](#) semantics can be added to work-group data parallel kernels using the features described in [Section 4.9.2](#).

### 3.9.5. Hierarchical data parallel kernels (deprecated)

Hierarchical data parallel kernels and all classes that are only available within such kernels are deprecated in SYCL 2020, and will be removed in a future version of SYCL.

The SYCL compiler provides a way of specifying data parallel kernels that execute within work-groups via a different syntax which highlights the hierarchical nature of the parallelism. This mode is purely a compiler feature and does not change the execution model of the kernel. Instead of calling `sycl::parallel_for` the user calls `sycl::parallel_for_work_group` with a `sycl::range` value representing the number of work-groups to launch and optionally a second `sycl::range` representing the size of each work-group for performance tuning. All code within the `parallel_for_work_group` scope effectively executes once per work-group. Within the `parallel_for_work_group` scope, it is possible to call `parallel_for_work_item` which creates a new scope in which all work-items within the current work-group execute. This enables a programmer to write code that looks like there is an inner work-item loop inside an outer work-group loop, which closely matches the effect of the execution model. All variables declared inside the `parallel_for_work_group` scope are allocated in work-group local memory, whereas all variables declared inside the `parallel_for_work_item` scope are declared in private memory. All `parallel_for_work_item` calls within a given `parallel_for_work_group` execution must have the same dimensions.

### 3.9.6. Kernels that are not launched over parallel instances

Simple kernels for which only a single instance of the kernel function will be executed are enqueued with the `sycl::single_task` function. The kernel enqueued takes no “work-item id” parameter and will only execute once. The behavior is logically equivalent to executing a kernel on a single compute unit with a single work-group comprising only one work-item. Such kernels may be enqueued on multiple queues and devices and as a result may be executed in task-parallel fashion.

### 3.9.7. Pre-defined kernels

Some [SYCL backends](#) may expose pre-defined functionality to users as kernels. These kernels are not programmable, hence they are not bound by the SYCL C++ programming model restrictions, and how they are written is implementation-defined.

### 3.9.8. Coordination and synchronization

Coordination between the host and any devices can be expressed in the host SYCL application using calls into the SYCL runtime. Coordination between work-items executing inside of device code can be expressed using group barriers.

Some function calls synchronize with other function calls performed by another thread (potentially on another device). Other functions are defined in terms of their synchronization operations. Such functions can be used to ensure that the host and any devices do not access data concurrently, and/or to reason about the ordering of operations across the host and any devices.

#### 3.9.8.1. Host-device coordination

The following operations can be used to coordinate host and device(s):

- *Buffer destruction*: The destructors for `sycl::buffer`, `sycl::unsampled_image` and `sycl::sampled_image` objects block until all submitted work on those objects completes and copy the data back to host memory before returning. These destructors only block if the object was constructed with attached host memory and if data needs to be copied back to the host.

More complex forms of buffer destruction can be specified by the user by constructing buffers with other kinds of references to memory, such as `shared_ptr` and `unique_ptr`.

- *Host Accessors*: The constructor for a host accessor blocks until all kernels that modify the same

buffer (or image) in any queues complete and then copies data back to host memory before the constructor returns. Any command groups with requirements to the same memory object cannot execute until the host accessor is destroyed as shown on [Table 6](#).

- *Command group enqueue*: The [SYCL runtime](#) internally ensures that any command groups added to queues have the correct event dependencies added to those queues to ensure correct operation. Adding command groups to queues never blocks, and the `sycl::event` returned by the queue's submit function contains event information related to the specific command group.
- *Queue operations*: The user can manually use queue operations, such as `sycl::queue::wait()` to block execution of the calling thread until all the command groups submitted to the queue have finished execution. Note that this will also affect the dependencies of those command groups in other queues.
- *SYCL event objects*: SYCL provides `sycl::event` objects which can be used to track and specify dependencies. The [SYCL runtime](#) must ensure that these objects can be used to enforce dependencies that span SYCL contexts from different [SYCL backends](#).

The specification for each of these blocking functions defines some set of operations that cause the function to unblock. These operations always happen before the blocking function returns (using the definition of "happens before" from the C++ specification).

Note that the destructors of other SYCL objects (`sycl::queue`, `sycl::context`,...) do not block. Only a `sycl::buffer`, `sycl::sampled_image` or `sycl::unsampled_image` destructor might block. The rationale is that an object without any side effect on the host does not need to block on destruction as it would impact the performance. So it is up to the programmer to use a member function to wait for completion in some cases if this does not fit the goal. See [Section 3.9.12](#) for more information on object life time.

### 3.9.8.2. Work-item coordination

A [group barrier](#) provides a mechanism to coordinate all work-items in the same group. All work-items in a group must execute the barrier before any are allowed to continue execution beyond the barrier. Note that the group barrier must be encountered by all work-items of a group executing the kernel or by none at all. [work-group barrier](#) and [sub-group barrier](#) functionality is exposed via the `group_barrier` function.

Coordination between work-items in different work-groups must take place via atomic operations, and is possible only on SYCL device with certain capabilities, as described in [Section 3.8.3](#).

## 3.9.9. Error handling

In SYCL, there are two types of errors: synchronous errors that can be detected immediately when an API call is made, and [asynchronous errors](#) that can only be detected later after an API call has returned. Synchronous errors, such as failure to construct an object, are reported immediately by the runtime throwing an exception. [Asynchronous errors](#), such as an error occurring during execution of a kernel on a device, are reported via an asynchronous error-handler mechanism.

[Asynchronous errors](#) are not reported immediately as they occur. The asynchronous error handler for a context or queue is called with a `sycl::exception_list` object, which contains a list of asynchronously-generated exception objects, on the conditions described by [Section 4.13.1.1](#) and [Section 4.13.1.2](#).

Asynchronous errors may be generated regardless of whether the user has specified any asynchronous error handler(s), as described in [Section 4.13.1.2](#).

Some [SYCL backends](#) can report errors that are specific to the platform they are targeting, or that are more concrete than the errors provided by the SYCL API. Any error reported by a [SYCL backend](#) must derive from the base `sycl::exception`. When a user wishes to capture specifically an error thrown by a [SYCL backend](#), she must include the [SYCL backend](#)-specific headers for said [SYCL backend](#).

### 3.9.10. Fallback mechanism

A [command group function object](#) can be submitted either to a single queue to be executed on, or to a secondary queue. If a [command group function object](#) fails to be enqueued to the primary queue, then the system will attempt to enqueue it to the secondary queue, if given as a parameter to the submit function. If the [command group function object](#) fails to be queued to both of these queues, then a synchronous SYCL exception will be thrown.

It is possible that a command group may be successfully enqueued, but then asynchronously fail to run, for some reason. In this case, it may be possible for the runtime system to execute the [command group function object](#) on the secondary queue, instead of the primary queue. The situations where a SYCL runtime may be able to achieve this asynchronous fall-back is implementation-defined.

### 3.9.11. Scheduling of kernels and data movement

A [command group function object](#) takes a reference to a command group [handler](#) as a parameter and anything within that scope is immediately executed and takes the handler object as a parameter. The intention is that a user will perform calls to SYCL functions, member functions, destructors and constructors inside that scope. These calls will be non-blocking on the host, but enqueue operations to the queue that the command group is submitted to. All user functions within the command group scope will be called on the host as the [command group function object](#) is executed, but any [commands](#) it invokes will be added to the SYCL [queue](#). All commands added to the [queue](#) will be executed out-of-order from each other, according to their data dependencies.

### 3.9.12. Managing object lifetimes

A SYCL application does not initialize any [SYCL backend](#) features until a [sycl::context](#) object is created. A user does not need to explicitly create a [sycl::context](#) object, but they do need to explicitly create a [sycl::queue](#) object, for which a [sycl::context](#) object will be implicitly created if not provided by the user.

All [SYCL backend](#) objects encapsulated in SYCL objects are reference-counted and will be destroyed once all references have been released. This means that a user needs only create a SYCL [queue](#) (which will automatically create an SYCL context) for the lifetime of their application to initialize and release any [SYCL backend](#) objects safely.

There is no global state specified to be required in SYCL implementations. This means, for example, that if the user creates two queues without explicitly constructing a common context, then a SYCL implementation does not have to create a shared context for the two queues. Implementations are free to share or cache state globally for performance, but it is not required.

Memory objects can be constructed with or without attached host memory. If no host memory is attached at the point of construction, then destruction of that memory object is non-blocking. The user may use C++ standard pointer classes for sharing the host data with the user application and for defining blocking, or non-blocking behavior of the buffers and images. If host memory is attached by using a raw pointer, then the default behavior is followed, which is that the destructor will block until any command groups operating on the memory object have completed, then, if the contents of the memory object is modified on a device those contents are copied back to host and only then does the destructor return.

In the case where host memory is shared between the user application and the [SYCL runtime](#) with a [std::shared\\_ptr](#), then the reference counter of the [std::shared\\_ptr](#) determines whether the buffer needs to copy data back on destruction, and in that case the blocking or non-blocking behavior depends on the user application.

Instead of a [std::shared\\_ptr](#), a [std::unique\\_ptr](#) may be provided, which uses move semantics for initializing and using the associated host memory. In this case, the behavior of the buffer in relation to the user application will be non-blocking on destruction.



As said in [Section 3.9.8](#), the only blocking operations in SYCL (apart from explicit wait operations) are:

- host accessor constructor, which waits for any kernels enqueued before its creation that write to the corresponding object to finish and be copied back to host memory before it starts processing. The host accessor does not necessarily copy back to the same host memory as initially given by the user;
- memory object destruction, in the case where copies back to host memory have to be done or when the host memory is used as a backing-store.

### 3.9.13. Device discovery and selection

A user specifies which queue to submit a [command group function object](#) and each [queue](#) is targeted to run on a specific [device](#) (and [context](#)). A user can specify the actual device on queue creation, or they can specify a [device selector](#) which causes the [SYCL runtime](#) to choose a device based on the user's provided preferences. Specifying a [device selector](#) causes the [SYCL runtime](#) to perform device discovery. No device discovery is performed until a SYCL [device selector](#) is passed to a queue constructor. Device topology may be cached by the [SYCL runtime](#), but this is not required.

Device discovery will return all [devices](#) from all [platforms](#) exposed by all the supported [SYCL backends](#).

### 3.9.14. Interfacing with the SYCL backend API

There are two styles of developing a SYCL application:

1. writing a pure SYCL generic application;
2. writing a SYCL application that relies on some [SYCL backend](#) specific behavior.

When users follow 1., there is no assumption about what [SYCL backend](#) will be used during compilation or execution of the SYCL application. Therefore, the [SYCL backend API](#) is not assumed to be available to the developer. Only standard C++ types and interfaces are assumed to be available, as described in [Section 3.9](#). Users only need to include the `<sycl/sycl.hpp>` header to write a SYCL generic application.

On the other hand, when users follow 2., they must know what [SYCL backend APIs](#) they are using. In this case, any header required for the normal programmability of the [SYCL backend API](#) is assumed to be available to the user. In addition to the `<sycl/sycl.hpp>` header, users must also include the [SYCL backend-specific](#) header as defined in [Section 4.3](#). The [SYCL backend-specific](#) header provides the interoperability interface for the SYCL API to interact with [native backend objects](#).

The interoperability API is defined in [Section 4.5.1](#).

## 3.10. Memory objects

SYCL memory objects represent data that is handled by the [SYCL runtime](#) and can represent allocations in one or multiple [devices](#) at any time. Memory objects, both buffers and images, may have one or more underlying [native backend objects](#) to ensure that [queues](#) objects can use data in any device. A SYCL implementation may have multiple [native backend objects](#) for the same device. The [SYCL runtime](#) is responsible for ensuring the different copies are up-to-date whenever necessary, using whatever mechanism is available in the system to update the copies of the underlying [native backend objects](#).

#### *Implementation note*



A valid mechanism for this update is to transfer the data from one [SYCL backend](#) into the system memory using the [SYCL backend-specific](#) mechanism available, and then transfer it to a different device using the mechanism exposed by the new [SYCL backend](#).

Memory objects in SYCL fall into one of two categories: [buffer](#) objects and [image](#) objects. A buffer object stores a one-, two- or three-dimensional collection of elements that are stored linearly directly back to back in the same way C or C++ stores arrays. An image object is used to store a one-, two- or three-dimen-

sional texture, frame-buffer or image data that may be stored in an optimized and device-specific format in memory and must be accessed through specialized operations.

Elements of a buffer object can be a scalar data type (such as an `int` or `float`), vector data type, or a user-defined structure. In SYCL, a `buffer` object is a templated type (`sycl::buffer`), parameterized by the element type and number of dimensions. An `image` object is stored in one of a limited number of formats. The elements of an image object are selected from a list of predefined image formats which are provided by an underlying SYCL backend implementation. Images are encapsulated in the `sycl::unsampled_image` or `sycl::sampled_image` types, which are templated by the number of dimensions in the image. The minimum number of elements in an image object is one. The minimum number of elements in a buffer object is zero.

The fundamental differences between a buffer and an image object are:

- elements in a buffer are stored in an array of 1, 2 or 3 dimensions and can be accessed using an accessor by a kernel executing on a device. The accessors for kernels provide a member function to get C++ pointer types, or the `sycl::global_ptr` class;
- elements of an image are stored in a format that is opaque to the user and cannot be directly accessed using a pointer. SYCL provides image accessors and samplers to allow a kernel to read from or write to an image;
- for a buffer object the data is accessed within a kernel in the same format as it is stored in memory, but in the case of an image object the data is not necessarily accessed within a kernel in the same format as it is stored in memory;
- image elements are always a 4-component vector (each component can be a float or signed/unsigned integer) in a kernel. Accessors that read an image convert image elements from their storage format into a 4-component vector.

Similarly, the SYCL accessor member functions provided to write to an image convert the image element from a 4-component vector to the appropriate image format specified such as four 8-bit elements, for example.

Users may want fine-grained control of the memory management and storage semantics of SYCL image or buffer objects. For example, a user may wish to specify the host memory for a memory object to use, but may not want the memory object to block on destruction.

Depending on the control and the use cases of the SYCL applications, well established C++ classes and patterns can be used for reference counting and sharing data between user applications and the SYCL runtime. For control over memory allocation on the host and mapping between host and device memory, pre-defined or user-defined C++ `std::allocator` classes are used. To avoid data races when sharing data between SYCL and non-SYCL applications, `std::shared_ptr` and `std::mutex` classes are used.

## 3.11. Multi-dimensional objects and linearization

SYCL defines a number of multi-dimensional objects such as buffers and accessors. The iteration space of work-items in a kernel may also be multi-dimensional. The size of each dimension is defined by a `range` object of one, two or three dimensions, and an element in the multi-dimensional space can be identified using an `id` object with the same number of dimensions as the corresponding `range`.

If the size of any dimension is zero, there are zero elements in the multi-dimensional range.

### 3.11.1. Linearization

Some multi-dimensional objects can be viewed in a linear form. When this happens, the right-most term in the object's range varies fastest in the linearization.

A three-dimensional element `id{id0, id1, id2}` within a three-dimensional object of range `range{r0,`

`r1`, `r2` has a linear position defined by:

$$id2 + (id1 \cdot r2) + (id0 \cdot r1 \cdot r2)$$

A two-dimensional element `id{id0, id1}` within a two-dimensional `range{r0, r1}` follows a similar equation:

$$id1 + (id0 \cdot r1)$$

A one-dimensional element `id{id0}` within a one-dimensional range `range{r0}` is equivalent to its linear form.

### 3.11.2. Multi-dimensional subscript operators

Some multi-dimensional objects can be indexed using the subscript operator where consecutive subscript operators correspond to each dimension. The right-most operator varies fastest, as with standard C++ arrays. Formally, a three-dimensional subscript access `a[id0][id1][id2]` references the element at `id{id0, id1, id2}`. A two-dimensional subscript access `a[id0][id1]` references the element at `id{id0, id1}`. A one-dimensional subscript access `a[id0]` references the element at `id{id0}`.

## 3.12. Implementation options

The SYCL language is designed to allow several different possible implementations. The contents of this section are non-normative, so implementations need not follow the guidelines listed here. However, this section is intended to help readers understand the possible strategies that can be used to implement SYCL.

### 3.12.1. Single source multiple compiler passes

With this technique, known as [SMCP](#), there are separate host and device compilers. Each SYCL source file is compiled two times: once by the host compiler and once by the device compiler. An implementation could support more than one device compiler, in which case each SYCL source file is compiled more than two times. The host compiler in this technique could be an off-the-shelf compiler with no special knowledge of SYCL, but the device compiler must be SYCL aware. The device compiler parses the source file to identify each [SYCL kernel function](#) and any [device functions](#) it calls. SYCL is designed so that this analysis can be done statically. The device compiler then generates code only for the [SYCL kernel functions](#) and the [device functions](#).

Typically, the device compilers generate header files which interface between the host compiler and the [SYCL runtime](#). Therefore, the device compiler runs first, and then the host compiler consumes these header files when generating the host code.

The device compilers in this technique generate one or more [device images](#) for the [SYCL kernel functions](#), which can be read by the [SYCL runtime](#). Each [device image](#) could either contain native ISA for a device or it could contain an intermediate language such as SPIR-V. In the later case, the [SYCL runtime](#) must translate the intermediate language into native device ISA when the [SYCL kernel function](#) is submitted to a device.

Since this technique has separate host and device compilers, there needs to be some way to associate a [SYCL kernel function](#) (which is compiled by the device compiler) with the code that invokes it (which is compiled by the host compiler). Implementations conformant to the reduced feature set ([Section B.2](#)) can do this by using the C++ type of the [SYCL kernel function](#). This type is specified via the [kernel name](#) template parameter if the [SYCL kernel function](#) is a lambda expression, or it is obtained from the class type if the [SYCL kernel function](#) is an object. Implementations conformant to the full feature set ([Section B.1](#)) do not require a [kernel name](#) at the invocation site, so they must implement some other way to make the association.



### 3.12.2. Single source single compiler pass

With this technique, known as [SSCP](#), the vendor implements a custom compiler that reads each SYCL source file only once, and that compiler generates the host code as well as the [device images](#) for the [SYCL kernel functions](#). As in the [SMCP](#) case, each [device image](#) could either contain native device ISA or an intermediate language.

### 3.12.3. Library-only implementation

It is also possible to implement SYCL purely as a library, using an off-the-shelf host compiler with no special support for SYCL. In such an implementation, each [kernel](#) may run on the host system.

## 3.13. Language restrictions in kernels

The SYCL [kernels](#) are executed on SYCL devices and all of the functions called from a SYCL kernel are going to be compiled for the device by a SYCL [device compiler](#). Due to restrictions of the heterogeneous devices where the SYCL kernel will execute, there are certain restrictions on the base C++ language features that can be used inside kernel code. For details on language restrictions please refer to [Section 5.4](#).

SYCL kernels use arguments that are captured by value in the [command group scope](#) or are passed from the host to the device using [accessors](#). Sharing data structures between host and device code imposes certain restrictions, such as using only objects that are [device copyable](#), and in general, no pointers initialized for the host can be used on the device. SYCL memory objects, such as `sycl::buffer`, `sycl::unsampled_image`, and `sycl::sampled_image`, cannot be passed to a kernel. Instead, a kernel must interact with these objects through [accessors](#). No hierarchical structures of these memory object classes are supported and any other data containers need to be converted to the SYCL data management classes using the SYCL interface. For more details on the rules for kernel parameter passing, please refer to [Section 4.12.4](#).

Pointers to [USM](#) allocations may be passed to a kernel either directly as arguments or indirectly inside of other objects. Pointers to [USM](#) allocations that are passed as kernel arguments are treated as being in the global address space.

### 3.13.1. Device copyable

The SYCL implementation may need to copy data between the host and a device or between two devices. For example, this may occur when a [command group](#) has a requirement for the contents of a buffer or when the application passes certain arguments to a [SYCL kernel function](#) (as described in [Section 4.12.4](#)). Such data must have a type that is [device copyable](#) as defined below.

Any type that is trivially copyable (as defined by the C++ core language) is implicitly device copyable.

Although implementations are not required to support device code that calls library functions from the C++ core language, some implementations may provide device support for some of these functions. If the implementation provides device support for one of the following classes, that type is also implicitly device copyable:

- `std::array<T, 0>;`
- `std::array<T, N>` if `T` is device copyable;
- `std::optional<T>` if `T` is device copyable;
- `std::pair<T1, T2>` if `T1` and `T2` are device copyable;
- `std::tuple<>;`
- `std::tuple<Types...>` if all the types in the parameter pack `Types` are device copyable;
- `std::variant<>;`
- `std::variant<Types...>` if all the types in the parameter pack `Types` are device copyable;

- `std::basic_string_view<CharT, Traits>`;
- `std::span<ElementType, Extent>` (the `std::span` type has been introduced in C++20);
- `sycl::span<ElementType, Extent>`.

If the implementation provides device support for one of the classes listed above, arrays of that class and cv-qualified versions of that class are also device copyable.



The types `std::basic_string_view<CharT, Traits>` and `std::span<ElementType, Extent>` are both view types, which reference underlying data that is not contained within their type. Although these view types are device copyable, the implementation copies just the view and not the contained data when doing an inter-device copy. In order to reference the contained data after such a copy, the application must allocate the contained data in unified shared memory (USM) that is accessible on both the host and device (or on both devices in the case of a device-to-device copy).

In addition, the implementation may allow the application to explicitly declare certain class types as device copyable. If the implementation has this support, it must predefine the preprocessor macro `SYCL_DEVICE_COPYABLE` to `1`, and it must not predefine this preprocessor macro if it does not have this support. When the implementation has this support, a class type `T` is device copyable if all of the following statements are true:

- The application defines the trait `is_device_copyable_v<T>` to `true`;
- Type `T` has at least one eligible copy constructor, move constructor, copy assignment operator, or move assignment operator;
- Each eligible copy constructor, move constructor, copy assignment operator, and move assignment operator is `public`;
- When doing an inter-device transfer of an object of type `T`, the effect of each eligible copy constructor, move constructor, copy assignment operator, and move assignment operator is the same as a bitwise copy of the object;
- Type `T` has a `public` non-deleted destructor;
- The destructor has no effect when executed on the device.

When the application explicitly declares a class type to be device copyable, arrays of that type and cv-qualified versions of that type are also device copyable, and the implementation sets the `is_device_copyable_v` trait to `true` for these array and cv-qualified types.



It is unspecified whether the implementation actually calls the copy constructor, move constructor, copy assignment operator, or move assignment operator of a class declared as `is_device_copyable_v` when doing an inter-device copy. Since these operations must all be the same as a bitwise copy, the implementation may simply copy the memory where the object resides. Likewise, it is unspecified whether the implementation actually calls the destructor for such a class on the device since the destructor must have no effect on the device.

## 3.14. Endianness support

SYCL does not mandate any particular byte order, but the byte order of the host always matches the byte order of the devices. This allows data to be copied between the host and the devices without any byte swapping.

## 3.15. Example SYCL application

Below is a more complex example application, combining some of the features described above.

```

1 #include <iostream>
2 #include <sycl/sycl.hpp>
3 using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names
4
5 // Size of the matrices
6 constexpr size_t N = 2000;
7 constexpr size_t M = 3000;
8
9 int main() {
10     // Create a queue to work on
11     queue myQueue;
12
13     // Create some 2D buffers of float for our matrices
14     buffer<float, 2> a{range<2>{N, M}};
15     buffer<float, 2> b{range<2>{N, M}};
16     buffer<float, 2> c{range<2>{N, M}};
17
18     // Launch an asynchronous kernel to initialize a
19     myQueue.submit([&](handler& cgh) {
20         // The kernel writes a, so get a write accessor on it
21         accessor A{a, cgh, write_only};
22
23         // Enqueue a parallel kernel iterating on a N*M 2D iteration space
24         cgh.parallel_for(range<2>{N, M},
25             [=](id<2> index) { A[index] = index[0] * 2 + index[1]; });
26     });
27
28     // Launch an asynchronous kernel to initialize b
29     myQueue.submit([&](handler& cgh) {
30         // The kernel writes b, so get a write accessor on it
31         accessor B{b, cgh, write_only};
32
33         // From the access pattern above, the SYCL runtime detects that this
34         // command_group is independent from the first one and can be
35         // scheduled independently
36
37         // Enqueue a parallel kernel iterating on a N*M 2D iteration space
38         cgh.parallel_for(range<2>{N, M}, [=](id<2> index) {
39             B[index] = index[0] * 2014 + index[1] * 42;
40         });
41     });
42
43     // Launch an asynchronous kernel to compute matrix addition c = a + b
44     myQueue.submit([&](handler& cgh) {
45         // In the kernel a and b are read, but c is written
46         accessor A{a, cgh, read_only};
47         accessor B{b, cgh, read_only};
48         accessor C{c, cgh, write_only};
49
50         // From these accessors, the SYCL runtime will ensure that when
51         // this kernel is run, the kernels computing a and b have completed

```

```

52
53     // Enqueue a parallel kernel iterating on a N*M 2D iteration space
54     cgh.parallel_for(range<2>{N, M},
55                     [=](id<2> index) { C[index] = A[index] + B[index]; });
56 });
57
58 // Ask for an accessor to read c from application scope. The SYCL runtime
59 // waits for c to be ready before returning from the constructor
60 host_accessor C{c, read_only};
61 std::cout << std::endl << "Result:" << std::endl;
62 for (size_t i = 0; i < N; i++) {
63     for (size_t j = 0; j < M; j++) {
64         // Compare the result to the analytic value
65         if (C[i][j] != i * (2 + 2014) + j * (1 + 42)) {
66             std::cout << "Wrong value " << C[i][j] << " on element " << i << " "
67                 << j << std::endl;
68             exit(-1);
69         }
70     }
71 }
72
73 std::cout << "Good computation!" << std::endl;
74 return 0;
75 }

```

# Chapter 4. SYCL programming interface

The SYCL programming interface provides a common abstracted feature set to one or more [SYCL backend](#) APIs. This section describes the C++ library interface to the [SYCL runtime](#) which executes across those [SYCL backends](#).

The entirety of the SYCL interface defined in this section is required to be available for any [SYCL backend](#), with the exception of the interoperability interface, which is described in general terms in this document, not pertaining to any particular [SYCL backend](#).

SYCL guarantees that all the member functions and special member functions of the SYCL classes described are thread safe.

The underlying types for all enumerations defined in this specification are implementation-defined. In addition, all enumerators within an enumeration have some implementation-defined unique value unless the specification specifically indicates a value for the enumerator.

## 4.1. Backends

The [SYCL backends](#) that can be supported by a SYCL implementation are identified using the `enum class backend`.

```
1 namespace sycl {  
2   enum class backend : /* unspecified */ {  
3     /* see below */  
4   };  
5 } // namespace sycl
```

The `enum class backend` is implementation-defined and must be populated with a unique identifier for each [SYCL backend](#) that the SYCL implementation can support. Note that the [SYCL backends](#) listed in the `enum class backend` are not guaranteed to be available in a given installation.

Each named [SYCL backend](#) enumerated in the `enum class backend` must be associated with a [SYCL backend](#) specification. Many sections of this specification will refer to the associated [SYCL backend](#) specification.

### 4.1.1. Backend macros

As the identifiers defined in `enum class backend` are implementation-defined, and the associated backends are not guaranteed to be available, a SYCL implementation must also define a preprocessor macro for each of these identifiers. If the [SYCL backend](#) is defined by the Khronos SYCL group, the name of the macro has the form `SYCL_BACKEND_<backend_name>`, where `backend_name` is the associated identifier from `backend` in all upper-case. See [Chapter 6](#) for the name of the macro if the vendor defines the [SYCL backend](#) outside of the Khronos SYCL group.

If a backend listed in the `enum class backend` is not available, the associated macro must be left undefined.

## 4.2. Generic vs non-generic SYCL

The SYCL programming API is split into two categories; generic SYCL and non-generic SYCL. Almost everything in the SYCL programming API is considered generic SYCL. However any usage of the `enum class backend` is considered non-generic SYCL and should only be used for [SYCL backend](#) specialized code paths, as the identifiers defined in `backend` are implementation-defined.

In any non-generic SYCL application code where the `backend` enum class is used, the expression must be guarded with a preprocessor `#ifdef` guard using the associated preprocessor macro to ensure that the SYCL application will compile even if the SYCL implementation does not support that SYCL backend being specialized for.

## 4.3. Header files and namespaces

SYCL provides one standard header file: `<sycl/sycl.hpp>`, which needs to be included in every translation unit that uses the SYCL programming API.

All SYCL classes, constants, types and functions defined by this specification should exist within the `::sycl` namespace.

For compatibility with SYCL 1.2.1, SYCL provides another standard header file: `<CL/sycl.hpp>`, which can be included in place of `<sycl/sycl.hpp>`. In that case, all SYCL classes, constants, types and functions defined by this specification should exist within the `::cl::sycl` C++ namespace.

For consistency, the programming API will only refer to the `<sycl/sycl.hpp>` header and the `::sycl` namespace, but this should be considered synonymous with the SYCL 1.2.1 header and namespace.

Include paths starting with `"sycl/ext/"` and `"sycl/backend/"` are reserved for extensions to SYCL and for backend interop headers respectively. Other include paths starting with `"sycl/"` and the `sycl::detail` namespace are reserved for implementation details.

When a SYCL backend is defined by the Khronos SYCL group, functionality for that SYCL backend is available via the header `"sycl/backend/<backend_name>.hpp"`, and all SYCL backend-specific functionality is made available in the namespace `sycl::<backend_name>` where `<backend_name>` is the name of the SYCL backend as defined in the SYCL backend specification.

Chapter 6 defines the allowable header files and namespaces for any extensions that a vendor may provide, including any SYCL backend that the vendor may define outside of the Khronos SYCL group.

Unless otherwise specified, the behavior of a SYCL program is undefined if it adds any entity to namespace `sycl` or to a namespace within namespace `sycl`.

## 4.4. Class availability

In SYCL some SYCL runtime classes are available to the SYCL application, some are available within a SYCL kernel function and some are available on both and can be passed as arguments to a SYCL kernel function.

Each of the following SYCL runtime classes: `buffer`, `buffer_allocator`, `context`, `device`, `device_image`, `event`, `exception`, `handler`, `host_accessor`, `host_sampled_image_accessor`, `host_unsampled_image_accessor`, `id`, `image_allocator`, `kernel`, `kernel_id`, `marray`, `kernel_bundle`, `nd_range`, `platform`, `queue`, `range`, `sampled_image`, `image_sampler`, `stream`, `unsampled_image` and `vec` must be available to the host application.

Each of the following SYCL runtime classes: `accessor`, `atomic_ref`, `device_event`, `group`, `h_item`, `id`, `item`, `local_accessor`, `marray`, `multi_ptr`, `nd_item`, `range`, `reducer`, `sampled_image_accessor`, `stream`, `sub_group`, `unsampled_image_accessor` and `vec` must be available within a SYCL kernel function.

## 4.5. Common interface

When a dimension template parameter is used in SYCL classes, it is defaulted as 1 in most cases.

### 4.5.1. Backend interoperability

Many of the [SYCL runtime](#) classes may be implemented such that they encapsulate an object unique to the [SYCL backend](#) that underpins the functionality of that class. Where appropriate, these classes may provide an interface for interoperating between the [SYCL runtime](#) object and the [native backend object](#) in order to support interoperability within an application between SYCL and the associated [SYCL backend API](#).

There are three forms of interoperability with [SYCL runtime](#) classes: interoperability on the [SYCL application](#) with the [SYCL backend API](#), interoperability within a [SYCL kernel function](#) with the equivalent kernel language types of the [SYCL backend](#), and interoperability within a [host task](#) with the [interop\\_handle](#).

[SYCL application](#) interoperability, [SYCL kernel function](#) interoperability and [host task](#) interoperability are provided via different interfaces and may have different behavior for the same SYCL object.

[SYCL application](#) interoperability may be provided for [buffer](#), [context](#), [device](#), [device\\_image](#), [event](#), [kernel](#), [kernel\\_bundle](#), [platform](#), [queue](#), [sampled\\_image](#), and [unsampled\\_image](#).

[SYCL kernel function](#) interoperability may be provided for [accessor](#), [device\\_event](#), [local\\_accessor](#), [sampled\\_image\\_accessor](#), [stream](#) and [unsampled\\_image\\_accessor](#) inside [kernel scope](#) only and is not available outside of that scope.

[host task](#) interoperability may be provided for [accessor](#), [sampled\\_image\\_accessor](#), [unsampled\\_image\\_accessor](#), [queue](#), [device](#), [context](#) inside the scope of a [host task](#) only, see [Section 4.10](#).

Support for [SYCL backend](#) interoperability is optional and therefore not required to be provided by a SYCL implementation. A SYCL application using [SYCL backend](#) interoperability is considered to be non-generic SYCL.

Details on the interoperability for a given [SYCL backend](#) are available on the [SYCL backend](#) specification document for that [SYCL backend](#).

#### 4.5.1.1. Type traits [backend\\_traits](#)

```

1 namespace sycl {
2
3 template <backend Backend> class backend_traits {
4 public:
5     template <class T> using input_type = /* see below */;
6
7     template <class T> using return_type = /* see below */;
8 };
9
10 template <backend Backend, typename SyclType>
11 using backend_input_t =
12     typename backend_traits<Backend>::template input_type<SyclType>;
13
14 template <backend Backend, typename SyclType>
15 using backend_return_t =
16     typename backend_traits<Backend>::template return_type<SyclType>;
17
18 } // namespace sycl

```

A series of type traits are provided for [SYCL backend](#) interoperability, defined in the [backend\\_traits](#) class.



A specialization of `backend_traits` must be provided for each named SYCL backend enumerated in the enum class `backend` that is available at compile time.

- For each SYCL runtime class `T` which supports SYCL application interoperability with the SYCL backend, a specialization of `input_type` must be defined as the type of SYCL application interoperability native backend object associated with `T` for the SYCL backend, specified in the SYCL backend specification. `input_type` is used when constructing SYCL objects from backend specific native objects. See the relevant backend specification for details.
- For each SYCL runtime class `T` which supports SYCL application interoperability with the SYCL backend, a specialization of `return_type` must be defined as the type of SYCL application interoperability native backend object associated with `T` for the SYCL backend, specified in the SYCL backend specification. `return_type` is used when retrieving the backend specific native object from a SYCL object. See the relevant backend specification for details.
- For each SYCL runtime class `T` which supports kernel function interoperability with the SYCL backend, a specialization of `return_type` within `backend_traits` must be defined as the type of the kernel function interoperability native backend object associated with `T` for the SYCL backend, specified in the backend specification. See the relevant backend specification for details.

The type alias `backend_input_t` is provided to enable less verbose access to the `input_type` type within `backend_traits` for a specific SYCL object of type `T`. The type alias `backend_return_t` is provided to enable less verbose access to the `return_type` type within `backend_traits` for a specific SYCL object of type `T`.

#### 4.5.1.2. Template function `get_native`

```
1 namespace sycl {
2
3 template <backend Backend, class T>
4 backend_return_t<Backend, T> get_native(const T& syclObject);
5
6 } // namespace sycl
```

For each SYCL runtime class `T` which supports SYCL application interoperability, a specialization of `get_native` must be defined, which takes an instance of `T` and returns a SYCL application interoperability native backend object associated with `syclObject` which can be used for SYCL application interoperability. The lifetime of the object returned is backend-defined and specified in the backend specification.

For each SYCL runtime class `T` which supports kernel function interoperability, a specialization of `get_native` must be defined, which takes an instance of `T` and returns the kernel function interoperability native backend object associated with `syclObject` which can be used for kernel function interoperability. The availability and behavior of these template functions are defined by the SYCL backend specification document.

The `get_native` function must throw an exception with the `errc::backend_mismatch` error code if the backend of the SYCL object doesn't match the target backend.

#### 4.5.1.3. Template functions `make_*`

```
1 namespace sycl {
2
3 template <backend Backend>
4 platform make_platform(const backend_input_t<Backend, platform>& backendObject);
5
6 template <backend Backend>
7 device make_device(const backend_input_t<Backend, device>& backendObject);
8
```



```

9  template <backend Backend>
10 context make_context(const backend_input_t<Backend, context>& backendObject,
11                      const async_handler asyncHandler = {});
12
13 template <backend Backend>
14 queue make_queue(const backend_input_t<Backend, queue>& backendObject,
15                 const context& targetContext,
16                 const async_handler asyncHandler = {});
17
18 template <backend Backend>
19 event make_event(const backend_input_t<Backend, event>& backendObject,
20                 const context& targetContext);
21
22 template <backend Backend, typename T, int Dimensions = 1,
23          typename AllocatorT = buffer_allocator<std::remove_const_t<T>>>
24 buffer<T, Dimensions, AllocatorT>
25 make_buffer(const backend_input_t<Backend, buffer<T, Dimensions, AllocatorT>>&
26             backendObject,
27             const context& targetContext, event availableEvent);
28
29 template <backend Backend, typename T, int Dimensions = 1,
30          typename AllocatorT = buffer_allocator<std::remove_const_t<T>>>
31 buffer<T, Dimensions, AllocatorT>
32 make_buffer(const backend_input_t<Backend, buffer<T, Dimensions, AllocatorT>>&
33             backendObject,
34             const context& targetContext);
35
36 template <backend Backend, int Dimensions = 1,
37          typename AllocatorT = sycl::image_allocator>
38 sampled_image<Dimensions, AllocatorT> make_sampled_image(
39     const backend_input_t<Backend, sampled_image<Dimensions, AllocatorT>>&
40     backendObject,
41     const context& targetContext, image_sampler imageSampler,
42     event availableEvent);
43
44 template <backend Backend, int Dimensions = 1,
45          typename AllocatorT = sycl::image_allocator>
46 sampled_image<Dimensions, AllocatorT> make_sampled_image(
47     const backend_input_t<Backend, sampled_image<Dimensions, AllocatorT>>&
48     backendObject,
49     const context& targetContext, image_sampler imageSampler);
50
51 template <backend Backend, int Dimensions = 1,
52          typename AllocatorT = sycl::image_allocator>
53 unsampled_image<Dimensions, AllocatorT> make_unsampled_image(
54     const backend_input_t<Backend, unsampled_image<Dimensions, AllocatorT>>&
55     backendObject,
56     const context& targetContext, event availableEvent);
57
58 template <backend Backend, int Dimensions = 1,
59          typename AllocatorT = sycl::image_allocator>
60 unsampled_image<Dimensions, AllocatorT> make_unsampled_image(
61     const backend_input_t<Backend, unsampled_image<Dimensions, AllocatorT>>&
62     backendObject,
63     const context& targetContext);
64

```

```

65 template <backend Backend, bundle_state State>
66 kernel_bundle<State> make_kernel_bundle(
67     const backend_input_t<Backend, kernel_bundle<State>>& backendObject,
68     const context& targetContext);
69
70 template <backend Backend>
71 kernel make_kernel(const backend_input_t<Backend, kernel>& backendObject,
72                  const context& targetContext);
73
74 } // namespace sycl

```

For each SYCL runtime class **T** which supports SYCL application interoperability, a specialization of the appropriate template function `make_{sycl_class}` where `{sycl_class}` is the class name of **T**, must be defined, which takes a SYCL application interoperability native backend object and constructs and returns an instance of **T**. The availability and behavior of these template functions are defined by the SYCL backend specification document.

Overloads of the `make_{sycl_class}` function which take a SYCL context object as an argument must throw an exception with the `errc::backend_mismatch` error code if the backend of the provided SYCL context doesn't match the target backend.

## 4.5.2. Common reference semantics

Each of the following SYCL runtime classes: `accessor`, `buffer`, `context`, `device`, `device_image`, `event`, `host_accessor`, `host_sampled_image_accessor`, `host_unsampled_image_accessor`, `kernel`, `kernel_id`, `kernel_bundle`, `local_accessor`, `platform`, `queue`, `sampled_image`, `sampled_image_accessor`, `stream`, `unsampled_image` and `unsampled_image_accessor` must obey the following statements, where **T** is the runtime class type:

- **T** must be copy constructible and copy assignable in the host application and within SYCL kernel functions in the case that **T** is a valid kernel argument. Any instance of **T** that is constructed as a copy of another instance, via either the copy constructor or copy assignment operator, must behave as-if it were the original instance and as-if any action performed on it were also performed on the original instance and must represent the same underlying native backend object as the original instance where applicable.
- **T** must be destructible in the host application and within SYCL kernel functions in the case that **T** is a valid kernel argument. When any instance of **T** is destroyed, including as a result of the copy assignment operator, any behavior specific to **T** that is specified as performed on destruction is only performed if this instance is the last remaining host copy, in accordance with the above definition of a copy.
- **T** must be move constructible and move assignable in the host application and within SYCL kernel functions in the case that **T** is a valid kernel argument. Any instance of **T** that is constructed as a move of another instance, via either the move constructor or move assignment operator, must replace the original instance rendering said instance invalid and must represent the same underlying native backend object as the original instance where applicable.
- **T** must be equality comparable in the host application. Equality between two instances of **T** (i.e. `a == b`) must be true if one instance is a copy of the other and non-equality between two instances of **T** (i.e. `a != b`) must be true if neither instance is a copy of the other, in accordance with the above definition of a copy, unless either instance has become invalidated by a move operation. By extension of the requirements above, equality on **T** must guarantee to be reflexive (i.e. `a == a`), symmetric (i.e. `a == b` implies `b == a` and `a != b` implies `b != a`) and transitive (i.e. `a == b && b == c` implies `c == a`).
- A specialization of `std::hash` for **T** must exist in the host application that returns a unique value such that if two instances of **T** are equal, in accordance with the above definition, then their resulting hash values are also equal and subsequently if two hash values are not equal, then their corresponding instances are also not equal, in accordance with the above definition.

Some [SYCL runtime](#) classes will have additional behavior associated with copy, movement, assignment or destruction semantics. If these are specified they are in addition to those specified above unless stated otherwise.

Each of the runtime classes mentioned above must provide a common interface of special member functions in order to fulfill the copy, move, destruction requirements and hidden friend functions in order to fulfill the equality requirements.

A hidden friend function is a function first declared via a **friend** declaration with no additional out of class or namespace scope declarations. Hidden friend functions are only visible to ADL (Argument Dependent Lookup) and are hidden from qualified and unqualified lookup. Hidden friend functions have the benefits of avoiding accidental implicit conversions and faster compilation.

These common special member functions and hidden friend functions are described in [Table 7](#) and [Table 8](#) respectively.

```

1 namespace sycl {
2
3 class T {
4     ...
5
6     public : T(const T& rhs);
7
8     T(T&& rhs);
9
10    T& operator=(const T& rhs);
11
12    T& operator=(T&& rhs);
13
14    ~T();
15
16    ...
17
18    friend bool
19    operator==(const T& lhs, const T& rhs) { /* ... */
20 }
21
22 friend bool operator!=(const T& lhs, const T& rhs) { /* ... */ }
23
24 ...
25 };
26 } // namespace sycl

```

Table 7. Common special member functions for reference semantics

Special member function	Description
<code>T(const T&amp; rhs)</code>	Constructs a <b>T</b> instance as a copy of the RHS SYCL <b>T</b> in accordance with the requirements set out above.
<code>T(T&amp;&amp; rhs)</code>	Constructs a SYCL <b>T</b> instance as a move of the RHS SYCL <b>T</b> in accordance with the requirements set out above.
<code>T&amp; operator=(const T&amp; rhs)</code>	Assigns this SYCL <b>T</b> instance with a copy of the RHS SYCL <b>T</b> in accordance with the requirements set out above.

Special member function	Description
<code>T&amp; operator=(T&amp;&amp; rhs)</code>	Assigns this SYCL <b>T</b> instance with a move of the RHS SYCL <b>T</b> in accordance with the requirements set out above.
<code>~T()</code>	Destroys this SYCL <b>T</b> instance in accordance with the requirements set out in <a href="#">Section 4.5.2</a> . On destruction of the last copy, may perform additional lifetime related operations required for the underlying <a href="#">native backend object</a> specified in the <a href="#">SYCL backend</a> specification document, if this SYCL <b>T</b> instance was originally constructed using one of the backend interoperability <code>make_*</code> functions specified in <a href="#">Section 4.5.1.3</a> . See the relevant backend specification for details.

Table 8. Common hidden friend functions for reference semantics

Hidden friend function	Description
<code>bool operator==(const T&amp; lhs, const T&amp; rhs)</code>	Returns true if this LHS SYCL <b>T</b> is equal to the RHS SYCL <b>T</b> in accordance with the requirements set out above, otherwise returns false.
<code>bool operator!=(const T&amp; lhs, const T&amp; rhs)</code>	Returns true if this LHS SYCL <b>T</b> is not equal to the RHS SYCL <b>T</b> in accordance with the requirements set out above, otherwise returns false.

### 4.5.3. Common by-value semantics

Each of the following [SYCL runtime](#) classes: `id`, `range`, `item`, `nd_item`, `h_item`, `group`, `sub_group` and `nd_range` must follow the following statements, where **T** is the runtime class type:

- T** must be default copy constructible and copy assignable in the host application (in the case where **T** is available on the host) and within SYCL kernel functions.
- T** must be default destructible in the host application (in the case where **T** is available on the host) and within SYCL kernel functions.
- T** must be default move constructible and default move assignable in the host application (in the case where **T** is available on the host) and within SYCL kernel functions.
- T** must be equality comparable in the host application (in the case where **T** is available on the host) and within SYCL kernel functions. Equality between two instances of **T** (i.e. `a == b`) must be true if the value of all members are equal and non-equality between two instances of **T** (i.e. `a != b`) must be true if the value of any members are not equal, unless either instance has become invalidated by a move operation. By extension of the requirements above, equality on **T** must guarantee to be reflexive (i.e. `a == a`), symmetric (i.e. `a == b` implies `b == a` and `a != b` implies `b != a`) and transitive (i.e. `a == b && b == c` implies `c == a`).

Some [SYCL runtime](#) classes will have additional behavior associated with copy, movement, assignment or destruction semantics. If these are specified they are in addition to those specified above unless stated otherwise.

Each of the runtime classes mentioned above must provide a common interface of special member functions and member functions in order to fulfill the copy, move, destruction and equality requirements, following the [rule of five](#) and the [rule of zero](#).

These common special member functions and hidden friend functions are described in [Table 9](#) and [Table 10](#) respectively.

```
1 namespace sycl {
2
```

```

3  class T {
4      ...
5
6      public
7      :
8      // If any of the following five special member functions are declared,
9      // then all five of them should be explicitly declared (see rule of
10     // five).
11     //
12     // Otherwise, none of them should be explicitly declared
13     // (see rule of zero).
14
15     // T(const T &rhs);
16
17     // T(T &&rhs);
18
19     // T &operator=(const T &rhs);
20
21     // T &operator=(T &&rhs);
22
23     // ~T();
24
25     ...
26
27     friend bool
28     operator==(const T& lhs, const T& rhs) { /* ... */
29 }
30
31 friend bool operator!=(const T& lhs, const T& rhs) { /* ... */ }
32
33 ...
34 };
35 } // namespace sycl

```

Table 9. Common special member functions for by-value semantics

Special member function (see <a href="#">rule of five</a> and <a href="#">rule of zero</a> )	Description
<code>T(const T&amp; rhs);</code>	Copy constructor.
<code>T(T&amp;&amp; rhs);</code>	Move constructor.
<code>T&amp; operator=(const T&amp; rhs);</code>	Copy assignment operator.
<code>T&amp; operator=(T&amp;&amp; rhs);</code>	Move assignment operator.
<code>~T();</code>	Destructor.

Table 10. Common hidden friend functions for by-value semantics

Hidden friend function	Description
<code>bool operator==(const T&amp; lhs, const T&amp; rhs)</code>	Returns true if this LHS SYCL <b>T</b> is equal to the RHS SYCL <b>T</b> in accordance with the requirements set out above, otherwise returns false.
<code>bool operator!=(const T&amp; lhs, const T&amp; rhs)</code>	Returns true if this LHS SYCL <b>T</b> is not equal to the RHS SYCL <b>T</b> in accordance with the requirements set out above, otherwise returns false.

## 4.5.4. Properties

Each of the following [SYCL runtime](#) classes: `accessor`, `buffer`, `host_accessor`, `host_sampled_image_accessor`, `host_unsampled_image_accessor`, `context`, `local_accessor`, `queue`, `sampled_image`, `sampled_image_accessor`, `stream`, `unsampled_image`, `unsampled_image_accessor` and `usm_allocator` provide an optional parameter in each of their constructors to provide a `property_list` which contains zero or more properties. Each of those properties augments the semantics of the class with a particular feature. Each of those classes must also provide `has_property` and `get_property` member functions for querying for a particular property.

The listing below illustrates the usage of various buffer properties, described in [Section 4.7.2.2](#).

The example illustrates how using properties does not affect the type of the object, thus, does not prevent the usage of SYCL objects in containers.

```

1 {
2   context myContext;
3
4   std::vector<buffer<int, 1>> bufferList{
5       buffer<int, 1>{ptr, rng},
6       buffer<int, 1>{ptr, rng, property::use_host_ptr{}},
7       buffer<int, 1>{ptr, rng, property::context_bound{myContext}}};
8
9   for (auto& buf : bufferList) {
10      if (buf.has_property<property::context_bound>()) {
11         auto prop = buf.get_property<property::context_bound>();
12         assert(myContext == prop.get_context());
13      }
14   }
15 }
```

Each property is represented by a unique class and an instance of a property is an instance of that type. Some properties can be default constructed while others will require an argument on construction. A property may be applicable to more than one class, however some properties may not be compatible with each other. See the requirements for the properties of the SYCL `buffer` class, SYCL `unsampled_image` class and SYCL `sampled_image` class in [Table 23](#) and [Table 30](#) respectively.

Properties can be passed to a [SYCL runtime](#) class via an instance of `property_list`. These properties get tied to the [SYCL runtime](#) class instance and copies of the object will contain the same properties.

A SYCL implementation or a [SYCL backend](#) may provide additional properties other than those defined here, provided they are defined in accordance with the requirements described in [Section 4.3](#).

### 4.5.4.1. Properties interface

Each of the runtime classes mentioned above must provide a common interface of member functions in order to fulfill the property interface requirements.

A synopsis of the common properties interface, the SYCL `property_list` class and the SYCL property classes is provided below. The member functions of the common properties interface are listed in [Table 12](#). The constructors of the SYCL `property_list` class are listed in [Table 13](#).

```

1 namespace sycl {
2
3 template <typename Property> struct is_property;
4
5 template <typename Property>
6 inline constexpr bool is_property_v = is_property<Property>::value;
7
8 template <typename Property, typename SyclObject> struct is_property_of;
9
10 template <typename Property, typename SyclObject>
11 inline constexpr bool is_property_of_v =
12     is_property_of<Property, SyclObject>::value;
13
14 class T {
15     ...
16
17     template <typename Property>
18     bool has_property() const noexcept;
19
20     template <typename Property> Property get_property() const;
21
22     ...
23 };
24
25 class property_list {
26 public:
27     template <typename... Properties> property_list(Properties... props);
28 };
29 } // namespace sycl

```

Table 11. Traits for properties

Traits	Description
<pre>template &lt;typename Property&gt; struct is_property</pre>	An explicit specialization of <code>is_property</code> that inherits from <code>std::true_type</code> must be provided for each property, where <code>Property</code> is the class defining the property. This includes both standard properties described in this specification and any additional non-standard properties defined by an implementation. All other specializations of <code>is_property</code> must inherit from <code>std::false_type</code> .
<pre>template &lt;typename Property&gt; inline constexpr bool is_property_v;</pre>	Variable containing value of <code>is_property&lt;Property&gt;</code> .



Traits	Description
<pre>template &lt;typename Property, SyclObject&gt; struct is_property_of</pre>	An explicit specialization of <code>is_property_of</code> that inherits from <code>std::true_type</code> must be provided for each property that can be used in constructing a given SYCL class, where <code>Property</code> is the class defining the property and <code>SyclObject</code> is the SYCL class. This includes both standard properties described in this specification and any additional non-standard properties defined by an implementation. All other specializations of <code>is_property_of</code> must inherit from <code>std::false_type</code> .
<pre>template &lt;typename Property, SyclObject&gt; inline constexpr bool is_property_of_v;</pre>	Variable containing value of <code>is_property_of&lt;Property, SyclObject&gt;</code> .

Table 12. Common member functions of the SYCL `property` interface

Member function	Description
<pre>template &lt;typename Property&gt; bool has_property() const noexcept</pre>	Returns true if <code>T</code> was constructed with the property specified by <code>Property</code> . Returns false if it was not.
<pre>template &lt;typename Property&gt; Property get_property() const</pre>	Returns a copy of the property of type <code>Property</code> that <code>T</code> was constructed with. Must throw an <code>exception</code> with the <code>errc::invalid</code> error code if <code>T</code> was not constructed with the <code>Property</code> property.

Table 13. Constructors of the SYCL `property_list` class

Constructor	Description
<pre>template &lt;typename... PropertyN&gt; property_list(PropertyN... props)</pre>	Available only when: <code>is_property&lt;property&gt;::value</code> evaluates to <code>true</code> where <code>property</code> is each property in <code>PropertyN</code> . Construct a SYCL <code>property_list</code> with zero or more properties.

## 4.6. SYCL runtime classes

### 4.6.1. Device selection

Since a system can have several SYCL-compatible devices attached, it is useful to have a way to select a specific device or a set of devices to construct a specific object such as a `device` (see [Section 4.6.4](#)) or a `queue` (see [Section 4.6.5](#)), or perform some operations on a device subset.

Device selection is done either by already having a specific instance of a `device` (see [Section 4.6.4](#)) or by providing a `device selector` which is a ranking function that will give an integer ranking value to all the devices on the system.

#### 4.6.1.1. Device selector

The interface for a `device selector` is any object that meets the C++ named requirement `Callable`, taking a parameter of type `const device &` and returning a value that is implicitly convertible to `int`.

At any point where the SYCL runtime needs to select a SYCL `device` using a `device selector`, the system queries all `root devices` from all SYCL `backends` in the system, calls the `device selector` on each device




and selects the one which returns the highest score. If the highest value is strictly negative no device is selected.

In places where only one device has to be picked and the high score is obtained by more than one device, then one of the tied devices will be returned, but which one is not defined and may depend on enumeration order, for example, outside the control of the SYCL runtime.

Some predefined [device selectors](#) are provided by the system as described on [Table 14](#) in a header file with some definition similar to the following:

Table 14. Standard device selectors included with all SYCL implementations

SYCL device selectors	Description
<code>default_selector_v</code>	<p>Select a SYCL <b>device</b> from any supported <a href="#">SYCL backend</a> based on an implementation-defined heuristic. Since all implementations must support at least one device, this selector must always return a device.</p> <div>  <p>Implementations may choose to return an emulated device (with <code>aspect::emulated</code>) as a fallback if there is no physical device available on the system.</p> </div>
<code>gpu_selector_v</code>	<p>Select a SYCL <b>device</b> from any supported <a href="#">SYCL backend</a> for which the device type is <code>info::device_type::gpu</code>. The SYCL class constructor using it must throw an <b>exception</b> with the <code>errc::runtime</code> error code if no device matching this requirement can be found.</p>
<code>accelerator_selector_v</code>	<p>Select a SYCL <b>device</b> from any supported <a href="#">SYCL backend</a> for which the device type is <code>info::device_type::accelerator</code>. The SYCL class constructor using it must throw an <b>exception</b> with the <code>errc::runtime</code> error code if no device matching this requirement can be found.</p>
<code>cpu_selector_v</code>	<p>Select a SYCL <b>device</b> from any supported <a href="#">SYCL backend</a> for which the device type is <code>info::device_type::cpu</code>. The SYCL class constructor using it must throw an <b>exception</b> with the <code>errc::runtime</code> error code if no device matching this requirement can be found.</p>

SYCL device selectors	Description
<pre> __unspecified_callable__ aspect_selector(const std::vector&lt;aspect&gt;&amp; aspectList,                 const std::vector&lt;aspect&gt;&amp; denyList = {});  template &lt;typename... AspectList&gt; __unspecified_callable__ aspect_selector (AspectList... aspectList);  template &lt;aspect... AspectList&gt; __unspecified_callable__ aspect_selector(); </pre>	<p>The free function <code>aspect_selector</code> has several overloads, each of which returns a selector object that selects a SYCL <code>device</code> from any supported <code>SYCL backend</code> which contains all the requested aspects, i.e. for the specific device <code>dev</code> and each aspect <code>devAspect</code> from <code>aspectList</code> <code>dev.has(devAspect)</code> equals <code>true</code>. If no aspects are passed in, the generated selector behaves like <code>default_selector_v</code>.</p> <p>Required aspects can be passed in as a vector, as function arguments, or as template parameters, depending on the function overload. The function overload that takes <code>aspectList</code> as a vector takes another vector argument <code>denyList</code> where the user can specify all the aspects that have to be avoided, i.e. for the specific device <code>dev</code> and each aspect <code>devAspect</code> from <code>denyList</code> <code>dev.has(devAspect)</code> equals <code>false</code>.</p> <p>The SYCL class constructor using the generated selector must throw an <code>exception</code> with the <code>errc::runtime</code> error code if no device matching this requirement can be found. There are multiple overloads of this function, please refer to <a href="#">[header:device-selector]</a> for full definitions and to <a href="#">[example:aspect-selector]</a> for examples.</p>

```

1 namespace sycl {
2
3 // Predefined device selectors
4 __unspecified__ default_selector_v;
5 __unspecified__ cpu_selector_v;
6 __unspecified__ gpu_selector_v;
7 __unspecified__ accelerator_selector_v;
8
9 // Predefined types for compatibility with old SYCL 1.2.1 device selectors
10 // Deprecated in SYCL 2020
11 using default_selector = __unspecified__;
12 using cpu_selector = __unspecified__;
13 using gpu_selector = __unspecified__;
14 using accelerator_selector = __unspecified__;
15
16 // Returns a selector that selects a device based on desired aspects
17 __unspecified_callable__
18 aspect_selector(const std::vector<aspect>& aspectList,
19                const std::vector<aspect>& denyList = {});
20 template <class... AspectList>
21 __unspecified_callable__ aspect_selector(AsspectList... aspectList);
22 template <aspect... AspectList> __unspecified_callable__ aspect_selector();
23
24 } // namespace sycl

```

Typical examples of default and user-provided `device selectors` could be:

```

1  sycl::device my_gpu { sycl::gpu_selector_v };
2
3  sycl::queue my_accelerator { sycl::accelerator_selector_v };
4
5  int prefer_my_vendor(const sycl::device& d) {
6    // Return 1 if the vendor name is "MyVendor" or 0 else.
7    // 0 does not prevent another device to be picked as a second choice
8    return d.get_info<info::device::vendor>() == "MyVendor";
9  }
10
11 // Get the preferred device or another one if not available
12 sycl::device preferred_device { prefer_my_vendor };
13
14 // This throws if there is no such device in the system
15 sycl::queue half_precision_controller {
16   // Can use a lambda as a device ranking function.
17   // Returns a negative number to fail in the case there is no such device
18   [] (auto& d) { return d.has(sycl::aspect::fp16) ? 1 : -1; }
19 };
20
21 // To ease porting SYCL 1.2.1 code, there are types whose
22 // construction leads to the equivalent predefined device selector
23 sycl::queue my_old_style_gpu { sycl::gpu_selector {} };

```

Examples of using **aspect\_selector**:

```

1  using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names
2
3  // Unrestrained selection, equivalent to default_selector_v
4  auto dev0 = device{aspect_selector()};
5
6  // Pass aspects in a vector
7  // Only accept CPUs that support half
8  auto dev1 = device{aspect_selector(std::vector{aspect::cpu, aspect::fp16})};
9
10 // Pass aspects without a vector
11 // Only accept GPUs that support half
12 auto dev2 = device{aspect_selector(aspect::gpu, aspect::fp16)};
13
14 // Pass aspects as compile-time parameters
15 // Only accept devices that can be debugged on host and support half
16 auto dev3 = device{aspect_selector<aspect::host_debuggable, aspect::fp16>()};
17
18 // Pass aspects in an allowlist and a denylist
19 // Only accept devices that support half and double floating point precision,
20 // but exclude emulated devices and devices of type "custom"
21 auto dev4 = device{aspect_selector(
22   std::vector{aspect::fp16, aspect::fp64},
23   std::vector{aspect::emulated, aspect::custom}
24 )};

```



In SYCL 1.2.1 the predefined device selectors were actually types that had to be instantiated to be used. Now they are just instances. To simplify porting code using the old type instantiations, a backward-compatible API is still provided, though deprecated, such as

`sycl::default_selector`. The new predefined device selectors have their new names appended with "\_v" to avoid conflicts, thus following the naming style used by traits in the C++ standard library. There is no requirement for the implementation to have for example `sycl::gpu_selector_v` being an instance of `sycl::gpu_selector`.

## 4.6.2. Platform class

The `platform` class encapsulates a single SYCL platform on which kernel functions may be executed. A platform must be associated with a single [SYCL backend](#).

A `platform` also contains a set of devices that are associated with the same [SYCL backend](#). A platform may contain no devices.

All member functions of the `platform` class are synchronous and errors are handled by throwing synchronous SYCL exceptions.

The execution environment for a SYCL application has a fixed number of platforms which does not vary as the application executes. The application can get a list of all these platforms via `platform::get_platforms`, and the order of the platform objects is the same each time the application calls that function. The `platform` class also provides constructors, but constructing a new `platform` instance merely creates a new object that is a copy of one of the objects returned by `platform::get_platforms`.

Each platform has an associated default context which contains all of the [root devices](#) in the platform. This default context does not have an asynchronous error handler. Applications can retrieve a copy of this default `context` object, for example, by constructing a `queue`. These copies follow the common reference semantics, as though they are all copies of an internal per-platform `context` object representing the platform's default context.

The `platform` class provides the common reference semantics as defined in [Section 4.5.2](#).

```
namespace sycl {
class platform {
public:
    platform();

    template <typename DeviceSelector>
    explicit platform(const DeviceSelector& deviceSelector);

    /* -- common interface members -- */

    backend get_backend() const noexcept;

    std::vector<device>
        get_devices(info::device_type type = info::device_type::all) const;

    template <typename Param> typename Param::return_type get_info() const;

    template <typename Param>
    typename Param::return_type get_backend_info() const;

    bool has(aspect asp) const;

    bool has_extension(const std::string& extension) const; // Deprecated

    static std::vector<platform> get_platforms();
};
```

---

```
} // namespace sycl
```

---

### 4.6.2.1. Constructors

#### Default constructor

```
platform()
```

*Effects:* Constructs a `platform` object that is a copy of the platform which contains the device returned by `default_selector_v`.

#### Selector constructor

```
template <typename DeviceSelector>
explicit platform(const DeviceSelector& selector)
```

*Constraints:* The `DeviceSelector` must be a type that satisfies the requirements of a `device selector` as defined in [Section 4.6.1.1](#).

*Effects:* The `selector` is called for every `root device` as described in [Section 4.6.1.1](#). Constructs a `platform` object that is a copy of the platform which contains the device that is selected by `selector`.

### 4.6.2.2. Member functions

#### `platform::get_backend`

```
backend get_backend() const noexcept
```

*Returns:* The `SYCL backend` that is associated with this platform.

#### `platform::get_info`

```
template <typename Param>
typename Param::return_type get_info() const
```

*Constraints:* The `Param` must be an information descriptor for the platform class.

Each information descriptor specifies the return value and may also specify preconditions, exceptions that are thrown, etc. See [Section 4.6.2.4](#) for the platform information descriptors that are defined by the [core SYCL specification](#).

#### `platform::get_backend_info`

```
template <typename Param>
typename Param::return_type get_backend_info() const
```

*Constraints:* The `Param` must be a backend information descriptor for the platform class.

*Throws:* An `exception` with the `errc::backend_mismatch` error code if the backend that corresponds with

**Param** is different from the backend that is associated with this platform.

Each information descriptor specifies the return value and may also specify preconditions, additional exceptions that are thrown, etc.

#### platform::has

```
bool has(aspect asp) const
```

*Returns:* The value **true** if all of the devices associated with this platform have the given **aspect**. Returns the value **false** if this platform does not contain any devices.

#### platform::has\_extension

```
bool has_extension(const std::string& extension) const
```

Deprecated by SYCL 2020.

[*Note:* Use **platform::has** instead. — *end note*]

*Returns:* The value **true** if this platform supports the extension queried by the **extension** parameter. A platform only supports an extension if all associated devices support that extension. Returns **false** if this platform does not contain any devices.

#### platform::get\_devices

```
std::vector<device>
get_devices(info::device_type type = info::device_type::all) const
```

*Returns:* A **std::vector** containing all of the **root devices** associated with this platform which have the device type specified by **type**.

### 4.6.2.3. Static member functions

#### platform::get\_platforms

```
static std::vector<platform> get_platforms()
```

*Returns:* A **std::vector** containing all of the platforms from all **backends** that are available in the system.

### 4.6.2.4. Information descriptors

This section describes the information descriptors that can be used as the **Param** template parameter to **platform::get\_info**. When the description has a *Returns*, *Throws*, etc. paragraph, this indicates the value returned by or the exceptions thrown by the **platform::get\_info** function.

**info::platform::version**

```
namespace sycl::info::platform {
  struct version {
    using return_type = std::string;
  };
} // namespace sycl::info::platform
```

*Remarks:* Template parameter to `platform::get_info`.

*Returns:* An implementation-defined platform version string.

**info::platform::name**

```
namespace sycl::info::platform {
  struct name {
    using return_type = std::string;
  };
} // namespace sycl::info::platform
```

*Remarks:* Template parameter to `platform::get_info`.

*Returns:* An implementation-defined name for this platform.

**info::platform::vendor**

```
namespace sycl::info::platform {
  struct vendor {
    using return_type = std::string;
  };
} // namespace sycl::info::platform
```

*Remarks:* Template parameter to `platform::get_info`.

*Returns:* An implementation-defined name for the vendor providing this platform.

**info::platform::extensions**

```
namespace sycl::info::platform {
  struct extensions {
    using return_type = std::vector<std::string>;
  };
} // namespace sycl::info::platform
```

Deprecated by SYCL 2020.

[*Note:* Use `device::get_info()` with `info::device::aspects` instead. — *end note*]

*Remarks:* Template parameter to `platform::get_info`.

*Returns:* The extensions supported by this platform. Returns an empty list if this platform does not contain any devices.

### 4.6.3. Context class

The `context` class represents a SYCL context. A context represents the runtime data structures and state required by a [SYCL backend](#) API to interact with a group of devices associated with a platform.

All member functions of the `context` class are synchronous and errors are handled by throwing synchronous SYCL exceptions.

All constructors of the `context` class construct an object that is associated with a particular [SYCL backend](#), determined by the constructor parameters or, in the case of the default constructor, the SYCL `device` produced by the `default_selector_v`.

A context can optionally be constructed with an `async_handler` parameter. In this case the `async_handler` is used to report asynchronous exceptions, as described in [Section 4.13](#).

The `context` class provides the common reference semantics as defined in [Section 4.5.2](#).

```
namespace sycl {
class context {
public:
    explicit context(const property_list& propList = {});

    explicit context(async_handler asyncHandler,
                    const property_list& propList = {});

    explicit context(const device& dev, const property_list& propList = {});

    explicit context(const device& dev, async_handler asyncHandler,
                    const property_list& propList = {});

    explicit context(const std::vector<device>& deviceList,
                    const property_list& propList = {});

    explicit context(const std::vector<device>& deviceList,
                    async_handler asyncHandler,
                    const property_list& propList = {});

    /* -- property interface members -- */

    /* -- common interface members -- */

    backend get_backend() const noexcept;

    platform get_platform() const;

    std::vector<device> get_devices() const;

    template <typename Param> typename Param::return_type get_info() const;

    template <typename Param>
    typename Param::return_type get_backend_info() const;
};
} // namespace sycl
```



### 4.6.3.1. Constructors

#### Default constructor

```
explicit context(async_handler asyncHandler = {})
```

*Effects:* Constructs a `context` object using the `default_selector_v` to determine the associated platform and devices. The associated platform is the platform which contains the device selected by `default_selector_v`. The `devices` that are associated with the constructed context are implementation-defined but must contain the device that is selected by `default_selector_v`. The constructed context uses the `asyncHandler` parameter to handle exceptions.

#### Construct from device

```
explicit context(const device& dev, async_handler asyncHandler = {})
```

*Effects:* Constructs a `context` object using the `dev` parameter to determine the associated platform and device. The associated platform is the platform that contains `dev`, and the associated device is `dev`. The constructed context uses the `asyncHandler` parameter to handle exceptions.

#### Construct from device list

```
explicit context(const std::vector<device>& deviceList,  
                async_handler asyncHandler = {})
```

*Preconditions:* All devices in `deviceList` must be associated with the same platform.

*Effects:* Constructs a `context` object using the `deviceList` parameter to determine the associated platform and device. The associated platform is the platform that contains all of the devices in `deviceList`. The associated devices are those devices in `deviceList`. The constructed context uses the `asyncHandler` parameter to handle exceptions.

### 4.6.3.2. Member functions

#### `context::get_backend`

```
backend get_backend() const noexcept
```

*Returns:* The SYCL backend that is associated with this context.

#### `context::get_platform`

```
platform get_platform() const
```

*Returns:* The platform that is associated with this context.

**context::get\_devices**

```
std::vector<device> get_devices() const
```

*Returns:* A `std::vector` containing all the `devices` that are associated with this context.

**context::get\_info**

```
template <typename Param>
typename Param::return_type get_info() const
```

*Constraints:* The `Param` must be an information descriptor for the context class.

Each information descriptor specifies the return value and may also specify preconditions, exceptions that are thrown, etc. See [Section 4.6.3.3](#) for the context information descriptors that are defined by the core SYCL specification.

**context::get\_backend\_info**

```
template <typename Param>
typename Param::return_type get_backend_info() const
```

*Constraints:* The `Param` must be a backend information descriptor for the context class.

*Throws:* An `exception` with the `errc::backend_mismatch` error code if the backend that corresponds with `Param` is different from the backend that is associated with this context.

Each information descriptor specifies the return value and may also specify preconditions, additional exceptions that are thrown, etc.

**4.6.3.3. Information descriptors**

This section describes the information descriptors that can be used as the `Param` template parameter to `context::get_info`. When the description has a *Returns*, *Throws*, etc. paragraph, this indicates the value returned by or the exceptions thrown by the `context::get_info` function.

**info::context::platform**

```
namespace sycl::info::context {
    struct platform {
        using return_type = platform;
    };
} // namespace sycl::info::context
```

*Remarks:* Template parameter to `context::get_info`.

*Returns:* The `platform` that is associated with this context.

**info::context::devices**

```
namespace sycl::info::context {
  struct devices {
    using return_type = std::vector<device>;
  };
} // namespace sycl::info::context
```

*Remarks:* Template parameter to `context::get_info`.

*Returns:* A `std::vector` containing all the `devices` that are associated with this context.

**info::context::atomic\_memory\_order\_capabilities**

```
namespace sycl::info::context {
  struct atomic_memory_order_capabilities {
    using return_type = std::vector<memory_order>;
  };
} // namespace sycl::info::context
```

*Remarks:* Template parameter to `context::get_info`.

*Returns:* This query applies only to the capabilities of atomic operations that are applied to memory that can be concurrently accessed by multiple devices in the context. If these capabilities are not uniform across all devices in the context, the query reports only the capabilities that are common for all devices.

Returns the set of memory orders supported by these atomic operations. When a context returns a "stronger" memory order in this set, it must also return all "weaker" memory orders. (See [Section 3.8.3.1](#) for a definition of "stronger" and "weaker" memory orders.) The memory orders `memory_order::acquire`, `memory_order::release`, and `memory_order::acq_rel` are all the same strength. If a context returns one of these, it must return them all.

At a minimum, each context must support `memory_order::relaxed`.

**info::context::atomic\_fence\_order\_capabilities**

```
namespace sycl::info::context {
  struct atomic_fence_order_capabilities {
    using return_type = std::vector<memory_order>;
  };
} // namespace sycl::info::context
```

*Remarks:* Template parameter to `context::get_info`.

*Returns:* This query applies only to the capabilities of `atomic_fence` when applied to memory that can be concurrently accessed by multiple devices in the context. If these capabilities are not uniform across all devices in the context, the query reports only the capabilities that are common for all devices.

Returns the set of memory orders supported by these `atomic_fence` operations. When a context returns a "stronger" memory order in this set, it must also return all "weaker" memory orders. (See [Section 3.8.3.1](#) for a definition of "stronger" and "weaker" memory orders.)

At a minimum, each context must support `memory_order::relaxed`, `memory_order::acquire`, `memory_order::release`, and `memory_order::acq_rel`.

**info::context::atomic\_memory\_scope\_capabilities**

```
namespace sycl::info::context {
  struct atomic_memory_scope_capabilities {
    using return_type = std::vector<memory_scope>;
  };
} // namespace sycl::info::context
```

*Remarks:* Template parameter to `context::get_info`.

*Returns:* The set of memory scopes supported by atomic operations on all devices in the context. When a context returns a "wider" memory scope in this set, it must also return all "narrower" memory scopes. (See [Section 3.8.3.2](#) for a definition of "wider" and "narrower" scopes.) At a minimum, each context must support `memory_scope::work_item`, `memory_scope::sub_group`, and `memory_scope::work_group`.

**info::context::atomic\_fence\_scope\_capabilities**

```
namespace sycl::info::context {
  struct atomic_fence_scope_capabilities {
    using return_type = std::vector<memory_scope>;
  };
} // namespace sycl::info::context
```

*Remarks:* Template parameter to `context::get_info`.

*Returns:* The set of memory orderings supported by `atomic_fence` on all devices in the context. When a context returns a "wider" memory scope in this set, it must also return all "narrower" memory scopes. (See [Section 3.8.3.2](#) for a definition of "wider" and "narrower" scopes.) At a minimum, each context must support `memory_scope::work_item`, `memory_scope::sub_group`, and `memory_scope::work_group`.

**4.6.3.4. Properties**

The `property_list` constructor parameters are present for extensibility.

**4.6.4. Device class**

The `device` class represents a single SYCL device on which [kernels](#) can be executed.

All member functions of the `device` class are synchronous and errors are handled by throwing synchronous SYCL exceptions.

The execution environment for a SYCL application has a fixed number of [root devices](#) which does not vary as the application executes. The application can get a list of all these devices via `device::get_devices`, and the order of the device objects is the same each time the application calls that function (assuming the parameter to that function is the same for each call). The `device` class also provides constructors, but constructing a new `device` instance merely creates a new object that is a copy of one of the objects returned by `device::get_devices`.

A device can be partitioned into multiple devices, by calling the `device::create_sub_devices` member function template. The resulting `device` objects are considered sub devices, and it is valid to partition these sub devices further. The range of support for this feature is [SYCL backend](#) and device specific and can be queried for through `device::get_info`.

The `device` class provides the common reference semantics as defined in [Section 4.5.2](#).

```
namespace sycl {

class device {
public:
    device();

    template <typename DeviceSelector>
    explicit device(const DeviceSelector& deviceSelector);

    /* -- common interface members -- */

    backend get_backend() const noexcept;

    bool is_cpu() const;

    bool is_gpu() const;

    bool is_accelerator() const;

    platform get_platform() const;

    template <typename Param> typename Param::return_type get_info() const;

    template <typename Param>
    typename Param::return_type get_backend_info() const;

    bool has(aspect asp) const;

    bool has_extension(const std::string& extension) const; // Deprecated

    // Available only when Prop == info::partition_property::partition_equally
    template <info::partition_property Prop>
    std::vector<device> create_sub_devices(size_t count) const;

    // Available only when Prop == info::partition_property::partition_by_counts
    template <info::partition_property Prop>
    std::vector<device>
    create_sub_devices(const std::vector<size_t>& counts) const;

    // Available only when Prop ==
    // info::partition_property::partition_by_affinity_domain
    template <info::partition_property Prop>
    std::vector<device>
    create_sub_devices(info::partition_affinity_domain affinityDomain) const;

    static std::vector<device>
    get_devices(info::device_type type = info::device_type::all);
};
} // namespace sycl
```

#### 4.6.4.1. Constructors

**Default constructor**

```
device()
```

*Effects:* Constructs a `device` object that is a copy of the device returned by `default_selector_v`.

**Selector constructor**

```
template <typename DeviceSelector>
explicit device(const DeviceSelector& selector)
```

*Constraints:* Available only when the `DeviceSelector` is a type that satisfies the requirements of a `device selector` as defined in [Section 4.6.1.1](#).

*Effects:* The `selector` is called for every `root device` as described in [Section 4.6.1.1](#). Constructs a `device` object that is a copy of the device selected by `selector`.

**4.6.4.2. Member functions****device::get\_backend**

```
backend get_backend() const noexcept
```

*Returns:* The `SYCL backend` that is associated with this device.

**device::get\_platform**

```
platform get_platform() const
```

*Returns:* The `platform` that is associated with this device.

**device::is\_cpu**

```
bool is_cpu() const
```

*Returns:* The same value as `has(aspect::cpu)`. See [Section 4.6.4.5](#).

**device::is\_gpu**

```
bool is_gpu() const
```

*Returns:* The same value as `has(aspect::gpu)`. See [Section 4.6.4.5](#).

**device::is\_accelerator**

```
bool is_accelerator() const
```

*Returns:* The same value as `has(aspect::accelerator)`. See [Section 4.6.4.5](#).

**device::get\_info**

```
template <typename Param>
typename Param::return_type get_info() const
```

*Constraints:* Available only when `Param` is an information descriptor for the device class.

Each information descriptor specifies the return value and may also specify preconditions, exceptions that are thrown, etc. See [Section 4.6.4.4](#) for the device information descriptors that are defined by the [core SYCL specification](#).

**device::get\_backend\_info**

```
template <typename Param>
typename Param::return_type get_backend_info() const
```

*Constraints:* Available only when `Param` is a backend information descriptor for the device class.

*Throws:* An `exception` with the `errc::backend_mismatch` error code if the backend that corresponds with `Param` is different from the backend that is associated with this device.

Each information descriptor specifies the return value and may also specify preconditions, additional exceptions that are thrown, etc.

**device::has**

```
bool has(aspect asp) const
```

*Returns:* The value `true` if this device has the given `aspect`. Applications can use this member function to determine which optional features this device supports (if any).

**device::has\_extension**

```
bool has_extension(const std::string& extension) const
```

Deprecated by SYCL 2020.

[*Note:* Use `device::has` instead. — *end note*]

*Returns:* The value `true` if this device supports the extension queried by the `extension` parameter.



**device::create\_sub\_devices (partition equally)**

```
template <info::partition_property Prop>
std::vector<device> create_sub_devices(size_t count) const
```

*Constraints:* Available only when `Prop` is `info::partition_property::partition_equally`.

*Returns:* A `std::vector` of sub devices partitioned from this `device` object based on the `count` parameter. The returned vector contains as many sub devices as can be created such that each sub device contains `count` compute units. If the device's total number of compute units (as returned by `info::device::max_compute_units`) is not evenly divided by `count`, then the remaining compute units are not included in any of the sub devices.

*Throws:*

- An `exception` with the `errc::feature_not_supported` error code if this device does not support `info::partition_property::partition_equally`.
- An `exception` with the `errc::invalid` error code if `count` exceeds the total number of compute units in the device.

**device::create\_sub\_devices (partition by counts)**

```
template <info::partition_property Prop>
std::vector<device> create_sub_devices(const std::vector<size_t>& counts) const
```

*Constraints:* Available only when `Prop` is `info::partition_property::partition_by_counts`.

*Returns:* A `std::vector` of sub devices partitioned from this `device` object based on the `counts` parameter. For each non-zero value  $M$  in the `counts` vector, a sub device with  $M$  compute units is created.

*Throws:*

- An `exception` with the `errc::feature_not_supported` error code if this device does not support `info::partition_property::partition_by_counts`.
- An `exception` with the `errc::invalid` error code if the number of non-zero values in `counts` exceeds the device's maximum number of sub devices (as returned by `info::device::partition_max_sub_devices`) or if the total of all the values in the `counts` vector exceeds the total number of compute units in the device (as returned by `info::device::max_compute_units`).

**device::create\_sub\_devices (partition by affinity domain)**

```
template <info::partition_property Prop>
std::vector<device>
create_sub_devices(info::partition_affinity_domain domain) const
```

*Constraints:* Available only when `Prop` is `info::partition_property::partition_by_affinity_domain`.

*Returns:* A `std::vector` of sub devices partitioned from this `device` object based on the `domain` parameter, which must be one of the following values:

- `info::partition_affinity_domain::numa`: Split the device into sub devices comprised of compute units that share a NUMA node.
- `info::partition_affinity_domain::L4_cache`: Split the device into sub devices comprised of compute

units that share a level 4 data cache.

- `info::partition_affinity_domain::L3_cache`: Split the device into sub devices comprised of compute units that share a level 3 data cache.
- `info::partition_affinity_domain::L2_cache`: Split the device into sub devices comprised of compute units that share a level 2 data cache.
- `info::partition_affinity_domain::L1_cache`: Split the device into sub devices comprised of compute units that share a level 1 data cache.
- `info::partition_affinity_domain::next_partitionable`: Split the device along the next partitionable affinity domain. The implementation shall find the first level along which the device or sub device may be further subdivided in the order `numa`, `L4_cache`, `L3_cache`, `L2_cache`, `L1_cache`, and partition the device into sub devices comprised of compute units that share memory subsystems at this level. The user may determine what happened via `info::device::partition_type_affinity_domain`.

*Throws:*

- An `exception` with the `errc::feature_not_supported` error code if this device does not support `info::partition_property::partition_by_affinity_domain` or if this device does not support the `info::partition_affinity_domain` provided.

#### 4.6.4.3. Static member functions

`device::get_devices`

```
static std::vector<device>
get_devices(info::device_type type = info::device_type::all)
```

*Returns:* A `std::vector` containing all the `root` devices from all `SYCL backends` available in the system which have the device type `type`.

#### 4.6.4.4. Information descriptors

This section describes the information descriptors that can be used as the `Param` template parameter to `device::get_info`. When the description has a *Returns*, *Throws*, etc. paragraph, this indicates the value returned by or the exceptions thrown by the `device::get_info` function.

`info::device::device_type`

```
namespace sycl::info::device {
    struct device_type {
        using return_type = info::device_type;
    };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The device type associated with the device. May not return `info::device_type::all`.

**info::device::vendor\_id**

```
namespace sycl::info::device {
  struct vendor_id {
    using return_type = uint32_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* A unique vendor device identifier.

**info::device::max\_compute\_units**

```
namespace sycl::info::device {
  struct max_compute_units {
    using return_type = uint32_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The number of parallel compute units available to the device. The minimum value is 1.

**info::device::max\_work\_item\_dimensions**

```
namespace sycl::info::device {
  struct max_work_item_dimensions {
    using return_type = uint32_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The maximum dimensions that specify the global and local work-item IDs used by the data parallel execution model. The minimum value is 3 if this device is not of device type `info::device_type::custom`.

**info::device::max\_work\_item\_sizes**

```
namespace sycl::info::device {
  template<int Dimensions = 3>
  struct max_work_item_sizes<Dimensions> {
    using return_type = range<Dimensions>;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Constraints:* Available only when `Dimensions` is 1, 2, or 3.

*Returns:* The maximum number of work-items that are permitted in a work-group for a kernel running

in an index space of `Dimensions` dimensions. When the device type is not `info::device_type::custom`, the minimum value returned from this query is: (1) when `Dimensions` is 1, (1, 1) when `Dimensions` is 2, and (1, 1, 1) when `Dimensions` is 3.

#### `info::device::max_work_group_size`

```
namespace sycl::info::device {
  struct max_work_group_size {
    using return_type = size_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The maximum number of work-items that this device is capable of executing in a work-group. The minimum value is 1. This value is an upper limit and will not necessarily maximize performance. The maximum number of work-items in a work-group depends on the kernel and the implementation. Use `info::kernel_device_specific::work_group_size` to query this limit.

#### `info::device::max_num_sub_groups`

```
namespace sycl::info::device {
  struct max_num_sub_groups {
    using return_type = uint32_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The maximum number of sub-groups in a work-group for any kernel executed on the device. The minimum value is 1.

#### `info::device::sub_group_sizes`

```
namespace sycl::info::device {
  struct sub_group_sizes {
    using return_type = std::vector<size_t>;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* A `std::vector` of `size_t` containing the set of sub-group sizes supported by the device.

#### `info::device::preferred_vector_width`

```
namespace sycl::info::device {
  struct preferred_vector_width_char {
    using return_type = uint32_t;
  };
}
```

```

struct preferred_vector_width_short {
    using return_type = uint32_t;
};
struct preferred_vector_width_int {
    using return_type = uint32_t;
};
struct preferred_vector_width_long {
    using return_type = uint32_t;
};
struct preferred_vector_width_float {
    using return_type = uint32_t;
};
struct preferred_vector_width_double {
    using return_type = uint32_t;
};
struct preferred_vector_width_half {
    using return_type = uint32_t;
};
} // namespace sycl::info::device

```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The preferred native vector width size for built-in scalar types that can be put into vectors. The vector width is defined as the number of scalar elements that can be stored in the vector. Must return 0 for `info::device::preferred_vector_width_double` if the device does not have `aspect::fp64` and must return 0 for `info::device::preferred_vector_width_half` if the device does not have `aspect::fp16`.

#### `info::device::native_vector_width`

```

namespace sycl::info::device {
struct native_vector_width_char {
    using return_type = uint32_t;
};
struct native_vector_width_short {
    using return_type = uint32_t;
};
struct native_vector_width_int {
    using return_type = uint32_t;
};
struct native_vector_width_long {
    using return_type = uint32_t;
};
struct native_vector_width_float {
    using return_type = uint32_t;
};
struct native_vector_width_double {
    using return_type = uint32_t;
};
struct native_vector_width_half {
    using return_type = uint32_t;
};
} // namespace sycl::info::device

```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The native ISA vector width. The vector width is defined as the number of scalar elements that can be stored in the vector. Must return 0 for `info::device::native_vector_width_double` if the device does not have `aspect::fp64` and must return 0 for `info::device::native_vector_width_half` if the device does not have `aspect::fp16`.

#### `info::device::max_clock_frequency`

```
namespace sycl::info::device {
  struct max_clock_frequency {
    using return_type = uint32_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The maximum configured clock frequency of this device in MHz.

#### `info::device::address_bits`

```
namespace sycl::info::device {
  struct address_bits {
    using return_type = uint32_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The default compute device address space size in bits. Must return either 32 or 64.

#### `info::device::max_mem_alloc_size`

```
namespace sycl::info::device {
  struct max_mem_alloc_size {
    using return_type = uint64_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The maximum size of memory object allocation in bytes.

#### `info::device::image_support`

```
namespace sycl::info::device {
  struct image_support {
    using return_type = bool;
  };
} // namespace sycl::info::device
```

Deprecated by SYCL 2020.

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The same value as `device::has(aspect::image)`.

**info::device::max\_read\_image\_args**

```
namespace sycl::info::device {
  struct max_read_image_args {
    using return_type = uint32_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The maximum number of simultaneous image objects that can be read from by a kernel. The minimum value is 128 if the device has `aspect::image`.

**info::device::max\_write\_image\_args**

```
namespace sycl::info::device {
  struct max_write_image_args {
    using return_type = uint32_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The maximum number of simultaneous image objects that can be written to by a kernel. The minimum value is 8 if the device has `aspect::image`.

**info::device::image2d\_max\_width**

```
namespace sycl::info::device {
  struct image2d_max_width {
    using return_type = size_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The maximum width of a 2D image or 1D image in pixels. The minimum value is 8192 if the device has `aspect::image`.

**info::device::image2d\_max\_height**

```
namespace sycl::info::device {
  struct image2d_max_height {
    using return_type = size_t;
  };
} // namespace sycl::info::device
```



*Remarks:* Template parameter to `device::get_info`.

*Returns:* The maximum height of a 2D image in pixels. The minimum value is 8192 if the device has `aspect::image`.

`info::device::image3d_max_width`

```
namespace sycl::info::device {
  struct image3d_max_width {
    using return_type = size_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The maximum width of a 3D image in pixels. The minimum value is 2048 if the device has `aspect::image`.

`info::device::image3d_max_height`

```
namespace sycl::info::device {
  struct image3d_max_height {
    using return_type = size_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The maximum height of a 3D image in pixels. The minimum value is 2048 if the device has `aspect::image`.

`info::device::image3d_max_depth`

```
namespace sycl::info::device {
  struct image3d_max_depth {
    using return_type = size_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The maximum depth of a 3D image in pixels. The minimum value is 2048 if the device has `aspect::image`.

`info::device::image_max_buffer_size`

```
namespace sycl::info::device {
  struct image_max_buffer_size {
    using return_type = size_t;
  };
}
```

---

```
} // namespace sycl::info::device
```

---

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The number of pixels for a 1D image created from a buffer object. The minimum value is 65536 if the device has `aspect::image`. Note that this information is intended for OpenCL interoperability only as this feature is not supported in SYCL.

---

#### `info::device::max_samplers`

```
namespace sycl::info::device {
  struct max_samplers {
    using return_type = uint32_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The maximum number of samplers that can be used in a kernel. The minimum value is 16 if the device has `aspect::image`.

---

#### `info::device::max_parameter_size`

```
namespace sycl::info::device {
  struct max_parameter_size {
    using return_type = size_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The maximum size in bytes of the arguments that can be passed to a kernel. The minimum value is 1024 if this device is not of device type `info::device_type::custom`. For this minimum value, only a maximum of 128 arguments can be passed to a kernel.

---

#### `info::device::mem_base_addr_align`

```
namespace sycl::info::device {
  struct mem_base_addr_align {
    using return_type = uint32_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The minimum value in bits of the largest supported SYCL built-in data type if this device is not of device type `info::device_type::custom`.

---

**info::device::half\_fp\_config**

```
namespace sycl::info::device {
  struct half_fp_config {
    using return_type = std::vector<info::fp_config>;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* A `std::vector` of `info::fp_config` values describing the half precision floating-point capability of this device. The `std::vector` may contain zero or more of the following values:

- `info::fp_config::denorm`
- `info::fp_config::inf_nan`
- `info::fp_config::round_to_nearest`
- `info::fp_config::round_to_zero`
- `info::fp_config::round_to_inf`
- `info::fp_config::fma`
- `info::fp_config::correctly_rounded_divide_sqrt`
- `info::fp_config::soft_float`

If half precision is supported by this device (i.e. the device has `aspect::fp16`) there is no minimum floating-point capability. If half support is not supported the returned `std::vector` must be empty.

**info::device::single\_fp\_config**

```
namespace sycl::info::device {
  struct single_fp_config {
    using return_type = std::vector<info::fp_config>;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* A `std::vector` of `info::fp_config` values describing the single precision floating-point capability of this device. The `std::vector` must contain one or more of the following values:

- `info::fp_config::denorm`
- `info::fp_config::inf_nan`
- `info::fp_config::round_to_nearest`
- `info::fp_config::round_to_zero`
- `info::fp_config::round_to_inf`
- `info::fp_config::fma`
- `info::fp_config::correctly_rounded_divide_sqrt`
- `info::fp_config::soft_float`

If this device is not of type `info::device_type::custom` then the minimum floating-point capability must be: `info::fp_config::round_to_nearest` and `info::fp_config::inf_nan`.

**info::device::double\_fp\_config**

```
namespace sycl::info::device {
  struct double_fp_config {
    using return_type = std::vector<info::fp_config>;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* A `std::vector` of `info::fp_config` values describing the double precision floating-point capability of this device. The `std::vector` may contain zero or more of the following values:

- `info::fp_config::denorm`
- `info::fp_config::inf_nan`
- `info::fp_config::round_to_nearest`
- `info::fp_config::round_to_zero`
- `info::fp_config::round_to_inf`
- `info::fp_config::fma`
- `info::fp_config::soft_float`

If double precision is supported by this device (i.e. the device has `aspect::fp64`) and this device is not of type `info::device_type::custom` then the minimum floating-point capability must be: `info::fp_config::fma`, `info::fp_config::round_to_nearest`, `info::fp_config::round_to_zero`, `info::fp_config::round_to_inf`, `info::fp_config::inf_nan` and `info::fp_config::denorm`. If double support is not supported the returned `std::vector` must be empty.

**info::device::global\_mem\_cache\_type**

```
namespace sycl::info::device {
  struct global_mem_cache_type {
    using return_type = info::global_mem_cache_type;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The type of global memory cache supported.

**info::device::global\_mem\_cache\_line\_size**

```
namespace sycl::info::device {
  struct global_mem_cache_line_size {
    using return_type = uint32_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The size of global memory cache line in bytes.

**info::device::global\_mem\_cache\_size**

```
namespace sycl::info::device {
  struct global_mem_cache_size {
    using return_type = uint64_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The size of global memory cache in bytes.

**info::device::global\_mem\_size**

```
namespace sycl::info::device {
  struct global_mem_size {
    using return_type = uint64_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The size of global device memory in bytes.

**info::device::max\_constant\_buffer\_size**

```
namespace sycl::info::device {
  struct max_constant_buffer_size {
    using return_type = uint64_t;
  };
} // namespace sycl::info::device
```

Deprecated by SYCL 2020.

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The maximum size in bytes of a constant buffer allocation. The minimum value is 64 KB if this device is not of type `info::device_type::custom`.

**info::device::max\_constant\_args**

```
namespace sycl::info::device {
  struct max_constant_args {
    using return_type = uint32_t;
  };
} // namespace sycl::info::device
```

Deprecated by SYCL 2020.

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The maximum number of constant arguments that can be declared in a kernel. The minimum value is 8 if this device is not of type `info::device_type::custom`.

#### `info::device::local_mem_type`

```
namespace sycl::info::device {
  struct local_mem_type {
    using return_type = info::local_mem_type;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The type of local memory supported. This can be `info::local_mem_type::local` implying dedicated local memory storage such as SRAM, or `info::local_mem_type::global`. If this device is of type `info::device_type::custom` this can also be `info::local_mem_type::none`, indicating local memory is not supported.

#### `info::device::local_mem_size`

```
namespace sycl::info::device {
  struct local_mem_size {
    using return_type = uint64_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The size of local memory arena in bytes. The minimum value is 32 KB if this device is not of type `info::device_type::custom`.

#### `info::device::error_correction_support`

```
namespace sycl::info::device {
  struct error_correction_support {
    using return_type = bool;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The value `true` if the device implements error correction for all accesses to compute device memory (global and constant). Returns false if the device does not implement such error correction.

#### `info::device::host_unified_memory`

```
namespace sycl::info::device {
  struct host_unified_memory {
    using return_type = bool;
  };
}
```

```
} // namespace sycl::info::device
```

Deprecated by SYCL 2020.

[*Note:* Use `device::has` with one of the `aspect::usm_*` aspects instead. — *end note*]

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The value true if the device and the host have a unified memory subsystem and returns `false` otherwise.

`info::device::atomic_memory_order_capabilities`

```
namespace sycl::info::device {
    struct atomic_memory_order_capabilities {
        using return_type = std::vector<memory_order>;
    };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The set of memory orders supported by atomic operations on this device. When a device returns a "stronger" memory order in this set, it must also return all "weaker" memory orders. (See [Section 3.8.3.1](#) for a definition of "stronger" and "weaker" memory orders.) The memory orders `memory_order::acquire`, `memory_order::release`, and `memory_order::acq_rel` are all the same strength. If a device returns one of these, it must return them all.

At a minimum, each device must support `memory_order::relaxed`.

`info::device::atomic_fence_order_capabilities`

```
namespace sycl::info::device {
    struct atomic_fence_order_capabilities {
        using return_type = std::vector<memory_order>;
    };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The set of memory orders supported by `atomic_fence` on this device. When a device returns a "stronger" memory order in this set, it must also return all "weaker" memory orders. (See [Section 3.8.3.1](#) for a definition of "stronger" and "weaker" memory orders.) At a minimum, each device must support `memory_order::relaxed`, `memory_order::acquire`, `memory_order::release`, and `memory_order::acq_rel`.

`info::device::atomic_memory_scope_capabilities`

```
namespace sycl::info::device {
    struct atomic_memory_scope_capabilities {
        using return_type = std::vector<memory_scope>;
    };
} // namespace sycl::info::device
```



*Remarks:* Template parameter to `device::get_info`.

*Returns:* The set of memory scopes supported by atomic operations on this device. When a device returns a "wider" memory scope in this set, it must also return all "narrower" memory scopes. (See [Section 3.8.3.2](#) for a definition of "wider" and "narrower" scopes.) At a minimum, each device must support `memory_scope::work_item`, `memory_scope::sub_group`, and `memory_scope::work_group`.

**info::device::atomic\_fence\_scope\_capabilities**

```
namespace sycl::info::device {
  struct atomic_fence_scope_capabilities {
    using return_type = std::vector<memory_scope>;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The set of memory scopes supported by `atomic_fence` on this device. When a device returns a "wider" memory scope in this set, it must also return all "narrower" memory scopes. (See [Section 3.8.3.2](#) for a definition of "wider" and "narrower" scopes.) At a minimum, each device must support `memory_scope::work_item`, `memory_scope::sub_group`, and `memory_scope::work_group`.

**info::device::profiling\_timer\_resolution**

```
namespace sycl::info::device {
  struct profiling_timer_resolution {
    using return_type = size_t;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The resolution of device timer in nanoseconds.

**info::device::is\_endian\_little**

```
namespace sycl::info::device {
  struct is_endian_little {
    using return_type = bool;
  };
} // namespace sycl::info::device
```

Deprecated by SYCL 2020.

[*Note:* Check the byte order of the host system instead. The host and device are required to have the same byte order. — *end note*]

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The value `true` if this device is a little endian device and returns `false` otherwise.

**info::device::is\_available**

```
namespace sycl::info::device {
  struct is_available {
    using return_type = bool;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The value `true` if the device is available and `false` if the device is not available. A device is considered to be available if the device can be expected to successfully execute commands enqueued to the device. The conditions that lead to a device being considered available or not available are implementation-defined.

**info::device::is\_compiler\_available**

```
namespace sycl::info::device {
  struct is_compiler_available {
    using return_type = bool;
  };
} // namespace sycl::info::device
```

Deprecated by SYCL 2020.

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The same value as `device::has(aspect::online_compiler)`.

**info::device::is\_linker\_available**

```
namespace sycl::info::device {
  struct is_linker_available {
    using return_type = bool;
  };
} // namespace sycl::info::device
```

Deprecated by SYCL 2020.

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The same value as `device::has(aspect::online_linker)`.

**info::device::execution\_capabilities**

```
namespace sycl::info::device {
  struct execution_capabilities {
    using return_type = std::vector<info::execution_capability>;
  };
} // namespace sycl::info::device
```

Deprecated by SYCL 2020.

*Remarks:* Template parameter to `device::get_info`.

*Returns:* Only supported when the backend of this device is OpenCL (see [Appendix C](#)). Returns a `std::vector` of the `info::execution_capability` values describing the supported execution capabilities. Unless the device type is `info::device_type::custom`, the returned vector will always include `info::execution_capability::exec_kernel`.

*Throws:* An `exception` with the `errc::invalid` error code if the backend of this device is not OpenCL.

#### `info::device::queue_profiling`

```
namespace sycl::info::device {
  struct queue_profiling {
    using return_type =
  };
} // namespace sycl::info::device
```

Deprecated by SYCL 2020.

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The same value as `device::has(aspect::queue_profiling)`.

#### `info::device::built_in_kernel_ids`

```
namespace sycl::info::device {
  struct built_in_kernel_ids {
    using return_type = std::vector<kernel_id>;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* A `std::vector` of identifiers for the built-in kernels supported by this device.

#### `info::device::built_in_kernels`

```
namespace sycl::info::device {
  struct built_in_kernels {
    using return_type = std::vector<std::string>;
  };
} // namespace sycl::info::device
```

Deprecated by SYCL 2020.

[Note: Use `info::device::built_in_kernel_ids` instead. — *end note*]

*Remarks:* Template parameter to `device::get_info`.

*Returns:* A `std::vector` of built-in OpenCL kernels supported by this device.

**info::device::platform**

```
namespace sycl::info::device {  
  struct platform {  
    using return_type = platform;  
  };  
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The `platform` that is associated with this device.

**info::device::name**

```
namespace sycl::info::device {  
  struct name {  
    using return_type = std::string;  
  };  
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* An implementation-defined name for this device.

**info::device::vendor**

```
namespace sycl::info::device {  
  struct vendor {  
    using return_type = std::string;  
  };  
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* An implementation-defined name for the vendor providing this device.

**info::device::driver\_version**

```
namespace sycl::info::device {  
  struct driver_version {  
    using return_type = std::string;  
  };  
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* An implementation-defined name describing the version of the underlying software driver for this device.

**info::device::profile**

```
namespace sycl::info::device {
  struct profile {
    using return_type = std::string;
  };
} // namespace sycl::info::device
```

Deprecated by SYCL 2020.

*Remarks:* Template parameter to `device::get_info`.

*Returns:* Only supported when the backend of this device is OpenCL (see [Appendix C](#)). The value returned can be one of the following strings:

- FULL\_PROFILE - if the device supports the OpenCL specification (functionality defined as part of the core specification and does not require any extensions to be supported).
- EMBEDDED\_PROFILE - if the device supports the OpenCL embedded profile.

*Throws:* An `exception` with the `errc::invalid` error code if the backend of this device is not OpenCL.

**info::device::version**

```
namespace sycl::info::device {
  struct version {
    using return_type = std::string;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* A backend-defined device version.

**info::device::backend\_version**

```
namespace sycl::info::device {
  struct backend_version {
    using return_type = std::string;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* A string describing the version of the [SYCL backend](#) associated with this device. The value returned from this query is defined by the backend interoperation specification that corresponds to this device's backend.

**info::device::aspects**

```
namespace sycl::info::device {
  struct aspects {
    using return_type = std::vector<aspect>;
  };
}
```

```
};
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* A `std::vector` of `aspect` values supported by this device.

#### `info::device::extensions`

```
namespace sycl::info::device {
  struct extensions {
    using return_type = std::vector<std::string>;
  };
} // namespace sycl::info::device
```

Deprecated by SYCL 2020.

[*Note:* Use `info::device::aspects` instead. — *end note*]

*Remarks:* Template parameter to `device::get_info`.

*Returns:* A `std::vector` of extension names (the extension names do not contain any spaces) supported by this device. The extension names returned can be vendor supported extension names and one or more of the following Khronos approved extension names:

- `cl_khr_int64_base_atomics`
- `cl_khr_int64_extended_atomics`
- `cl_khr_3d_image_writes`
- `cl_khr_fp16`
- `cl_khr_gl_sharing`
- `cl_khr_gl_event`
- `cl_khr_d3d10_sharing`
- `cl_khr_dx9_media_sharing`
- `cl_khr_d3d11_sharing`
- `cl_khr_depth_images`
- `cl_khr_gl_depth_images`
- `cl_khr_gl_msaa_sharing`
- `cl_khr_image2d_from_buffer`
- `cl_khr_initialize_memory`
- `cl_khr_context_abort`
- `cl_khr_spir`

If the backend associated with this device is OpenCL, then following approved Khronos extension names must be returned by all device that support OpenCL C 1.2:

- `cl_khr_global_int32_base_atomics`
- `cl_khr_global_int32_extended_atomics`
- `cl_khr_local_int32_base_atomics`

- `cl_khr_local_int32_extended_atomics`
- `cl_khr_byte_addressable_store`
- `cl_khr_fp64` (for backward compatibility if double precision is supported)

Please refer to the OpenCL 1.2 Extension Specification for a detailed description of these extensions.

#### `info::device::printf_buffer_size`

```
namespace sycl::info::device {
  struct printf_buffer_size {
    using return_type = size_t;
  };
} // namespace sycl::info::device
```

Deprecated by SYCL 2020.

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The maximum size of the internal buffer that holds the output of `printf` calls from a kernel. The minimum value is 1 MB if `info::device::profile` returns true for this device.

#### `info::device::preferred_interop_user_sync`

```
namespace sycl::info::device {
  struct preferred_interop_user_sync {
    using return_type = bool;
  };
} // namespace sycl::info::device
```

Deprecated by SYCL 2020.

*Remarks:* Template parameter to `device::get_info`.

*Returns:* Only supported when the backend of this device is OpenCL (see [Appendix C](#)). Returns `true` if the preference for this device is for the user to be responsible for synchronization, when sharing memory objects between OpenCL and other APIs such as DirectX, `false` if the device/implementation has a performant path for performing synchronization of memory object shared between OpenCL and other APIs such as DirectX.

*Throws:* An `exception` with the `errc::invalid` error code if the backend of this device is not OpenCL.

#### `info::device::parent_device`

```
namespace sycl::info::device {
  info::device::parent_device
  struct parent_device {
    using return_type = device;
  };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.



*Returns:* The parent device to which this sub device is a child if this is a sub device.

*Throws:* An **exception** with the **errc::invalid** error code if this device is not a sub device.

#### **info::device::partition\_max\_sub\_devices**

```
namespace sycl::info::device {
    struct partition_max_sub_devices {
        using return_type = uint32_t;
    };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to **device::get\_info**.

*Returns:* The maximum number of sub devices that can be created when this device is partitioned. The value returned cannot exceed the value returned by **info::device::max\_compute\_units**.

#### **info::device::partition\_properties**

```
namespace sycl::info::device {
    struct partition_properties {
        using return_type = std::vector<info::partition_property>;
    };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to **device::get\_info**.

*Returns:* A **std::vector** of the partition properties supported by this device. An element is returned in this vector only if the device can be partitioned into at least two sub devices along that partition property.

#### **info::device::partition\_affinity\_domains**

```
namespace sycl::info::device {
    struct partition_affinity_domains {
        using return_type = std::vector<info::partition_affinity_domain>;
    };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to **device::get\_info**.

*Returns:* A **std::vector** of the partition affinity domains supported by this device when partitioning with **info::partition\_property::partition\_by\_affinity\_domain**. An element is returned in this vector only if the device can be partitioned into at least two sub devices along that affinity domain.

#### **info::device::partition\_type\_property**

```
namespace sycl::info::device {
    struct partition_type_property {
        using return_type = info::partition_property;
    };
};
```

```
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The partition property of this device. If this device is not a sub device then the return value is `info::partition_property::no_partition`, otherwise it is one of the following values:

- `info::partition_property::partition_equally`
- `info::partition_property::partition_by_counts`
- `info::partition_property::partition_by_affinity_domain`

`info::device::partition_type_affinity_domain`

```
namespace sycl::info::device {
    struct partition_type_affinity_domain {
        using return_type = info::partition_affinity_domain;
    };
} // namespace sycl::info::device
```

*Remarks:* Template parameter to `device::get_info`.

*Returns:* The partition affinity domain of this device. If this device is not a sub device or the sub device was not partitioned with `info::partition_property::partition_by_affinity_domain` then the return value is `info::partition_affinity_domain::not_applicable`, otherwise it is one of the following values:

- `info::partition_affinity_domain::numa`
- `info::partition_affinity_domain::L4_cache`
- `info::partition_affinity_domain::L3_cache`
- `info::partition_affinity_domain::L2_cache`
- `info::partition_affinity_domain::L1_cache`

#### 4.6.4.5. Aspects

Every device has an associated set of aspects which identify characteristics of the device. Aspects are defined via the `aspect` enumeration:

```
namespace sycl {
    enum class aspect : /* unspecified */ {
        cpu,
        gpu,
        accelerator,
        custom,
        emulated,
        host_debuggable,
        fp16,
        fp64,
        atomic64,
        image,
        online_compiler,
        online_linker,
    };
}
```

```

queue_profiling,
usm_device_allocations,
usm_host_allocations,
usm_atomic_host_allocations,
usm_shared_allocations,
usm_atomic_shared_allocations,
usm_system_allocations
};

} // namespace sycl

```

Applications can query the aspects of a device via `device::has` in order to determine whether the device supports any optional features. The following list describes the aspects that are defined in the [core SYCL specification](#) and tells which optional features correspond to each. Backends and extensions may provide additional aspects and additional optional device features. If so, the [SYCL backend specification document](#) or the extension document describes them.

#### **`aspect::cpu`**

Indicates that the implementation identifies this device as a CPU that has device type `info::device_type::cpu`.

[*Note:* A device with this aspect will typically share some or all of the execution resources available to the host C++ application. — *end note*]

#### **`aspect::gpu`**

Indicates that the implementation identifies this device as a GPU that has device type `info::device_type::gpu`.

[*Note:* A device with this aspect may have additional capabilities for accelerating graphics operations, via SYCL image functionality and/or interoperability with graphics APIs. — *end note*]

#### **`aspect::accelerator`**

Indicates that the implementation identifies this device as an accelerator that has device type `info::device_type::accelerator`.

[*Note:* A device with this aspect will typically be a dedicated accelerator device, with a peripheral interconnect for communication. — *end note*]

#### **`aspect::custom`**

Indicates that this device is a custom accelerator that exposes only fixed functionality, and has device type `info::device_type::custom`.

A device with this aspect does not support execution of arbitrary kernels, and can only execute pre-defined kernels (see [Section 3.9.7](#)).

#### **`aspect::emulated`**

Indicates that the device is somehow emulated.

A device with this aspect is not intended for performance, and instead will generally have another pur-

pose such as emulation or profiling. The precise definition of this aspect is left open to the SYCL implementation.

[*Note:* As an example, a vendor might support both a hardware FPGA device and a software emulated FPGA, where the emulated FPGA has all the same features as the hardware one but runs more slowly and can provide additional profiling or diagnostic information. In such a case, an application's [device selector](#) can use `aspect::emulated` to distinguish the two. — *end note*]

---

**aspect::host\_debuggable**

Indicates that [kernels](#) running on this device can be debugged using standard debuggers that are normally available on the host system where the SYCL implementation resides. The precise definition of this aspect is left open to the SYCL implementation.

---

**aspect::fp16**

Indicates that kernels submitted to the device may use the `sycl::half` data type.

---

**aspect::fp64**

Indicates that kernels submitted to the device may use the `double` data type.

---

**aspect::atomic64**

Indicates that kernels submitted to the device may perform 64-bit atomic operations.

---

**aspect::image**

Indicates that the device supports [images](#).

---

**aspect::online\_compiler**

Indicates that the device supports online compilation of device code. Devices that have this aspect support the `build` and `compile` functions defined in [Section 4.11.11](#).

---

**aspect::online\_linker**

Indicates that the device supports online linking of device code. Devices that have this aspect support the `link` functions defined in [Section 4.11.11](#). All devices that have this aspect also have `aspect::online_compiler`.

---

**aspect::queue\_profiling**

Indicates that the device supports queue profiling via `property::queue::enable_profiling`.

---

**aspect::usm\_device\_allocations**

Indicates that the device supports explicit USM allocations as described in [Section 4.8](#).

---

**aspect::usm\_host\_allocations**

Indicates that the device can access USM memory allocated via `usm::alloc::host`. The device only supports atomic modification of a host allocation if `aspect::usm_atomic_host_allocations` is also supported. (See [Section 4.8](#).)

**aspect::usm\_atomic\_host\_allocations**

Indicates that the device supports USM memory allocated via `usm::alloc::host`. The host and this device may concurrently access and atomically modify host allocations. (See [Section 4.8](#).)

**aspect::usm\_shared\_allocations**

Indicates that the device supports USM memory allocated via `usm::alloc::shared` on the same device. Concurrent access and atomic modification of a shared allocation is only supported if `aspect::usm_atomic_shared_allocations` is also supported. (See [Section 4.8](#).)

**aspect::usm\_atomic\_shared\_allocations**

Indicates that the device supports USM memory allocated via `usm::alloc::shared`. The host and other devices in the same context that also support this capability may concurrently access and atomically modify shared allocations. The allocation is free to migrate between the host and the appropriate devices. (See [Section 4.8](#).)

**aspect::usm\_system\_allocations**

Indicates that the system allocator may be used instead of SYCL USM allocation mechanisms for `usm::alloc::shared` allocations on this device. (See [Section 4.8](#).)

**4.6.4.6. Aspect traits**

The implementation also provides two traits that the application can use to query aspects at compilation time. The traits `any_device_has<aspect>` and `all_devices_have<aspect>` are set according to the collection of devices  $D$  that can possibly execute device code, as determined by the compilation environment. The trait `any_device_has<aspect>` inherits from `std::true_type` only if at least one device in  $D$  has the specified aspect. The trait `all_devices_have<aspect>` inherits from `std::true_type` only if all devices in  $D$  have the specified aspect.

```
namespace sycl {

template <aspect Aspect> struct any_device_has;
template <aspect Aspect> struct all_devices_have;

template <aspect A>
inline constexpr bool any_device_has_v = any_device_has<A>::value;
template <aspect A>
inline constexpr bool all_devices_have_v = all_devices_have<A>::value;

} // namespace sycl
```

Applications can use these traits to reduce their code size. The following example demonstrates one way to use these traits to avoid instantiating a templated kernel for device features that are not supported by

any device.

```

1 #include <sycl/sycl.hpp>
2 using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names
3
4 constexpr int N = 512;
5
6 template <bool HasFp16>
7 class MyKernel {
8 public:
9     void operator()(id<1> i) {
10         if constexpr (HasFp16) {
11             // Algorithm using sycl::half type
12         } else {
13             // Fall back code for devices that don't support sycl::half
14         }
15     }
16 };
17
18 int main() {
19     queue myQueue;
20     myQueue.submit([&](handler& cgh) {
21         device dev = myQueue.get_device();
22         if (dev.has(aspect::fp16)) {
23             cgh.parallel_for(range{N}, MyKernel<any_device_has_v<aspect::fp16>>{});
24         } else {
25             cgh.parallel_for(range{N}, MyKernel<all_devices_have_v<aspect::fp16>>{});
26         }
27     });
28
29     myQueue.wait();
30 }

```

The kernel function `MyKernel` is templated to use a different algorithm depending on whether the device has the aspect `aspect::fp16`, and the call to `dev.has()` chooses the kernel function instantiation that matches the device's capabilities. However, the use of `any_device_has_v` and `all_devices_have_v` entirely avoid useless instantiations of the kernel function. For example, when the compilation environment does not support any devices with `aspect::fp16`, `any_device_has_v<aspect::fp16>` is `false`, and the kernel function is never instantiated with support for the `sycl::half` type.

[Note: Like any trait, the definitions of `any_device_has` and `all_devices_have` are uniform across all parts of a SYCL application. If an implementation uses `SMCP`, all compiler passes define a particular aspect's specialization of the traits the same way, regardless of whether that compiler pass' device supports the aspect. Thus, `any_device_has` and `all_devices_have` cannot be used to determine whether any particular device supports an aspect. Instead, applications must use `device::has` or `platform::has` for this. — end note]

[Note: An implementation could choose to provide command line options which affect the set of devices that it supports. If so, those command line options would also affect these traits. For example, if an implementation provides a command line option that disables `aspect::accelerator` devices, the trait `any_device_has<aspect::accelerator>` would inherit from `std::false_type` when that command line option was specified. — end note]

[Note: These traits only reflect the supported devices at the time the SYCL application is compiled. It's possible that unsupported devices are still visible to the application when it runs. However, if a device *D* is not supported when the application is compiled, the application will not be able to submit kernels to

that device *D*. — *end note*]

#### 4.6.4.7. Other enumerations

##### 4.6.4.7.1. Device type

```
namespace sycl::info {
enum class device_type : /* unspecified */ {
    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
};
} // namespace sycl::info
```

##### 4.6.4.7.2. Partition property

```
namespace sycl::info {
enum class partition_property : /* unspecified */ {
    no_partition,
    partition_equally,
    partition_by_counts,
    partition_by_affinity_domain
};
} // namespace sycl::info
```

##### 4.6.4.7.3. Partition affinity domain

```
namespace sycl::info {
enum class partition_affinity_domain : /* unspecified */ {
    not_applicable,
    numa,
    L4_cache,
    L3_cache,
    L2_cache,
    L1_cache,
    next_partitionable
};
} // namespace sycl::info
```

##### 4.6.4.7.4. Floating point configuration

The `info::fp_config` enumeration tells the behavior of floating point operations on a device.

```
namespace sycl::info {
enum class fp_config : /* unspecified */ {
    denorm,
    inf_nan,
    round_to_nearest,
    round_to_zero,
```



```

round_to_inf,
fma,
correctly_rounded_divide_sqrt,
soft_float
};
} // namespace sycl::info

```

**info::fp\_config::denorm**

Denormalized numbers are supported.

**info::fp\_config::inf\_nan**

INF and NaNs are supported.

**info::fp\_config::round\_to\_nearest**

Round to nearest even rounding mode is supported.

**info::fp\_config::round\_to\_zero**

Round to zero rounding mode is supported.

**info::fp\_config::round\_to\_inf**

Round to positive and negative infinity rounding modes are supported.

**info::fp\_config::fma**

IEEE754-2008 fused multiply-add is supported.

**info::fp\_config::correctly\_rounded\_divide\_sqrt**

Deprecated by SYCL 2020.

Divide and sqrt are correctly rounded as defined by the IEEE754 specification.

**info::fp\_config::soft\_float**

Basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software.

**4.6.4.7.5. Local memory type**

```

namespace sycl::info {
enum class local_mem_type : /* unspecified */ {
    none,
    local,
    global
};

```

```
} // namespace sycl::info
```

#### 4.6.4.7.6. Global memory cache type

```
namespace sycl::info {
enum class global_mem_cache_type : /* unspecified */ {
    none,
    read_only,
    read_write
};
} // namespace sycl::info
```

#### 4.6.4.7.7. Execution capability

Deprecated by SYCL 2020.

The `info::execution_capability` enumeration tells the type of kernels that can be submitted to a device from the OpenCL backend.

```
namespace sycl::info {
enum class execution_capability : /* unspecified */ {
    exec_kernel,
    exec_native_kernel
};
} // namespace sycl::info
```

`info::execution_capability::exec_kernel`

Device can execute SYCL kernels.

`info::execution_capability::exec_native_kernel`

Device can execute native OpenCL kernels.

### 4.6.5. Queue class

The `queue` class encapsulates a single SYCL queue which schedules kernels on a device.

A `queue` can be used to submit `command groups` to be executed by the `SYCL runtime` using the `queue::submit` member function.

All member functions of the `queue` class are synchronous and errors are handled by throwing synchronous SYCL exceptions. The `queue::submit` member function synchronously invokes the provided `command group function object` (as described in [Section 3.7.1.2](#)) in the calling thread, thereby scheduling a `command group` for asynchronous execution. Any error in the submission of a `command group` is handled by throwing a synchronous SYCL exception. Any errors from the `command group` after it has been submitted are handled by passing `asynchronous errors` at specific times to an `async_handler`, as described in [Section 4.13](#).

The application can wait for all `command groups` submitted to a queue calling `queue::wait` or `queue::wait_and_throw`.

A queue may be destroyed even when there are uncompleted [commands](#) that have been submitted to the queue. Doing so does not block. Instead, any commands that have been submitted to the queue begin execution when their requisites are satisfied, just as they would had the queue not been destroyed. Any event objects for those commands are signaled in the normal manner when the command completes. Resources associated with the queue are freed by the time the last command completes.

The `queue` class provides the common reference semantics as defined in [Section 4.5.2](#).

```
namespace sycl {
class queue {
public:
    explicit queue(const property_list& propList = {});

    explicit queue(const async_handler& asyncHandler,
                  const property_list& propList = {});

    template <typename DeviceSelector>
    explicit queue(const DeviceSelector& deviceSelector,
                  const property_list& propList = {});

    template <typename DeviceSelector>
    explicit queue(const DeviceSelector& 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 = {});

    template <typename DeviceSelector>
    explicit queue(const context& syclContext,
                  const DeviceSelector& deviceSelector,
                  const property_list& propList = {});

    template <typename DeviceSelector>
    explicit queue(const context& syclContext,
                  const DeviceSelector& deviceSelector,
                  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 = {});

    /* -- common interface members -- */

    /* -- property interface members -- */

    backend get_backend() const noexcept;

    context get_context() const;
};
```

```

device get_device() const;

bool is_in_order() const;

template <typename Param>
typename Param::return_type get_info() const;

template <typename Param>
typename Param::return_type get_backend_info() const;

template <typename T>
event submit(T cgf);

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

void wait();

void wait_and_throw();

void throw_asynchronous();

/* -- Shortcut functions: single_task -- */

template <typename KernelName, typename KernelType>
event single_task(const KernelType& kernelFunc);

template <typename KernelName, typename KernelType>
event single_task(event depEvent, const KernelType& kernelFunc);

template <typename KernelName, typename KernelType>
event single_task(const std::vector<event>& depEvents,
                  const KernelType& kernelFunc);

/* -- Shortcut functions: parallel_for -- */

template <typename KernelName, int Dims, typename... Rest>
event parallel_for(range<Dims> numWorkItems, Rest&&... rest);

template <typename KernelName, int Dims, typename... Rest>
event parallel_for(range<Dims> numWorkItems, event depEvent, Rest&&... rest);

template <typename KernelName, int Dims, typename... Rest>
event parallel_for(range<Dims> numWorkItems,
                  const std::vector<event>& depEvents, Rest&&... rest);

template <typename KernelName, int Dims, typename... Rest>
event parallel_for(nd_range<Dims> executionRange, Rest&&... rest);

template <typename KernelName, int Dims, typename... Rest>
event parallel_for(nd_range<Dims> executionRange, event depEvent,
                  Rest&&... rest);

template <typename KernelName, int Dims, typename... Rest>
event parallel_for(nd_range<Dims> executionRange,
                  const std::vector<event>& depEvents, Rest&&... rest);

```

```

/* -- Shortcut functions: memcpy -- */

event memcpy(void* dest, const void* src, size_t numBytes);
event memcpy(void* dest, const void* src, size_t numBytes, event depEvent);
event memcpy(void* dest, const void* src, size_t numBytes,
             const std::vector<event>& depEvents);

/* -- Shortcut functions: copy -- */

template <typename T>
event copy(const T* src, T* dest, size_t count);
template <typename T>
event copy(const T* src, T* dest, size_t count, event depEvent);
template <typename T>
event copy(const T* src, T* dest, size_t count,
          const std::vector<event>& depEvents);

template <typename SrcT, int SrcDims, access_mode SrcMode, target SrcTgt,
          access::placeholder IsPlaceholder, typename DestT>
event copy(accessor<SrcT, SrcDims, SrcMode, SrcTgt, IsPlaceholder> src,
          std::shared_ptr<DestT> dest);

template <typename SrcT, typename DestT, int DestDims, access_mode DestMode,
          target DestTgt, access::placeholder IsPlaceholder>
event copy(std::shared_ptr<SrcT> src,
          accessor<DestT, DestDims, DestMode, DestTgt, IsPlaceholder> dest);

template <typename SrcT, int SrcDims, access_mode SrcMode, target SrcTgt,
          access::placeholder IsPlaceholder, typename DestT>
event copy(accessor<SrcT, SrcDims, SrcMode, SrcTgt, IsPlaceholder> src,
          DestT* dest);

template <typename SrcT, typename DestT, int DestDims, access_mode DestMode,
          target DestTgt, access::placeholder IsPlaceholder>
event copy(const SrcT* src,
          accessor<DestT, DestDims, DestMode, DestTgt, IsPlaceholder> dest);

template <typename SrcT, int SrcDims, access_mode SrcMode, target SrcTgt,
          access::placeholder IsSrcPlaceholder, typename DestT, int DestDims,
          access_mode DestMode, target DestTgt,
          access::placeholder IsDestPlaceholder>
event copy(accessor<SrcT, SrcDims, SrcMode, SrcTgt, IsSrcPlaceholder> src,
          accessor<DestT, DestDims, DestMode, DestTgt, IsDestPlaceholder> dest);

/* -- Shortcut functions: memset -- */

event memset(void* ptr, int value, size_t numBytes);
event memset(void* ptr, int value, size_t numBytes, event depEvent);
event memset(void* ptr, int value, size_t numBytes,
             const std::vector<event>& depEvents);

/* -- Shortcut functions: fill -- */

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

```

```

template <typename T>
event fill(void* ptr, const T& pattern, size_t count, event depEvent);
template <typename T>
event fill(void* ptr, const T& pattern, size_t count,
           const std::vector<event>& depEvents);

template <typename T, int Dims, access_mode Mode, target Tgt,
         access::placeholder IsPlaceholder>
event fill(accessor<T, Dims, Mode, Tgt, IsPlaceholder> dest, const T& src);

/* -- Shortcut functions: prefetch -- */

event prefetch(void* ptr, size_t numBytes);
event prefetch(void* ptr, size_t numBytes, event depEvent);
event prefetch(void* ptr, size_t numBytes,
               const std::vector<event>& depEvents);

/* -- Shortcut functions: mem_advise -- */

event mem_advise(void* ptr, size_t numBytes, int advice);
event mem_advise(void* ptr, size_t numBytes, int advice, event depEvent);
event mem_advise(void* ptr, size_t numBytes, int advice,
                 const std::vector<event>& depEvents);

/* -- Shortcut functions: update_host -- */

template <typename T, int Dims, access_mode Mode, target Tgt,
         access::placeholder IsPlaceholder>
event update_host(accessor<T, Dims, Mode, Tgt, IsPlaceholder> acc);
};
} // namespace sycl

```

#### 4.6.5.1. Constructors

All queue constructors take a parameter named `propList` which allows the application to pass zero or more properties. These properties may specify additional effects of the constructor and may also specify exceptions that the constructor throws. See [Section 4.6.5.5](#) for the queue properties that are defined by the [core SYCL specification](#).

##### Default constructor

```
explicit queue(const property_list& propList = {})
```

*Effects:* Constructs a `queue` object using the device selected by `default_selector_v`. The queue's platform is the platform that contains this device. The queue's context is this platform's default context as described in [Section 4.6.2](#).

##### Constructor with async handler

```
explicit queue(const async_handler& asyncHandler,
               const property_list& propList = {})
```

*Effects:* Constructs a **queue** object using the device selected by **default\_selector\_v**. The queue's platform is the platform that contains this device. The queue's context is this platform's default context as described in [Section 4.6.2](#). The queue has the asynchronous error handler **asyncHandler**.

#### Constructor with device selector

```
template <typename DeviceSelector>
explicit queue(const DeviceSelector& deviceSelector,
               const property_list& propList = {})
```

*Constraints:* Available only when the **DeviceSelector** is a type that satisfies the requirements of a **device selector** as defined in [Section 4.6.1.1](#).

*Effects:* The **deviceSelector** is called for every **root device** as described in [Section 4.6.1.1](#), and a **queue** object is constructed using the device it selects. The queue's platform is the platform that contains this device. The queue's context is this platform's default context as described in [Section 4.6.2](#).

#### Constructor with device selector and async handler

```
template <typename DeviceSelector>
explicit queue(const DeviceSelector& deviceSelector,
               const async_handler& asyncHandler,
               const property_list& propList = {})
```

*Constraints:* Available only when the **DeviceSelector** is a type that satisfies the requirements of a **device selector** as defined in [Section 4.6.1.1](#).

*Effects:* The **deviceSelector** is called for every **root device** as described in [Section 4.6.1.1](#), and a **queue** object is constructed using the device it selects. The queue's platform is the platform that contains this device. The queue's context is this platform's default context as described in [Section 4.6.2](#). The queue has the asynchronous error handler **asyncHandler**.

#### Constructor with device

```
explicit queue(const device& syclDevice, const property_list& propList = {})
```

*Effects:* Constructs a **queue** object using the device **syclDevice**. The queue's platform is the platform that contains this device. The queue's context is this platform's default context as described in [Section 4.6.2](#).

#### Constructor with device and async handler

```
explicit queue(const device& syclDevice, const async_handler& asyncHandler,
               const property_list& propList = {})
```

*Effects:* Constructs a **queue** object using the device **syclDevice**. The queue's platform is the platform that contains this device. The queue's context is this platform's default context as described in [Section 4.6.2](#). The queue has the asynchronous error handler **asyncHandler**.



**Constructor with context and device selector**

```
template <typename DeviceSelector>
explicit queue(const context& syclContext, const DeviceSelector& deviceSelector,
              const property_list& propList = {})
```

*Constraints:* Available only when the `DeviceSelector` is a type that satisfies the requirements of a `device selector` as defined in [Section 4.6.1.1](#).

*Effects:* The `deviceSelector` is called for every `root device` as described in [Section 4.6.1.1](#), and a `queue` object is constructed using the device it selects. The queue's platform is the platform that contains this device. The queue's context is `syclContext`.

*Throws:* An `exception` with the `errc::invalid` error code if `syclContext` does not contain the device selected by `deviceSelector`.

**Constructor with context, device selector, and async handler**

```
template <typename DeviceSelector>
explicit queue(const context& syclContext, const DeviceSelector& deviceSelector,
              const async_handler& asyncHandler,
              const property_list& propList = {})
```

*Constraints:* Available only when the `DeviceSelector` is a type that satisfies the requirements of a `device selector` as defined in [Section 4.6.1.1](#).

*Effects:* The `deviceSelector` is called for every `root device` as described in [Section 4.6.1.1](#), and a `queue` object is constructed using the device it selects. The queue's platform is the platform that contains this device. The queue's context is `syclContext`. The queue has the asynchronous error handler `asyncHandler`.

*Throws:* An `exception` with the `errc::invalid` error code if `syclContext` does not contain the device selected by `deviceSelector`.

**Constructor with context and device**

```
explicit queue(const context& syclContext, const device& syclDevice,
              const property_list& propList = {})
```

*Effects:* Constructs a `queue` object using the device `syclDevice`. The queue's platform is the platform that contains this device. The queue's context is `syclContext`.

*Throws:* An `exception` with the `errc::invalid` error code unless `syclDevice` is contained by `syclContext` or is a `descendent device` of some device that is contained by `syclContext`.

**Constructor with context, device, and async handler**

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

*Effects:* Constructs a `queue` object using the device `syclDevice`. The queue's platform is the platform that

contains this device. The queue's context is `syclContext`. The queue has the asynchronous error handler `asyncHandler`.

*Throws:* An `exception` with the `errc::invalid` error code unless `syclDevice` is contained by `syclContext` or is a `descendent device` of some device that is contained by `syclContext`.

#### 4.6.5.2. Member functions

##### `queue::get_backend`

```
backend get_backend() const noexcept
```

*Returns:* The SYCL backend that is associated with this queue.

##### `queue::get_context`

```
context get_context() const
```

*Returns:* The context that is associated with this queue.

##### `queue::get_device`

```
device get_device() const
```

*Returns:* The device that is associated with this queue.

##### `queue::is_in_order`

```
bool is_in_order() const
```

*Returns:* The same value as `has_property<property::queue::in_order>()`.

##### `queue::get_info`

```
template <typename Param>
typename Param::return_type get_info() const
```

*Constraints:* Available only when `Param` is an information descriptor for the queue class.

Each information descriptor specifies the return value and may also specify preconditions, exceptions that are thrown, etc. See [Section 4.6.5.4](#) for the queue information descriptors that are defined by the core SYCL specification.

##### `queue::get_backend_info`

```
template <typename Param>
```

```
typename Param::return_type get_backend_info() const
```

*Constraints:* Available only when `Param` is a backend information descriptor for the queue class.

*Throws:* An `exception` with the `errc::backend_mismatch` error code if the backend that corresponds with `Param` is different from the backend that is associated with this queue.

Each information descriptor specifies the return value and may also specify preconditions, additional exceptions that are thrown, etc.

#### `queue::submit`

```
template <typename T>
event submit(T cgf)
```

*Effects:* Immediately calls the `command group function object` `cgf`, which may submit no more than one `command` to the queue for execution on the device.

*Returns:* An event which represents the `command` which is submitted to the queue.

#### `queue::submit (with secondary queue)`

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

*Effects:* Immediately calls the `command group function object` `cgf`, which may submit no more than one `command` to the queue for execution on the device. On a kernel error, this `command group function object` may be scheduled for execution on the secondary queue `secondaryQueue` as described in [Section 3.9.10](#).

*Returns:* An event which represents the `command` which is submitted to the queue. If the command is scheduled on `secondaryQueue`, the event is associated with that queue.

#### `queue::wait`

```
void wait()
```

*Effects:* Blocks the calling thread until all commands previously submitted to this queue have completed. Synchronous errors are reported through SYCL exceptions.

#### `queue::wait_and_throw`

```
void wait_and_throw()
```

*Effects:* Blocks the calling thread until all commands previously submitted to this queue have completed. Synchronous errors are reported through SYCL exceptions.

At least all unconsumed `asynchronous errors` held by this queue (or its associated context) are passed to the appropriate `async_handler` as described in [Section 4.13.1.3](#). ""

**queue::throw\_asynchronous**

```
void throw_asynchronous()
```

*Effects:* Checks to see if any unconsumed [asynchronous errors](#) have been produced by the queue and if so reports them by passing them to the [async\\_handler](#) associated with the queue or to the [async\\_handler](#) associated with the queue's context. If no user defined asynchronous error handler is associated with the queue or its context, then an implementation-defined default [async\\_handler](#) is called to handle any errors, as described in [Section 4.13.1.2](#).

**4.6.5.3. Shortcut member functions**

The functions described in this section are shortcuts for `queue::submit` that allow an application to submit a command to the queue without defining a [command group function object](#). Each of these functions implicitly creates a command group that acts as though it calls one of the [handler](#) member functions to submit a single command. For example, `queue::single_task` creates a command group that acts as though it calls `handler::single_task`. These shortcut functions return an [event](#) object that represents the command that is submitted to the queue. In addition, some forms of the shortcut functions allow the application to pass input events, and these forms act as though the command group calls `handler::depends_on` with these same events.

Because there is no explicit command group function when using these shortcuts, it is not possible to create accessors for the command that is submitted. Therefore, kernels that are submitted using these shortcuts must not use accessors. Typically, applications use USM pointers instead. However, there is a special exception for non-kernel commands (e.g. shortcuts for the explicit memory copy commands). These non-kernel commands may use placeholder accessors, and the implicit command group function acts as though it calls `handler::require` on each of the placeholder accessors that the shortcut function uses.

The following example demonstrates the use of these shortcut functions.

```
1 class MyKernel;
2
3 queue myQueue;
4 auto usmPtr = malloc_device<int>(1024, myQueue); // USM pointer
5
6 int* data = /* pointer to some data */;
7 buffer buf{data, 1024};
8 accessor acc{buf}; // Placeholder accessor
9
10 // Queue shortcut for a kernel invocation
11 myQueue.single_task<MyKernel>([=] {
12     // Allowed to use USM pointers,
13     // not allowed to use accessors
14     usmPtr[0] = 0;
15 });
16
17 // Placeholder accessor will automatically be registered
18 myQueue.copy(data, acc);
```

**queue::single\_task**

```

template <typename KernelName, typename KernelType>           (1)
event single_task(const KernelType& kernelFunc)

template <typename KernelName, typename KernelType>           (2)
event single_task(event depEvent, const KernelType& kernelFunc)

template <typename KernelName, typename KernelType>           (3)
event single_task(const std::vector<event>& depEvents,
                  const KernelType& kernelFunc)

```

*Effects (1):* Equivalent to calling `queue::submit` with a command group function that calls `handler::single_task(kernelFunc)`.

*Effects (2):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvent)` and `handler::single_task(kernelFunc)`.

*Effects (3):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvents)` and `handler::single_task(kernelFunc)`.

*Returns:* An event which represents the `command` which is submitted to the queue.

**queue::parallel\_for**

```

template <typename KernelName, int Dimensions, typename... Rest> (1)
event parallel_for(range<Dimensions> numWorkItems, Rest&&... rest)

template <typename KernelName, int Dimensions, typename... Rest> (2)
event parallel_for(range<Dimensions> numWorkItems, event depEvent,
                  Rest&&... rest)

template <typename KernelName, int Dimensions, typename... Rest> (3)
event parallel_for(range<Dimensions> numWorkItems,
                  const std::vector<event>& depEvents, Rest&&... rest)

template <typename KernelName, int Dimensions, typename... Rest> (4)
event parallel_for(nd_range<Dimensions> executionRange, Rest&&... rest)

template <typename KernelName, int Dimensions, typename... Rest> (5)
event parallel_for(nd_range<Dimensions> executionRange, event depEvent,
                  Rest&&... rest)

template <typename KernelName, int Dimensions, typename... Rest> (6)
event parallel_for(nd_range<Dimensions> executionRange,
                  const std::vector<event>& depEvents, Rest&&... rest)

```

*Effects (1):* Equivalent to calling `queue::submit` with a command group function that calls `handler::parallel_for(numWorkItems, rest)`.

*Effects (2):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvent)` and `handler::parallel_for(numWorkItems, rest)`.

*Effects (3):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvents)` and `handler::parallel_for(numWorkItems, rest)`.

*Effects (4):* Equivalent to calling `queue::submit` with a command group function that calls `handler::parallel_for(executionRange, rest)`.

*Effects (5):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvent)` and `handler::parallel_for(executionRange, rest)`.

*Effects (6):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvents)` and `handler::parallel_for(executionRange, rest)`.

*Returns:* An event which represents the `command` which is submitted to the queue.

#### `queue::memcpy`

```
event memcpy(void* dest, const void* src, size_t numBytes) (1)
```

```
event memcpy(void* dest, const void* src, size_t numBytes, event depEvent) (2)
```

```
event memcpy(void* dest, const void* src, size_t numBytes, (3)
               const std::vector<event>& depEvents)
```

*Effects (1):* Equivalent to calling `queue::submit` with a command group function that calls `handler::memcpy(dest, src, numBytes)`.

*Effects (2):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvent)` and `handler::memcpy(dest, src, numBytes)`.

*Effects (3):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvents)` and `handler::memcpy(dest, src, numBytes)`.

*Returns:* An event which represents the `command` which is submitted to the queue.

#### `queue::copy`

```
template <typename T> (1)
event copy(const T* src, T* dest, size_t count)
```

```
template <typename T> (2)
event copy(const T* src, T* dest, size_t count, event depEvent)
```

```
template <typename T> (3)
event copy(const T* src, T* dest, size_t count,
           const std::vector<event>& depEvents)
```

```
template <typename SrcT, int SrcDims, access_mode SrcMode, target SrcTgt, (4)
          access::placeholder IsPlaceholder, typename DestT>
event copy(accessor<SrcT, SrcDims, SrcMode, SrcTgt, IsPlaceholder> src,
           std::shared_ptr<DestT> dest)
```

```
template <typename SrcT, typename DestT, int DestDims, access_mode DestMode, (5)
          target DestTgt, access::placeholder IsPlaceholder>
event copy(std::shared_ptr<SrcT> src,
           accessor<DestT, DestDims, DestMode, DestTgt, IsPlaceholder> dest)
```

```
template <typename SrcT, int SrcDims, access_mode SrcMode, target SrcTgt, (6)
          access::placeholder IsPlaceholder, typename DestT>
```

```

event copy(accessor<SrcT, SrcDims, SrcMode, SrcTgt, IsPlaceholder> src,
           DestT* dest)

template <typename SrcT, typename DestT, int DestDims, access_mode DestMode, (7)
         target DestTgt, access::placeholder IsPlaceholder>
event copy(const SrcT* src,
           accessor<DestT, DestDims, DestMode, DestTgt, IsPlaceholder> dest)

template <typename SrcT, int SrcDims, access_mode SrcMode, target SrcTgt, (8)
         access::placeholder IsSrcPlaceholder, typename DestT, int DestDims,
         access_mode DestMode, target DestTgt,
         access::placeholder IsDestPlaceholder>
event copy(
    accessor<SrcT, SrcDims, SrcMode, SrcTgt, IsSrcPlaceholder> src,
    accessor<DestT, DestDims, DestMode, DestTgt, IsDestPlaceholder> dest)

```

*Effects (1):* Equivalent to calling `queue::submit` with a command group function that calls `handler::copy(src, dest, count)`.

*Effects (2):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvent)` and `handler::copy(src, dest, count)`.

*Effects (3):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvents)` and `handler::copy(src, dest, count)`.

*Effects (4):* Equivalent to calling `queue::submit` with a command group function that calls `handler::require(src)` and `handler::copy(src, dest)`.

*Effects (5):* Equivalent to calling `queue::submit` with a command group function that calls `handler::require(dest)` and `handler::copy(src, dest)`.

*Effects (6):* Equivalent to calling `queue::submit` with a command group function that calls `handler::require(src)` and `handler::copy(src, dest)`.

*Effects (7):* Equivalent to calling `queue::submit` with a command group function that calls `handler::require(dest)` and `handler::copy(src, dest)`.

*Effects (8):* Equivalent to calling `queue::submit` with a command group function that calls `handler::require(src)`, `handler::require(dest)`, and `handler::copy(src, dest)`.

*Returns:* An event which represents the `command` which is submitted to the queue.

#### `queue::memset`

```

event memset(void* ptr, int value, size_t numBytes) (1)

event memset(void* ptr, int value, size_t numBytes, event depEvent) (2)

event memset(void* ptr, int value, size_t numBytes, (3)
             const std::vector<event>& depEvents)

```

*Effects (1):* Equivalent to calling `queue::submit` with a command group function that calls `handler::memset(ptr, value, numBytes)`.

*Effects (2):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvent)` and `handler::memset(ptr, value, numBytes)`.

*Effects (3):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvents)` and `handler::memcpy(ptr, value, numBytes)`.

*Returns:* An event which represents the `command` which is submitted to the queue.

#### `queue::fill`

```
template <typename T> (1)
event fill(void* ptr, const T& pattern, size_t count)

template <typename T> (2)
event fill(void* ptr, const T& pattern, size_t count, event depEvent)

template <typename T> (3)
event fill(void* ptr, const T& pattern, size_t count,
           const std::vector<event>& depEvents)

template <typename T, int Dims, access_mode Mode, target Tgt, (4)
           access::placeholder IsPlaceholder>
event fill(accessor<T, Dims, Mode, Tgt, IsPlaceholder> dest, const T& src)
```

*Effects (1):* Equivalent to calling `queue::submit` with a command group function that calls `handler::fill(ptr, pattern, count)`.

*Effects (2):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvent)` and `handler::fill(ptr, pattern, count)`.

*Effects (3):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvents)` and `handler::fill(ptr, pattern, count)`.

*Effects (4):* Equivalent to calling `queue::submit` with a command group function that calls `handler::require(dest)` and `handler::fill(dest, src)`.

*Returns:* An event which represents the `command` which is submitted to the queue.

#### `queue::prefetch`

```
event prefetch(void* ptr, size_t numBytes) (1)

event prefetch(void* ptr, size_t numBytes, event depEvent) (2)

event prefetch(void* ptr, size_t numBytes, const std::vector<event>& depEvents) (3)
```

*Effects (1):* Equivalent to calling `queue::submit` with a command group function that calls `handler::prefetch(ptr, numBytes)`.

*Effects (2):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvent)` and `handler::prefetch(ptr, numBytes)`.

*Effects (3):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvents)` and `handler::prefetch(ptr, numBytes)`.

*Returns:* An event which represents the `command` which is submitted to the queue.



**queue::mem\_advise**

```

event mem_advise(void* ptr, size_t numBytes, int advice) (1)

event mem_advise(void* ptr, size_t numBytes, int advice, event depEvent) (2)

event mem_advise(void* ptr, size_t numBytes, int advice,
                  const std::vector<event>& depEvents) (3)

```

*Effects (1):* Equivalent to calling `queue::submit` with a command group function that calls `handler::mem_advise(ptr, numBytes, advice)`.

*Effects (2):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvent)` and `handler::mem_advise(ptr, numBytes, advice)`.

*Effects (3):* Equivalent to calling `queue::submit` with a command group function that calls `handler::depends_on(depEvents)` and `handler::mem_advise(ptr, numBytes, advice)`.

*Returns:* An event which represents the `command` which is submitted to the queue.

**queue::update\_host**

```

template <typename T, int Dims, access_mode Mode, target Tgt,
          access::placeholder IsPlaceholder>
event update_host(accessor<T, Dims, Mode, Tgt, IsPlaceholder> acc)

```

*Effects:* Equivalent to calling `queue::submit` with a command group function that calls `handler::require(acc)` and `handler::update_host(acc)`.

*Returns:* An event which represents the `command` which is submitted to the queue.

**4.6.5.4. Information descriptors**

This section describes the information descriptors that can be used as the `Param` template parameter to `queue::get_info`. When the description has a *Returns*, *Throws*, etc. paragraph, this indicates the value returned by or the exceptions thrown by the `queue::get_info` function.

**info::queue::context**

```

namespace sycl::info::queue {
struct context {
    using return_type = context;
};
} // namespace sycl::info::queue

```

*Remarks:* Template parameter to `queue::get_info`.

*Returns:* The `context` that is associated with this queue.

**info::queue::device**

```
namespace sycl::info::queue {
  struct device {
    using return_type = device;
  };
} // namespace sycl::info::queue
```

*Remarks:* Template parameter to `queue::get_info`.

*Returns:* The `device` that is associated with this queue.

**4.6.5.5. Properties**

This section describes the properties that can be passed in the `propList` parameter of the `queue` constructors.

**property::queue::enable\_profiling**

```
namespace sycl::property::queue {
  struct enable_profiling {
    enable_profiling(); (1)
  };
} // namespace sycl::property::queue
```

When a queue is constructed with this property, the implementation captures profiling information for the `command groups` that are submitted to this queue. Applications can retrieve this profiling information by calling `event::get_profiling_info` on the event that is returned when submitting the command group. If the queue's associated device does not have `aspect::queue_profiling`, passing this property to the queue's constructor causes the constructor to throw a synchronous `exception` with the `errc::feature_not_supported` error code.

*Effects (1):* Constructs an `enable_profiling` property object.

**property::queue::in\_order**

```
namespace sycl::property::queue {
  struct in_order {
    in_order(); (1)
  };
} // namespace sycl::property::queue
```

When a queue is constructed with this property, commands that are submitted to the queue are guaranteed to execute in the order in which they are submitted, as if there is an implicit dependency on the previous command that was submitted to the same queue. The `in_order` property does not provide any guarantee about the order of commands submitted to other queues with respect to commands submitted to this queue.

*Effects (1):* Constructs an `in_order` property object.

#### 4.6.5.6. Error handling

Queue errors come in two forms:

- Synchronous errors are those that we would expect to be reported directly at the point of waiting on an event, and hence waiting for a queue to complete, as well as any immediate errors reported by enqueueing work onto a queue. Such errors are reported through C++ exceptions.
- [Asynchronous errors](#) are those that are produced or detected after associated host API calls have returned (so can't be thrown as exceptions by the API call), and that are handled by an [async\\_handler](#) through which the errors are reported. Handling of asynchronous errors from a queue occurs at specific times, as described by [Section 4.13](#).

Note that if there are [asynchronous errors](#) to be processed when a queue is destroyed, the handler is called and this might delay or block the destruction, according to the behavior of the handler.

#### 4.6.6. Event class

An [event](#) in SYCL is an object that represents the status of an operation that is being executed by the SYCL runtime.

Typically in SYCL, data dependency and execution order is handled implicitly by the SYCL runtime. However, in some circumstances developers want fine grain control of the execution, or want to retrieve properties of a command that is running.

Note that, although an event represents the status of a particular operation, the dependencies of a certain event can be used to keep track of multiple steps required to block on the results of said operation.

A SYCL event is returned by the submission of a [command group](#). The dependencies of the event returned via the submission of the command group are the implementation-defined commands associated with the [command group](#) execution.

The SYCL [event](#) class provides the common reference semantics (see [Section 4.5.2](#)).

The constructors and member functions of the SYCL [event](#) class are listed in [Table 15](#) and [Table 16](#), respectively. The additional common special member functions and common member functions are listed in [Table 7](#) and [Table 8](#), respectively.

```

1 namespace sycl {
2
3 class event {
4 public:
5     event();
6
7     /* -- common interface members -- */
8
9     backend get_backend() const noexcept;
10
11     std::vector<event> get_wait_list();
12
13     void wait();
14
15     static void wait(const std::vector<event>& eventList);
16
17     void wait_and_throw();
18
19     static void wait_and_throw(const std::vector<event>& eventList);
20

```

```

21  template <typename Param> typename Param::return_type get_info() const;
22
23  template <typename Param>
24  typename Param::return_type get_backend_info() const;
25
26  template <typename Param>
27  typename Param::return_type get_profiling_info() const;
28 };
29
30 } // namespace sycl


```

Table 15. Constructors of the `event` class

Constructor	Description
<code>event()</code>	<p>Constructs an <code>event</code> that is immediately ready. The <code>event</code> has no dependencies and no associated commands. Waiting on this <code>event</code> will return immediately and querying its status will return <code>info::event_command_status::complete</code>.</p> <p>The event is constructed as though it was created from a default-constructed <code>queue</code>. Therefore, its backend is the same as the backend from the default device.</p>

Table 16. Member functions for the `event` class

Member function	Description
<code>backend get_backend() const noexcept</code>	Returns a <code>backend</code> identifying the SYCL backend associated with this <code>event</code> .
<code>std::vector&lt;event&gt; get_wait_list()</code>	Return the list of events that this event waits for in the dependence graph. Only direct dependencies are returned, and not transitive dependencies that direct dependencies wait on. Whether already completed events are included in the returned list is implementation-defined.
<code>void wait()</code>	Blocks until all commands associated with this event and any dependent events have completed.

Member function	Description
<code>void wait_and_throw()</code>	<p>Blocks until all commands associated with this event and any dependent events have completed.</p> <p>At least all unconsumed <a href="#">asynchronous errors</a> held by queues (or their associated contexts) which were used to enqueue commands associated with this event and any dependent events, are passed to the appropriate <a href="#">async_handler</a> as described in <a href="#">Section 4.13.1.3</a>.</p> <div>  <p>This behavior is equivalent to calling <code>queue::throw_asynchronous()</code> on the queue associated with this event and any dependent events.</p> </div>
<code>static void wait(const std::vector&lt;event&gt;&amp; eventList)</code>	Behaves as if calling <code>event::wait()</code> on each event in <code>eventList</code> .
<code>static void wait_and_throw(const std::vector&lt;event&gt;&amp; eventList)</code>	Behaves as if calling <code>event::wait_and_throw()</code> on each event in <code>eventList</code> .
<code>template &lt;typename Param&gt; typename Param::return_type get_info() const</code>	Queries this SYCL <a href="#">event</a> for information requested by the template parameter <code>Param</code> . The type alias <code>Param::return_type</code> must be defined in accordance with the info parameters in <a href="#">Table 17</a> to facilitate returning the type associated with the <code>Param</code> parameter.
<code>template &lt;typename Param&gt; typename Param::return_type get_backend_info() const</code>	Queries this SYCL <a href="#">event</a> for <a href="#">SYCL backend</a> -specific information requested by the template parameter <code>Param</code> . The type alias <code>Param::return_type</code> must be defined in accordance with the <a href="#">SYCL backend</a> specification. Must throw an <a href="#">exception</a> with the <code>errc::backend_mismatch</code> error code if the <a href="#">SYCL backend</a> that corresponds with <code>Param</code> is different from the <a href="#">SYCL backend</a> that is associated with this <a href="#">event</a> .

Member function	Description
<pre>template &lt;typename Param&gt; typename Param::return_type get_profiling_info() const</pre>	<p>Queries this SYCL <b>event</b> for profiling information requested by the parameter <b>Param</b>. If the requested profiling information is unavailable when <b>get_profiling_info</b> is called due to incompleteness of <b>command groups</b> associated with the <b>event</b>, then the call to <b>get_profiling_info</b> will block until the requested profiling information is available. An example is asking for <b>info::event_profiling::command_end</b> when the associated <b>command group</b> action has yet to finish execution. Calls to <b>get_profiling_info</b> must throw an <b>exception</b> with the <b>errc::invalid</b> error code if the SYCL <b>queue</b> which submitted the <b>command group</b> this SYCL <b>event</b> is associated with was not constructed with the <b>property::queue::enable_profiling</b> property. The type alias <b>Param::return_type</b> must be defined in accordance with the info parameters in <a href="#">Table 19</a> to facilitate returning the type associated with the <b>Param</b> parameter.</p>

#### 4.6.6.1. Event information and profiling descriptors

An **event** can be queried for information using the **get\_info** member function of the **event** class, specifying one of the info parameters in **info::event**. The possible values for each info parameter and any restrictions are defined in the specification of the SYCL **backend** associated with the **event**. All info parameters in **info::event** are specified in [Table 17](#) and the synopsis for **info::event** is described in [Section A.6](#).

Table 17. Event class information descriptors

Event Descriptors	Return type	Description
<b>info::event::command_execution_status</b>	<b>info::event_command_status</b>	Returns the event status of the command group and contained action (e.g. kernel invocation) associated with this SYCL <b>event</b> .

The **info::event::command\_execution\_status** query returns one of the values defined in [Table 18](#).

Table 18. Event command status

Status	Description
<b>info::event_command_status::submitted</b>	Indicates that the command has been submitted to the SYCL queue but has not yet started running on the device.

Status	Description
<code>info::event_command_status::running</code>	Indicates that the command has started running on the device but has not yet completed.
<code>info::event_command_status::complete</code>	Indicates that the command has finished running on the device. Attempting to wait on such an event will not block.

An [event](#) can be queried for profiling information using the `get_profiling_info` member function of the `event` class, specifying one of the profiling info parameters enumerated in `info::event_profiling`. The possible values for each info parameter and any restrictions are defined in the specification of the [SYCL backend](#) associated with the `event`. All info parameters in `info::event_profiling` are specified in [Table 19](#) and the synopsis for `info::event_profiling` is described in [Section A.6](#).

Each profiling descriptor returns a 64-bit timestamp that represents the number of nanoseconds that have elapsed since some implementation-defined timebase. All events that share the same backend are guaranteed to share the same timebase, therefore the difference between two timestamps from the same backend yields the number of nanoseconds that have elapsed between those events.

Table 19. Profiling information descriptors for the SYCL `event` class

Event information profiling descriptor	Return type	Description
<code>info::event_profiling::command_submit</code>	<code>uint64_t</code>	Returns a timestamp telling when the associated <a href="#">command group</a> was submitted to the <a href="#">queue</a> . This is always some time after the <a href="#">command group function object</a> returns and before the associated call to <code>queue::submit</code> returns.
<code>info::event_profiling::command_start</code>	<code>uint64_t</code>	Querying this profiling descriptor blocks until the event's state becomes either <code>info::event_command_status::running</code> or <code>info::event_command_status::complete</code> . The returned timestamp tells when the action associated with the <a href="#">command group</a> (e.g. kernel invocation) started executing on the device. For any given event, this timestamp is always greater than or equal to the <code>info::event_profiling::command_submit</code> timestamp. Implementations are encouraged to return a timestamp that is as close as possible to the point when the action starts running on the device, but there is no specific accuracy that is guaranteed.
<code>info::event_profiling::command_end</code>	<code>uint64_t</code>	Querying this profiling descriptor blocks until the event's state becomes <code>info::event_command_status::complete</code> . The returned timestamp tells when the action associated with the <a href="#">command group</a> (e.g. kernel invocation) finished executing on the device. For any given event, this timestamp is always greater than or equal to the <code>info::event_profiling::command_start</code> timestamp.

## 4.7. Data access and storage in SYCL

In SYCL, when using [buffers](#) and [images](#), data storage and access are handled by separate classes. [Buffers](#) and [images](#) handle storage and ownership of the data, whereas [accessors](#) handle access to the data. Buffers and images in SYCL can be bound to more than one device or context, including across different [SYCL backends](#). They also handle ownership of the data, while allowing exception handling for blocking and non-blocking data transfers. Accessors manage data transfers between the host and all of the devices in the system, as well as tracking of data dependencies.

Zero-sized buffers and accessors are permitted, but attempting to access data within them produces undefined behavior, similar to dereferencing a null pointer in C++. Note that zero-sized accessors can be created in several ways: by creating an accessor from a zero-sized buffer, by creating an accessor with a zero-sized buffer sub-range, or by creating an accessor with its default constructor.

When using [USM](#) allocations, data storage is managed by USM allocation functions, and data access is via pointers. See [Section 4.8](#) for greater detail.

### 4.7.1. Host allocation

A [SYCL runtime](#) may need to allocate temporary objects on the host to handle some operations (such as copying data from one context to another). Allocation on the host is managed using an allocator object, following the standard C++ allocator class definition. The default allocator for memory objects is implementation-defined, but the user can supply their own allocator class.

```
1 {
2     buffer<int, 1, UserDefinedAllocator<int>> b(d);
3 }
```

When an allocator returns a [nullptr](#), the runtime cannot allocate data on the host. Note that in this case the runtime will raise an error if it requires host memory but it is not available (e.g. when moving data across [SYCL backend](#) contexts).

In some cases, the implementation may retain a copy of the allocator object even after the buffer is destroyed. For example, this can happen when the buffer object is destroyed before commands using accessors to the buffer have completed. Therefore, the application must be prepared for calls to the allocator even after the buffer is destroyed.



If the application needs to know when the implementation has destroyed all copies of the allocator, it can maintain a reference count within the allocator.

The definition of allocators extends the current functionality of SYCL, ensuring that users can define allocator functions for specific hardware or certain complex shared memory mechanisms (e.g. NUMA), and improves interoperability with STL-based libraries (e.g. Intel's TBB provides an allocator).

#### 4.7.1.1. Default allocators

A default allocator is always defined by the implementation. For allocations greater than size zero, when successful it is guaranteed to return non-[nullptr](#) and new memory positions every call. The default allocator for const buffers will remove the const-ness of the type (therefore, the default allocator for a buffer of type `const int` will be an `Allocator<int>`). This implies that host [accessors](#) will not share memory with the pointer given by the user in the buffer/image constructor, but will use the memory returned by the `Allocator` itself for that purpose. The user can implement an allocator that returns the same address as the one passed in the buffer constructor, but it is the responsibility of the user to handle the potential race conditions.



Table 20. SYCL Default Allocators

Allocators	Description
<code>template &lt;class T&gt; buffer_allocator</code>	It is the default buffer allocator used by the runtime, when no allocator is defined by the user. Meets the C++ named requirement <code>Allocator</code> . A buffer of data type <code>const T</code> uses <code>buffer_allocator&lt;T&gt;</code> by default.
<code>image_allocator</code>	It is the default allocator used by the runtime for the SYCL <code>unsampled_image</code> and <code>sampled_image</code> classes when no allocator is provided by the user. The <code>image_allocator</code> is required to allocate in elements of <code>std::byte</code> .

See [Section 4.7.5](#) for details of using manual synchronization to avoid data races between host and device.

## 4.7.2. Buffers

The `buffer` class defines a shared array of one, two or three dimensions that can be used by the SYCL `kernel` and has to be accessed using `accessor` classes. Buffers are templated on both the type of their data, and the number of dimensions that the data is stored and accessed through.

A `buffer` does not map to only one underlying backend object, and all SYCL `backend` memory objects may be temporary for use within a command group on a specific device.

The underlying data type of a buffer `T` must be `device copyable` as defined in [Section 3.13.1](#). Some overloads of the `buffer` constructor initialize the buffer contents by copying objects from host memory while other overloads construct the buffer without copying objects from the host. For the overloads that do not copy host objects, the initial state of the objects in the buffer depends on whether `T` is an implicit-lifetime type (as defined in the C++ core language). If `T` is an implicit-lifetime type, objects of that type are implicitly created in the buffer with indeterminate values. For other types, these constructor overloads merely allocate uninitialized memory, and the application is responsible for constructing objects by calling placement-new and for destroying them later by manually calling the object's destructor.

For the overloads that do copy objects from host memory, the `hostData` pointer must point to at least  $N$  bytes of memory where  $N$  is `sizeof(T) * bufferRange.size()`. If  $N$  is zero, `hostData` is permitted to be a null pointer.

A SYCL `buffer` can construct an instance of a SYCL `buffer` that reinterprets the original SYCL `buffer` with a different type, dimensionality and range using the member function `reinterpret`. The reinterpreted SYCL `buffer` that is constructed must behave as though it were a copy of the SYCL `buffer` that constructed it (see [Section 4.5.2](#)) with the exception that the type, dimensionality and range of the reinterpreted SYCL `buffer` must reflect the type, dimensionality and range specified when calling the `reinterpret` member function. By extension of this, the class member types `value_type`, `reference` and `const_reference`, and the member functions `get_range()` and `size()` of the reinterpreted SYCL `buffer` must reflect the new type, dimensionality and range. The data that the original SYCL `buffer` and the reinterpreted SYCL `buffer` manage remains unaffected, though the representation of the data when accessed through the reinterpreted SYCL `buffer` may alter to reflect the new type, dimensionality and range. It is important to note that a reinterpreted SYCL `buffer` is a copy of the original SYCL `buffer` only, and not a new SYCL `buffer`. Constructing more than one SYCL `buffer` managing the same host pointer is still undefined behavior.

The SYCL `buffer` class template provides the common reference semantics (see [Section 4.5.2](#)).

#### 4.7.2.1. Buffer interface

The constructors and member functions of the SYCL `buffer` class template are listed in [Table 21](#) and [Table 22](#), respectively. The additional common special member functions and common member functions are listed in [Table 7](#) and [Table 8](#), respectively.

Each constructor takes as the last parameter an optional SYCL `property_list` to provide properties to the SYCL `buffer`.

The SYCL `buffer` class template takes a template parameter `AllocatorT` for specifying an allocator which is used by the `SYCL runtime` when allocating temporary memory on the host. If no template argument is provided, then the default allocator for the SYCL `buffer` class `buffer_allocator<T>` will be used (see [Section 4.7.1.1](#)).

```

1 namespace sycl {
2 namespace property {
3 namespace buffer {
4 class use_host_ptr {
5 public:
6     use_host_ptr() = default;
7 };
8
9 class use_mutex {
10 public:
11     use_mutex(std::mutex& mutexRef);
12
13     std::mutex* get_mutex_ptr() const;
14 };
15
16 class context_bound {
17 public:
18     context_bound(context boundContext);
19
20     context get_context() const;
21 };
22 } // namespace buffer
23 } // namespace property
24
25 template <typename T, int Dimensions = 1,
26           typename AllocatorT = buffer_allocator<std::remove_const_t<T>>>
27 class buffer {
28 public:
29     using value_type = T;
30     using reference = value_type&
31     using const_reference = const value_type&
32     using allocator_type = AllocatorT;
33
34     buffer(const range<Dimensions>& bufferRange,
35           const property_list& proplist = {});
36
37     buffer(const range<Dimensions>& bufferRange, AllocatorT allocator,
38           const property_list& proplist = {});
39
40     buffer(T* hostData, const range<Dimensions>& bufferRange,
```

```

41     const property_list& proplist = {});
42
43     buffer(T* hostData, const range<Dimensions>& bufferRange,
44           AllocatorT allocator, const property_list& proplist = {});
45
46     buffer(const T* hostData, const range<Dimensions>& bufferRange,
47           const property_list& proplist = {});
48
49     buffer(const T* hostData, const range<Dimensions>& bufferRange,
50           AllocatorT allocator, const property_list& proplist = {});
51
52     /* Available only if Container is a contiguous container:
53        - std::data(container) and std::size(container) are well formed
54        - return type of std::data(container) is convertible to T*
55        and Dimensions == 1 */
56     template <typename Container>
57     buffer(Container& container, AllocatorT allocator,
58           const property_list& proplist = {});
59
60     /* Available only if Container is a contiguous container:
61        - std::data(container) and std::size(container) are well formed
62        - return type of std::data(container) is convertible to T*
63        and Dimensions == 1 */
64     template <typename Container>
65     buffer(Container& container, const property_list& proplist = {});
66
67     buffer(const std::shared_ptr<T>& hostData,
68           const range<Dimensions>& bufferRange, AllocatorT allocator,
69           const property_list& proplist = {});
70
71     buffer(const std::shared_ptr<T>& hostData,
72           const range<Dimensions>& bufferRange,
73           const property_list& proplist = {});
74
75     buffer(const std::shared_ptr<T[]>& hostData,
76           const range<Dimensions>& bufferRange, AllocatorT allocator,
77           const property_list& proplist = {});
78
79     buffer(const std::shared_ptr<T[]>& hostData,
80           const range<Dimensions>& bufferRange,
81           const property_list& proplist = {});
82
83     template <class InputIterator>
84     buffer<T, 1>(InputIterator first, InputIterator last, AllocatorT allocator,
85           const property_list& proplist = {});
86
87     template <class InputIterator>
88     buffer<T, 1>(InputIterator first, InputIterator last,
89           const property_list& proplist = {});
90
91     buffer(buffer& b, const id<Dimensions>& baseIndex,
92           const range<Dimensions>& subRange);
93
94     /* -- common interface members -- */
95
96     /* -- property interface members -- */

```

```

97
98  range<Dimensions> get_range() const;
99
100 size_t byte_size() const noexcept;
101
102 size_t size() const noexcept;
103
104 // Deprecated
105 size_t get_count() const;
106
107 // Deprecated
108 size_t get_size() const;
109
110 AllocatorT get_allocator() const;
111
112 template <access_mode Mode = access_mode::read_write,
113          target Targ = target::device>
114 accessor<T, Dimensions, Mode, Targ> get_access(handler& commandGroupHandler);
115
116 // Deprecated
117 template <access_mode Mode>
118 accessor<T, Dimensions, Mode, target::host_buffer> get_access();
119
120 template <access_mode Mode = access_mode::read_write,
121          target Targ = target::device>
122 accessor<T, Dimensions, Mode, Targ>
123 get_access(handler& commandGroupHandler, range<Dimensions> accessRange,
124           id<Dimensions> accessOffset = {});
125
126 // Deprecated
127 template <access_mode Mode>
128 accessor<T, Dimensions, Mode, target::host_buffer>
129 get_access(range<Dimensions> accessRange, id<Dimensions> accessOffset = {});
130
131 template <typename... Ts> auto get_access(Ts...);
132
133 template <typename... Ts> auto get_host_access(Ts...);
134
135 template <typename Destination = std::nullptr_t>
136 void set_final_data(Destination finalData = nullptr);
137
138 void set_write_back(bool flag = true);
139
140 bool is_sub_buffer() const;
141
142 template <typename ReinterpretT, int ReinterpretDim>
143 buffer<ReinterpretT, ReinterpretDim,
144       typename std::allocator_traits<AllocatorT>::template rebind_alloc<
145       ReinterpretT>>
146 reinterpret(range<ReinterpretDim> reinterpretRange) const;
147
148 // Only available when ReinterpretDim == 1
149 // or when (ReinterpretDim == Dimensions) &&
150 //         (sizeof(ReinterpretT) == sizeof(T))
151 template <typename ReinterpretT, int ReinterpretDim = Dimensions>
152 buffer<ReinterpretT, ReinterpretDim,

```

```

153         typename std::allocator_traits<AllocatorT>::template rebind_alloc<
154             ReinterpretT>>
155     reinterpret() const;
156 };
157
158 // Deduction guides
159 template <class InputIterator, class AllocatorT>
160 buffer(InputIterator, InputIterator, AllocatorT, const property_list& = {})
161     -> buffer<typename std::iterator_traits<InputIterator>::value_type, 1,
162         AllocatorT>;
163
164 template <class InputIterator>
165 buffer(InputIterator, InputIterator, const property_list& = {})
166     -> buffer<typename std::iterator_traits<InputIterator>::value_type, 1>;
167
168 template <class T, int Dimensions, class AllocatorT>
169 buffer(const T*, const range<Dimensions>&, AllocatorT,
170     const property_list& = {}) -> buffer<T, Dimensions, AllocatorT>;
171
172 template <class T, int Dimensions>
173 buffer(const T*, const range<Dimensions>&, const property_list& = {})
174     -> buffer<T, Dimensions>;
175
176 template <class Container, class AllocatorT>
177 buffer(Container&, AllocatorT, const property_list& = {})
178     -> buffer<typename Container::value_type, 1, AllocatorT>;
179
180 template <class Container>
181 buffer(Container&, const property_list& = {})
182     -> buffer<typename Container::value_type, 1>;
183
184 } // namespace sycl

```

Table 21. Constructors of the `buffer` class

Constructor	Description
<pre>buffer(const range&lt;Dimensions&gt;&amp; bufferRange,       const property_list&amp; propList = {})</pre>	<p>Construct a SYCL <code>buffer</code> instance with uninitialized memory. The constructed SYCL <code>buffer</code> will use a default constructed <code>AllocatorT</code> when allocating memory on the host. The range of the constructed SYCL <code>buffer</code> is specified by the <code>bufferRange</code> parameter provided. Data is not written back to the host on destruction of the <code>buffer</code> unless the <code>buffer</code> has a valid non-null pointer specified via the member function <code>set_final_data()</code>. Zero or more properties can be provided to the constructed SYCL <code>buffer</code> via an instance of <code>property_list</code>.</p>

Constructor	Description
<pre>buffer(const range&lt;Dimensions&gt;&amp; bufferRange,         AllocatorT allocator,         const property_list&amp; propList = {})</pre>	<p>Construct a SYCL <b>buffer</b> instance with uninitialized memory. The constructed SYCL <b>buffer</b> will use the <b>allocator</b> parameter provided when allocating memory on the host. The range of the constructed SYCL <b>buffer</b> is specified by the <b>bufferRange</b> parameter provided. Data is not written back to the host on destruction of the <b>buffer</b> unless the <b>buffer</b> has a valid non-null pointer specified via the member function <b>set_final_data()</b>. Zero or more properties can be provided to the constructed SYCL <b>buffer</b> via an instance of <b>property_list</b>.</p>
<pre>buffer(T* hostData, const range&lt;Dimensions&gt;&amp; bufferRange,         const property_list&amp; propList = {})</pre>	<p>Construct a SYCL <b>buffer</b> instance with the <b>hostData</b> parameter provided. The buffer is initialized with the memory specified by <b>hostData</b>, and the buffer assumes exclusive access to this memory for the duration of its lifetime. The constructed SYCL <b>buffer</b> will use a default constructed <b>AllocatorT</b> when allocating memory on the host. The range of the constructed SYCL <b>buffer</b> is specified by the <b>bufferRange</b> parameter provided. Zero or more properties can be provided to the constructed SYCL <b>buffer</b> via an instance of <b>property_list</b>.</p>
<pre>buffer(T* hostData, const range&lt;Dimensions&gt;&amp; bufferRange,         AllocatorT allocator,         const property_list&amp; propList = {})</pre>	<p>Construct a SYCL <b>buffer</b> instance with the <b>hostData</b> parameter provided. The buffer is initialized with the memory specified by <b>hostData</b>, and the buffer assumes exclusive access to this memory for the duration of its lifetime. The constructed SYCL <b>buffer</b> will use the <b>allocator</b> parameter provided when allocating memory on the host. The range of the constructed SYCL <b>buffer</b> is specified by the <b>bufferRange</b> parameter provided. Zero or more properties can be provided to the constructed SYCL <b>buffer</b> via an instance of <b>property_list</b>.</p>

Constructor	Description
<pre data-bbox="161 185 751 293">buffer(const T* hostData,         const range&lt;Dimensions&gt;&amp; bufferRange,         const property_list&amp; propList = {})</pre>	<p data-bbox="1002 159 1458 338">Construct a SYCL <b>buffer</b> instance with the <b>hostData</b> parameter provided. The buffer assumes exclusive access to this memory for the duration of its lifetime.</p> <p data-bbox="1002 367 1458 510">The constructed SYCL <b>buffer</b> will use a default constructed <b>AllocatorT</b> when allocating memory on the host.</p> <p data-bbox="1002 539 1458 1003">The host address is <b>const T</b>, so the host accesses can be read-only. However, the <b>typename T</b> is not <b>const</b> so the device accesses can be both read and write accesses. Since the <b>hostData</b> is <b>const</b>, this buffer is only initialized with this memory and there is no write back after its destruction, unless the <b>buffer</b> has another valid non-null final data address specified via the member function <b>set_final_data()</b> after construction of the <b>buffer</b>.</p> <p data-bbox="1002 1032 1458 1140">The range of the constructed SYCL <b>buffer</b> is specified by the <b>buffer-Range</b> parameter provided.</p> <p data-bbox="1002 1169 1458 1305">Zero or more properties can be provided to the constructed SYCL <b>buffer</b> via an instance of <b>property_list</b>.</p>

Constructor	Description
<pre> buffer(const T* hostData,       const range&lt;Dimensions&gt;&amp; bufferRange,       AllocatorT allocator,       const property_list&amp; propList = {}) </pre>	<p>Construct a SYCL <b>buffer</b> instance with the <b>hostData</b> parameter provided. The buffer assumes exclusive access to this memory for the duration of its lifetime.</p> <p>The constructed SYCL <b>buffer</b> will use the <b>allocator</b> parameter provided when allocating memory on the host.</p> <p>The host address is <b>const T</b>, so the host accesses can be read-only. However, the <b>typename T</b> is not <b>const</b> so the device accesses can be both read and write accesses. Since, the <b>hostData</b> is <b>const</b>, this buffer is only initialized with this memory and there is no write back after its destruction, unless the <b>buffer</b> has another valid non-null final data address specified via the member function <b>set_final_data()</b> after construction of the <b>buffer</b>.</p> <p>The range of the constructed SYCL <b>buffer</b> is specified by the <b>buffer-Range</b> parameter provided.</p> <p>Zero or more properties can be provided to the constructed SYCL <b>buffer</b> via an instance of <b>property_list</b>.</p>



Constructor	Description
<pre>template &lt;typename Container&gt; buffer(Container&amp; container,         const property_list&amp; propList = {})</pre>	<p>Construct a one dimensional SYCL <b>buffer</b> instance from the elements starting at <code>std::data(container)</code> and containing <code>std::size(container)</code> number of elements. The buffer is initialized with the contents of <b>container</b>, and the buffer assumes exclusive access to <b>container</b> for the duration of its life-time.</p> <p>Data is written back to <b>container</b> before the completion of <b>buffer</b> destruction if the return type of <code>std::data(container)</code> is not <code>const</code>.</p> <p>The constructed SYCL <b>buffer</b> will use a default constructed <b>AllocatorT</b> when allocating memory on the host.</p> <p>Zero or more properties can be provided to the constructed SYCL <b>buffer</b> via an instance of <b>property_list</b>.</p> <p>This constructor is only defined for a <b>buffer</b> parameterized with <code>Dimensions == 1</code>, and when <code>std::data(container)</code> is convertible to <code>T*</code>.</p>

Constructor	Description
<pre data-bbox="161 185 831 293"> template &lt;typename Container&gt; buffer(Container&amp; container, AllocatorT allocator,         const property_list&amp; propList = {}) </pre>	<p data-bbox="1007 159 1460 517">Construct a one dimensional SYCL <b>buffer</b> instance from the elements starting at <code>std::data(container)</code> and containing <code>std::size(container)</code> number of elements. The buffer is initialized with the contents of <b>container</b>, and the buffer assumes exclusive access to <b>container</b> for the duration of its life-time.</p> <p data-bbox="1007 551 1460 685">Data is written back to <b>container</b> before the completion of <b>buffer</b> destruction if the return type of <code>std::data(container)</code> is not <code>const</code>.</p> <p data-bbox="1007 719 1460 853">The constructed SYCL <b>buffer</b> will use the <b>allocator</b> parameter provided when allocating memory on the host.</p> <p data-bbox="1007 887 1460 1021">Zero or more properties can be provided to the constructed SYCL <b>buffer</b> via an instance of <b>property_list</b>.</p> <p data-bbox="1007 1055 1460 1223">This constructor is only defined for a <b>buffer</b> parameterized with <code>Dimensions == 1</code>, and when <code>std::data(container)</code> is convertible to <code>T*</code>.</p>

Constructor	Description
<pre data-bbox="161 185 751 293">buffer(const std::shared_ptr&lt;T&gt;&amp; hostData,        const range&lt;Dimensions&gt;&amp; bufferRange,        const property_list&amp; propList = {})</pre>	<p data-bbox="1002 159 1460 622">When <code>hostData</code> is not empty, construct a SYCL buffer with the contents of its stored pointer. The buffer assumes exclusive access to this memory for the duration of its lifetime. The buffer also creates its own internal copy of the <code>shared_ptr</code> that shares ownership of the <code>hostData</code> memory, which means the application can safely release ownership of this <code>shared_ptr</code> when the constructor returns.</p> <p data-bbox="1002 656 1460 757">When <code>hostData</code> is empty, construct a SYCL buffer with uninitialized memory.</p> <p data-bbox="1002 790 1460 1144">The constructed SYCL <code>buffer</code> will use a default constructed <code>AllocatorT</code> when allocating memory on the host. The range of the constructed SYCL <code>buffer</code> is specified by the <code>bufferRange</code> parameter provided. Zero or more properties can be provided to the constructed SYCL <code>buffer</code> via an instance of <code>property_list</code>.</p>

Constructor	Description
<pre data-bbox="161 185 751 327">buffer(const std::shared_ptr&lt;T&gt;&amp; hostData,        const range&lt;Dimensions&gt;&amp; bufferRange,        AllocatorT allocator,        const property_list&amp; propList = {})</pre>	<p data-bbox="1003 159 1458 622">When <code>hostData</code> is not empty, construct a SYCL buffer with the contents of its stored pointer. The buffer assumes exclusive access to this memory for the duration of its lifetime. The buffer also creates its own internal copy of the <code>shared_ptr</code> that shares ownership of the <code>hostData</code> memory, which means the application can safely release ownership of this <code>shared_ptr</code> when the constructor returns.</p> <p data-bbox="1003 656 1458 757">When <code>hostData</code> is empty, construct a SYCL buffer with uninitialized memory.</p> <p data-bbox="1003 790 1458 1149">The constructed SYCL <code>buffer</code> will use the <code>allocator</code> parameter provided when allocating memory on the host. The range of the constructed SYCL <code>buffer</code> is specified by the <code>bufferRange</code> parameter provided. Zero or more properties can be provided to the constructed SYCL <code>buffer</code> via an instance of <code>property_list</code>.</p>

Constructor	Description
<pre>buffer(const std::shared_ptr&lt;T[]&gt;&amp; hostData,       const range&lt;Dimensions&gt;&amp; bufferRange,       const property_list&amp; propList = {})</pre>	<p>When <code>hostData</code> is not empty, construct a SYCL buffer with the contents of its stored pointer. The buffer assumes exclusive access to this memory for the duration of its lifetime. The buffer also creates its own internal copy of the <code>shared_ptr</code> that shares ownership of the <code>hostData</code> memory, which means the application can safely release ownership of this <code>shared_ptr</code> when the constructor returns.</p> <p>When <code>hostData</code> is empty, construct a SYCL buffer with uninitialized memory.</p> <p>The constructed SYCL <code>buffer</code> will use a default constructed <code>AllocatorT</code> when allocating memory on the host. The range of the constructed SYCL <code>buffer</code> is specified by the <code>bufferRange</code> parameter provided. Zero or more properties can be provided to the constructed SYCL <code>buffer</code> via an instance of <code>property_list</code>.</p>

Constructor	Description
<pre>buffer(const std::shared_ptr&lt;T[]&gt;&amp; hostData,       const range&lt;Dimensions&gt;&amp; bufferRange,       AllocatorT allocator,       const property_list&amp; propList = {})</pre>	<p>When <b>hostData</b> is not empty, construct a SYCL buffer with the contents of its stored pointer. The buffer assumes exclusive access to this memory for the duration of its lifetime. The buffer also creates its own internal copy of the <b>shared_ptr</b> that shares ownership of the <b>hostData</b> memory, which means the application can safely release ownership of this <b>shared_ptr</b> when the constructor returns.</p> <p>When <b>hostData</b> is empty, construct a SYCL buffer with uninitialized memory.</p> <p>The constructed SYCL <b>buffer</b> will use the <b>allocator</b> parameter provided when allocating memory on the host. The range of the constructed SYCL <b>buffer</b> is specified by the <b>bufferRange</b> parameter provided. Zero or more properties can be provided to the constructed SYCL <b>buffer</b> via an instance of <b>property_list</b>.</p>
<pre>template &lt;typename InputIterator&gt; buffer(InputIterator first, InputIterator last,       const property_list&amp; propList = {})</pre>	<p>Create a new allocated 1D buffer initialized from the given elements ranging from <b>first</b> up to one before <b>last</b>. The data is copied to an intermediate memory position by the runtime. Data is not written back to the same iterator set provided. However, if the <b>buffer</b> has a valid non-const iterator specified via the member function <b>set_final_data()</b>, data will be copied back to that iterator. The constructed SYCL <b>buffer</b> will use a default constructed <b>AllocatorT</b> when allocating memory on the host. Zero or more properties can be provided to the constructed SYCL <b>buffer</b> via an instance of <b>property_list</b>.</p>

Constructor	Description
<pre>template &lt;typename InputIterator&gt; buffer(InputIterator first, InputIterator last,        AllocatorT allocator = {},        const property_list&amp; propList = {})</pre>	<p>Create a new allocated 1D buffer initialized from the given elements ranging from <code>first</code> up to one before <code>last</code>. The data is copied to an intermediate memory position by the runtime. Data is not written back to the same iterator set provided. However, if the <code>buffer</code> has a valid non-const iterator specified via the member function <code>set_final_data()</code>, data will be copied back to that iterator. The constructed SYCL <code>buffer</code> will use the <code>allocator</code> parameter provided when allocating memory on the host. Zero or more properties can be provided to the constructed SYCL <code>buffer</code> via an instance of <code>property_list</code>.</p>

Constructor	Description
<pre data-bbox="164 185 831 253">buffer(buffer&amp; b, const id&lt;Dimensions&gt;&amp; baseIndex,         const range&lt;Dimensions&gt;&amp; subRange)</pre>	<p data-bbox="1005 159 1458 658">Create a new sub-buffer without allocation to have separate accessors later. <code>b</code> is the buffer with the real data, which must not be a sub-buffer. <code>baseIndex</code> specifies the origin of the sub-buffer inside the buffer <code>b</code>. <code>subRange</code> specifies the size of the sub-buffer. The sum of <code>baseIndex</code> and <code>subRange</code> in any dimension must not exceed the parent buffer (<code>b</code>) size (<code>bufferRange</code>) in that dimension, and an <code>exception</code> with the <code>errc::invalid</code> error code must be thrown if violated.</p> <p data-bbox="1005 692 1458 831">The offset and range specified by <code>baseIndex</code> and <code>subRange</code> together must represent a contiguous region of the original SYCL <code>buffer</code>.</p> <p data-bbox="1005 864 1458 1037">If a non-contiguous region of a buffer is requested when constructing a sub-buffer, then an <code>exception</code> with the <code>errc::invalid</code> error code must be thrown.</p> <p data-bbox="1005 1070 1458 1933">The origin (based on <code>baseIndex</code>) of the sub-buffer being constructed must be a multiple of the memory base address alignment of each SYCL <code>device</code> which accesses data from the buffer. This value is retrievable via the SYCL <code>device</code> class <code>info</code> query <code>info::device::mem_base_addr_align</code>. Violating this requirement causes the implementation to throw an <code>exception</code> with the <code>errc::invalid</code> error code from the <code>accessor</code> constructor (if the accessor is not a placeholder) or from <code>handler::require()</code> (if the accessor is a placeholder). If the accessor is bound to a <code>command group</code> with a secondary queue, the sub-buffer's alignment must be compatible with both the primary queue's device and the secondary queue's device, otherwise this exception is thrown.</p> <p data-bbox="1005 1966 1458 2063">Must throw an <code>exception</code> with the <code>errc::invalid</code> error code if <code>b</code> is a sub-buffer.</p>



Table 22. Member functions for the `buffer` class

Member function	Description
<code>range&lt;Dimensions&gt; get_range() const</code>	Return a range object representing the size of the buffer in terms of number of elements in each dimension as passed to the constructor.
<code>size_t size() const noexcept</code>	Returns the total number of elements in the buffer. Equal to <code>get_range()[0] * ... * get_range()[Dimensions-1]</code> .
<code>size_t get_count() const</code>	Returns the same value as <code>size()</code> . Deprecated.
<code>size_t byte_size() const noexcept</code>	Returns the size of the buffer storage in bytes. Equal to <code>size()*sizeof(T)</code> .
<code>size_t get_size() const</code>	Returns the same value as <code>byte_size()</code> . Deprecated.
<code>AllocatorT get_allocator() const</code>	Returns the allocator provided to the buffer.
<pre>template &lt;access_mode Mode = access_mode::read_write,           target Targ = target::device&gt; accessor&lt;T, Dimensions, Mode, Targ&gt; get_access(handler&amp; commandGroupHandler)</pre>	Returns a valid <code>accessor</code> to the buffer with the specified access mode and target in the command group buffer. The value of target can be <code>target::device</code> , <code>target::constant_buffer</code> or <code>target::host_task</code> .
<pre>template &lt;access_mode Mode&gt; accessor&lt;T, Dimensions, Mode, target::host_buffer&gt; get_access()</pre>	<p>Deprecated in SYCL 2020. Use <code>get_host_access()</code> instead.</p> <p>Returns a valid host <code>accessor</code> to the buffer with the specified access mode and target.</p>
<pre>template &lt;access_mode Mode = access_mode::read_write,           target Targ = target::device&gt; accessor&lt;T, Dimensions, Mode, Targ&gt; get_access(handler&amp; commandGroupHandler,   range &lt;Dimensions&gt; accessRange,   id &lt;Dimensions&gt; accessOffset = {})</pre>	<p>Returns a valid <code>accessor</code> to the buffer with the specified access mode and target in the command group buffer. The accessor is a <code>ranged accessor</code>, where the range starts at the given offset from the beginning of the buffer. The value of target can be <code>target::device</code>, <code>target::constant_buffer</code> or <code>target::host_task</code>.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if the sum of <code>accessRange</code> and <code>accessOffset</code> exceeds the range of the buffer in any dimension.</p>

Member function	Description
<pre>template &lt;access_mode Mode&gt; accessor&lt;T, Dimensions, Mode, target::host_buffer&gt; get_access(range&lt;Dimensions&gt; accessRange, id&lt;Dimensions&gt; accessOffset = {})</pre>	<p>Deprecated in SYCL 2020. Use <code>get_host_access()</code> instead.</p> <p>Returns a valid host <code>accessor</code> to the buffer with the specified access mode and target. The accessor is a <code>ranged accessor</code>, where the range starts at the given offset from the beginning of the buffer. The value of <code>target</code> can only be <code>target::host_buffer</code>.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if the sum of <code>accessRange</code> and <code>accessOffset</code> exceeds the range of the buffer in any dimension.</p>
<pre>template &lt;typename... Ts&gt; auto get_access(Ts... args)</pre>	<p>Returns a valid <code>accessor</code> as if constructed via passing the buffer and all provided arguments to the <code>accessor</code> constructor.</p> <p>Possible implementation:</p> <pre>return accessor{*this, args...};</pre>
<pre>template &lt;typename... Ts&gt; auto get_host_access(Ts... args)</pre>	<p>Returns a valid <code>host_accessor</code> as if constructed via passing the buffer and all provided arguments to the <code>host_accessor</code> constructor.</p> <p>Possible implementation:</p> <pre>return host_accessor{*this, args...};</pre>
<pre>template &lt;typename Destination = std::nullptr_t&gt; void set_final_data(Destination finalData = nullptr)</pre>	<p>The <code>finalData</code> points to where the outcome of all the buffer processing is going to be copied to at destruction time, if the buffer was involved with a write accessor.</p> <p>Destination can be either an output iterator or a <code>std::weak_ptr&lt;T&gt;</code>.</p> <p>Note that a raw pointer is a special case of output iterator and thus defines the host memory to which the result is to be copied.</p> <p>In the case of a weak pointer, the output is not updated if the weak pointer has expired.</p> <p>If <code>Destination</code> is <code>std::nullptr_t</code>, then the copy back will not happen.</p>

Member function	Description
<pre>void set_write_back(bool flag = true)</pre>	<p>This member function allows dynamically forcing or canceling the write-back of the data of a buffer on destruction according to the value of <code>flag</code>.</p> <p>Forcing the write-back is similar to what happens during a normal write-back as described in <a href="#">Section 4.7.2.3</a> and <a href="#">Section 4.7.4</a>.</p> <p>If there is nowhere to write-back, using this function does not have any effect.</p>
<pre>bool is_sub_buffer() const</pre>	<p>Returns true if this SYCL <code>buffer</code> is a sub-buffer, otherwise returns false.</p>
<pre>template &lt;typename ReinterpretT, int ReinterpretDim&gt; buffer&lt;ReinterpretT, ReinterpretDim,       typename std::allocator_traits&lt;AllocatorT&gt;::template rebind_alloc&lt;       std::remove_const_t&lt;ReinterpretT&gt;&gt;&gt; reinterpret(range&lt;ReinterpretDim&gt; reinterpretRange) const</pre>	<p>Creates and returns a reinterpreted SYCL <code>buffer</code> with the type specified by <code>ReinterpretT</code>, dimensions specified by <code>ReinterpretDim</code> and range specified by <code>reinterpretRange</code>. The buffer object being reinterpreted can be a SYCL sub-buffer that was created from a SYCL <code>buffer</code> and must throw <code>exception</code> with the <code>errc::invalid</code> error code if the total size in bytes represented by the type and range of the reinterpreted SYCL <code>buffer</code> (or sub-buffer) does not equal the total size in bytes represented by the type and range of this SYCL <code>buffer</code> (or sub-buffer). Reinterpreting a sub-buffer provides a reinterpreted view of the sub-buffer only, and does not change the offset or size of the sub-buffer view (in bytes) relative to the parent <code>buffer</code>.</p>

Member function	Description
<pre> template &lt;typename ReinterpretT, int ReinterpretDim = Dimensions&gt; buffer&lt;ReinterpretT, ReinterpretDim,       typename std::allocator_traits&lt;AllocatorT&gt;::template rebind_alloc&lt;       std::remove_const_t&lt;ReinterpretT&gt;&gt;&gt; reinterpret() const </pre>	<p>Creates and returns a reinterpreted SYCL <b>buffer</b> with the type specified by <b>ReinterpretT</b> and dimensions specified by <b>ReinterpretDim</b>. Only valid when (<b>ReinterpretDim</b> == 1) or when ((<b>ReinterpretDim</b> == <b>Dimensions</b>) &amp;&amp; (sizeof(<b>ReinterpretT</b>) == sizeof(<b>T</b>))). The buffer object being reinterpreted can be a SYCL sub-buffer that was created from a SYCL <b>buffer</b>. The implementation must throw an <b>exception</b> with the <b>errc::invalid</b> error code if the total size in bytes represented by this SYCL <b>buffer</b> (or sub-buffer) is not evenly divisible by <b>sizeof(ReinterpretT)</b>. Reinterpreting a sub-buffer provides a reinterpreted view of the sub-buffer only, and does not change the offset or size of the sub-buffer view (in bytes) relative to the parent <b>buffer</b>.</p>

#### 4.7.2.2. Buffer properties

The properties that can be provided when constructing the SYCL **buffer** class are described in [Table 23](#).

Table 23. Properties supported by the SYCL **buffer** class

Property	Description
<pre>property::buffer::use_host_ptr</pre>	<p>The <b>use_host_ptr</b> property adds the requirement that the <b>SYCL runtime</b> must not allocate any memory for the SYCL <b>buffer</b> and instead uses the provided host pointer directly. This prevents the <b>SYCL runtime</b> from allocating additional temporary storage on the host.</p> <p>This property has a special guarantee for buffers that are constructed from a <b>hostData</b> pointer. If a <b>host_accessor</b> is constructed from such a buffer, then the address of the <b>reference</b> type returned from the accessor's member functions such as <b>operator[](id&lt;&gt;)</b> will be the same as the corresponding <b>hostData</b> address.</p>

Property	Description
<code>property::buffer::use_mutex</code>	The <code>use_mutex</code> property is valid for the SYCL <code>buffer</code> , <code>unsampled_image</code> and <code>sampled_image</code> classes. The property adds the requirement that the memory which is owned by the SYCL <code>buffer</code> can be shared with the application via a <code>std::mutex</code> provided to the property. The mutex <code>m</code> is locked by the runtime whenever the data is in use and unlocked otherwise. The contents of <code>hostData</code> are guaranteed to reflect the contents of the buffer when the <code>std::mutex</code> is unlocked by the runtime.
<code>property::buffer::context_bound</code>	The <code>context_bound</code> property adds the requirement that the SYCL <code>buffer</code> can only be associated with a single SYCL <code>context</code> that is provided to the property.

The constructors and special member functions of the buffer property classes are listed in [Table 24](#) and [Table 25](#) respectively.

Table 24. Constructors of the `buffer` property classes

Constructor	Description
<code>property::buffer::use_host_ptr::use_host_ptr()</code>	Constructs a SYCL <code>use_host_ptr</code> property instance.
<code>property::buffer::use_mutex::use_mutex(std::mutex&amp; mutexRef)</code>	Constructs a SYCL <code>use_mutex</code> property instance with a reference to <code>mutexRef</code> parameter provided.
<code>property::buffer::context_bound::context_bound(context boundContext)</code>	Constructs a SYCL <code>context_bound</code> property instance with a copy of a SYCL <code>context</code> .

Table 25. Member functions of the `buffer` property classes

Member function	Description
<code>std::mutex* property::buffer::use_mutex::get_mutex_ptr() const</code>	Returns the <code>std::mutex</code> which was specified when constructing this SYCL <code>use_mutex</code> property.
<code>context property::buffer::context_bound::get_context() const</code>	Returns the <code>context</code> which was specified when constructing this SYCL <code>context_bound</code> property.

#### 4.7.2.3. Buffer destruction rules

Buffers are reference-counted. When a buffer value is constructed from another buffer, the two values reference the same buffer and a reference count is incremented. When a buffer value is destroyed, the

reference count is decremented. Only when there are no more buffer values that reference a specific buffer is the actual buffer destroyed and the buffer destruction behavior defined below is followed.

If any error occurs on buffer destruction, it is reported via the associated queue's asynchronous error handling mechanism.

The basic rule for the blocking behavior of a buffer destructor is that it blocks if there is some data to write back because a write accessor on it has been created, or if the buffer was constructed with attached host memory and is still in use.

More precisely:

1. A buffer can be constructed from a `range` (and without a `hostData` pointer). The memory management for this type of buffer is entirely handled by the SYCL system. The destructor for this type of buffer does not need to block, even if work on the buffer has not completed. Instead, the SYCL system frees any storage required for the buffer asynchronously when it is no longer in use in queues. The initial contents of the buffer are unspecified.
2. A buffer can be constructed from a `hostData` pointer. The buffer will use this host memory for its full lifetime, but the contents of this host memory are unspecified for the lifetime of the buffer. If the host memory is modified on the host or if it is used to construct another buffer or image during the lifetime of this buffer, then the results are undefined. The initial contents of the buffer will be the contents of the host memory at the time of construction.

When the buffer is destroyed, the destructor will block until all work in queues on the buffer have completed, then copy the contents of the buffer back to the host memory (if required) and then return.

- a. If the type of the host data is `const`, then the buffer is read-only; only read accessors are allowed on the buffer and no-copy-back to host memory is performed (although the host memory must still be kept available for use by SYCL). When using the default buffer allocator, the const-ness of the type will be removed in order to allow host allocation of memory, which will allow temporary host copies of the data by the [SYCL runtime](#), for example for speeding up host accesses.

When the buffer is destroyed, the destructor will block until all work in queues on the buffer have completed and then return, as there is no copy of data back to host.

- b. If the type of the host data is not `const` but the pointer to host data is `const`, then the read-only restriction applies only on host and not on device accesses.

When the buffer is destroyed, the destructor will block until all work in queues on the buffer have completed.

3. A buffer can be constructed using a `shared_ptr` to host data. This pointer is shared between the SYCL application and the runtime. In order to allow synchronization between the application and the runtime a `mutex` is used which will be locked by the runtime whenever the data is in use, and unlocked when it is no longer needed.

The `shared_ptr` reference counting is used in order to prevent destroying the buffer host data prematurely. If the `shared_ptr` is deleted from the user application before buffer destruction, the buffer can continue securely because the pointer hasn't been destroyed yet. It will not copy data back to the host before destruction, however, as the application side has already deleted its copy.

Note that since there is an implicit conversion of a `std::unique_ptr` to a `std::shared_ptr`, a `std::unique_ptr` can also be used to pass the ownership to the [SYCL runtime](#).

4. A buffer can be constructed from a pair of iterator values. In this case, the buffer construction will copy the data from the data range defined by the iterator pair. The destructor will not copy back any data and does not need to block.

5. A buffer can be constructed from a container on which `std::data(container)` and `std::size(container)` are well-formed. The initial contents of the buffer will be the contents of the container at the time of construction.

The buffer may use the memory within the container for its full lifetime, and the contents of this memory are unspecified for the lifetime of the buffer. If the container memory is modified by the host during the lifetime of this buffer, then the results are undefined.

When the buffer is destroyed, the destructor will block until all work in queues on the buffer have completed. If the return type of `std::data(container)` is not `const` then the destructor will also copy the contents of the buffer to the container (if required).

If `set_final_data()` is used to change where to write the data back to, then the destructor of the buffer will block if a write accessor on it has been created.

A sub-buffer object can be created which is a sub-range reference to a base buffer. This sub-buffer can be used to create accessors to the base buffer, which have access to the range specified at time of construction of the sub-buffer. Sub-buffers cannot be created from sub-buffers, but only from a base buffer which is not already a sub-buffer.

Sub-buffers must be constructed from a contiguous region of memory in a buffer. This requirement is potentially non-intuitive when working with buffers that have dimensionality larger than one, but maps to one-dimensional SYCL backend native allocations without performance cost due to index mapping computation. For example:

```

1  buffer<int, 2> parent_buffer{
2      range<2>{8, 8}}; // Create 2-d buffer with 8x8 ints
3
4  // OK: Contiguous region from middle of buffer
5  buffer<int, 2> sub_buf1{parent_buffer, /*offset*/ range<2>{2, 0},
6                          /*size*/ range<2>{2, 8}};
7
8  // invalid exception: Non-contiguous regions of 2-d buffer
9  buffer<int, 2> sub_buf2{parent_buffer, /*offset*/ range<2>{2, 0},
10                         /*size*/ range<2>{2, 2}};
11 buffer<int, 2> sub_buf3{parent_buffer, /*offset*/ range<2>{2, 2},
12                         /*size*/ range<2>{2, 6}};
13
14 // invalid exception: Out-of-bounds size
15 buffer<int, 2> sub_buf4{parent_buffer, /*offset*/ range<2>{2, 2},
16                         /*size*/ range<2>{2, 8}};

```

### 4.7.3. Images

The classes `unsampled_image` (Table 26) and `sampled_image` (Table 28) define shared image data of one, two or three dimensions, that can be used by kernels in queues and have to be accessed using the image `accessor` classes.

The constructors and member functions of the SYCL `unsampled_image` and `sampled_image` class templates are listed in Table 26, Table 27, Table 28 and Table 29, respectively. The additional common special member functions and common member functions are listed in Table 7 and Table 8, respectively.

Where relevant, it is the responsibility of the user to ensure that the format of the data matches the format described by `image_format`.

The allocator template parameter of the SYCL `unsampled_image` and `sampled_image` classes can be any allo-



cator type including a custom allocator, however it must allocate in units of `std::byte`.

For any image that is constructed with the range ( $r1$ ,  $r2$ ,  $r3$ ) with an element type size in bytes of  $s$ , the image row pitch and image slice pitch should be calculated as follows:

$$r1 \cdot s$$

$$r1 \cdot r2 \cdot s$$

The SYCL `unsampled_image` and `sampled_image` class templates provide the common reference semantics (see [Section 4.5.2](#)).

#### 4.7.3.1. Unsampled image interface

Each constructor of the `unsampled_image` takes an `image_format` to describe the data layout of the image data.

Each constructor additionally takes as the last parameter an optional SYCL `property_list` to provide properties to the SYCL `unsampled_image`.

The SYCL `unsampled_image` class template takes a template parameter `AllocatorT` for specifying an allocator which is used by the SYCL runtime when allocating temporary memory on the host. If no template argument is provided, the default allocator for the SYCL `unsampled_image` class `image_allocator` is used (see [Section 4.7.1.1](#)).

```

1 namespace sycl {
2
3 enum class image_format : /* unspecified */ {
4     r8g8b8a8_unorm,
5     r16g16b16a16_unorm,
6     r8g8b8a8_sint,
7     r16g16b16a16_sint,
8     r32b32g32a32_sint,
9     r8g8b8a8_uint,
10    r16g16b16a16_uint,
11    r32b32g32a32_uint,
12    r16b16g16a16_sfloat,
13    r32g32b32a32_sfloat,
14    b8g8r8a8_unorm
15 };
16
17 template <int Dimensions = 1, typename AllocatorT = sycl::image_allocator>
18 class unsampled_image {
19 public:
20     unsampled_image(image_format format, const range<Dimensions>& rangeRef,
21                     const property_list& propList = {});
22
23     unsampled_image(image_format format, const range<Dimensions>& rangeRef,
24                     AllocatorT allocator, const property_list& propList = {});
25
26     /* Available only when: Dimensions > 1 */
27     unsampled_image(image_format format, const range<Dimensions>& rangeRef,
28                     const range<Dimensions - 1>& pitch,
29                     const property_list& propList = {});
30
31     /* Available only when: Dimensions > 1 */
32     unsampled_image(image_format format, const range<Dimensions>& rangeRef,
33                     const range<Dimensions - 1>& pitch, AllocatorT allocator,
```



```

34         const property_list& propList = {}));
35
36     unsampled_image(void* hostPointer, image_format format,
37         const range<Dimensions>& rangeRef,
38         const property_list& propList = {}));
39
40     unsampled_image(void* hostPointer, image_format format,
41         const range<Dimensions>& rangeRef, AllocatorT allocator,
42         const property_list& propList = {}));
43
44     /* Available only when: Dimensions > 1 */
45     unsampled_image(void* hostPointer, image_format format,
46         const range<Dimensions>& rangeRef,
47         const range<Dimensions - 1>& pitch,
48         const property_list& propList = {}));
49
50     /* Available only when: Dimensions > 1 */
51     unsampled_image(void* hostPointer, image_format format,
52         const range<Dimensions>& rangeRef,
53         const range<Dimensions - 1>& pitch, AllocatorT allocator,
54         const property_list& propList = {}));
55
56     unsampled_image(std::shared_ptr<void>& hostPointer, image_format format,
57         const range<Dimensions>& rangeRef,
58         const property_list& propList = {}));
59
60     unsampled_image(std::shared_ptr<void>& hostPointer, image_format format,
61         const range<Dimensions>& rangeRef, AllocatorT allocator,
62         const property_list& propList = {}));
63
64     /* Available only when: Dimensions > 1 */
65     unsampled_image(std::shared_ptr<void>& hostPointer, image_format format,
66         const range<Dimensions>& rangeRef,
67         const range<Dimensions - 1>& pitch,
68         const property_list& propList = {}));
69
70     /* Available only when: Dimensions > 1 */
71     unsampled_image(std::shared_ptr<void>& hostPointer, image_format format,
72         const range<Dimensions>& rangeRef,
73         const range<Dimensions - 1>& pitch, AllocatorT allocator,
74         const property_list& propList = {}));
75
76     /* -- common interface members -- */
77
78     /* -- property interface members -- */
79
80     range<Dimensions> get_range() const;
81
82     /* Available only when: Dimensions > 1 */
83     range<Dimensions - 1> get_pitch() const;
84
85     size_t byte_size() const noexcept;
86
87     size_t size() const noexcept;
88
89     AllocatorT get_allocator() const;

```

```

90
91  template <typename DataT,
92           access_mode Mode = (std::is_const_v<DataT>
93                               ? access_mode::read
94                               : access_mode::read_write),
95           image_target Targ = image_target::device>
96  unsampled_image_accessor<DataT, Dimensions, Mode, Targ>
97  get_access(handler& commandGroupHandler, const property_list& propList = {});
98
99  template <typename DataT, access_mode Mode = (std::is_const_v<DataT>
100                                                ? access_mode::read
101                                                : access_mode::read_write)>
102  host_unsampled_image_accessor<DataT, Dimensions, Mode>
103  get_host_access(const property_list& propList = {});
104
105  template <typename Destination = std::nullptr_t>
106  void set_final_data(Destination finalData = nullptr);
107
108  void set_write_back(bool flag = true);
109 };
110
111 } // namespace sycl

```

Table 26. Constructors of the `unsampled_image` class template

Constructor	Description
<pre> unsampled_image(image_format format,                 const range&lt;Dimensions&gt;&amp; rangeRef,                 const property_list&amp; propList = {}) </pre>	<p>Construct a SYCL <code>unsampled_image</code> instance with uninitialized memory. The constructed SYCL <code>unsampled_image</code> will use a default constructed <code>AllocatorT</code> when allocating memory on the host. The element size of the constructed SYCL <code>unsampled_image</code> will be derived from the <code>format</code> parameter. The range of the constructed SYCL <code>unsampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>unsampled_image</code> will be the default size determined by the <a href="#">SYCL runtime</a>. Unless the member function <code>set_final_data()</code> is called with a valid non-null pointer, there will be no write back on destruction. Zero or more properties can be provided to the constructed SYCL <code>unsampled_image</code> via an instance of <code>property_list</code>.</p>

Constructor	Description
<pre> unsampled_image(image_format format,                  const range&lt;Dimensions&gt;&amp; rangeRef,                  AllocatorT allocator,                  const property_list&amp; proplist = {}) </pre>	<p>Construct a SYCL <code>unsampled_image</code> instance with uninitialized memory. The constructed SYCL <code>unsampled_image</code> will use the <code>allocator</code> parameter provided when allocating memory on the host. The element size of the constructed SYCL <code>unsampled_image</code> will be derived from the <code>format</code> parameter. The range of the constructed SYCL <code>unsampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>unsampled_image</code> will be the default size determined by the SYCL runtime. Unless the member function <code>set_final_data()</code> is called with a valid non-null pointer, there will be no write back on destruction. Zero or more properties can be provided to the constructed SYCL <code>unsampled_image</code> via an instance of <code>property_list</code>.</p>
<pre> unsampled_image(image_format format,                  const range&lt;Dimensions&gt;&amp; rangeRef,                  const range&lt;Dimensions - 1&gt;&amp; pitch,                  const property_list&amp; proplist = {}) </pre>	<p>Available only when: <code>Dimensions &gt; 1</code>.</p> <p>Construct a SYCL <code>unsampled_image</code> instance with uninitialized memory. The constructed SYCL <code>unsampled_image</code> will use a default constructed <code>AllocatorT</code> when allocating memory on the host. The element size of the constructed SYCL <code>unsampled_image</code> will be derived from the <code>format</code> parameter. The range of the constructed SYCL <code>unsampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>unsampled_image</code> will be the <code>pitch</code> parameter provided. Unless the member function <code>set_final_data()</code> is called with a valid non-null pointer, there will be no write back on destruction. Zero or more properties can be provided to the constructed SYCL <code>unsampled_image</code> via an instance of <code>property_list</code>.</p>

Constructor	Description
<pre> unsampled_image(image_format format,                  const range&lt;Dimensions&gt;&amp; rangeRef,                  const range&lt;Dimensions - 1&gt;&amp; pitch,                  AllocatorT allocator,                  const property_list&amp; propList = {}) </pre>	<p>Available only when: <code>Dimensions &gt; 1</code>.</p> <p>Construct a SYCL <code>unsampled_image</code> instance with uninitialized memory. The constructed SYCL <code>unsampled_image</code> will use the <code>allocator</code> parameter provided when allocating memory on the host. The element size of the constructed SYCL <code>unsampled_image</code> will be derived from the <code>format</code> parameter. The range of the constructed SYCL <code>unsampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>unsampled_image</code> will be the <code>pitch</code> parameter provided. Unless the member function <code>set_final_data()</code> is called with a valid non-null pointer, there will be no write back on destruction. Zero or more properties can be provided to the constructed SYCL <code>unsampled_image</code> via an instance of <code>property_list</code>.</p>
<pre> unsampled_image(void* hostPointer, image_format format,                  const range&lt;Dimensions&gt;&amp; rangeRef,                  const property_list&amp; propList = {}) </pre>	<p>Construct a SYCL <code>unsampled_image</code> instance with the <code>hostPointer</code> parameter provided. The <code>unsampled_image</code> assumes exclusive access to this memory for the duration of its lifetime. The constructed SYCL <code>unsampled_image</code> will use a default constructed <code>AllocatorT</code> when allocating memory on the host. The element size of the constructed SYCL <code>unsampled_image</code> will be derived from the <code>format</code> parameter. The range of the constructed SYCL <code>unsampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>unsampled_image</code> will be the default size determined by the <code>SYCL runtime</code>. Unless the member function <code>set_final_data()</code> is called with a valid non-null pointer, any memory allocated by the <code>SYCL runtime</code> is written back to <code>hostPointer</code>. Zero or more properties can be provided to the constructed SYCL <code>unsampled_image</code> via an instance of <code>property_list</code>.</p>

Constructor	Description
<pre data-bbox="164 185 900 324"> unsampled_image(void* hostPointer, image_format format,                 const range&lt;Dimensions&gt;&amp; rangeRef,                 AllocatorT allocator,                 const property_list&amp; proplist = {}) </pre>	<p data-bbox="1002 159 1460 1126">Construct a SYCL <code>unsampled_image</code> instance with the <code>hostPointer</code> parameter provided. The <code>unsampled_image</code> assumes exclusive access to this memory for the duration of its lifetime. The constructed SYCL <code>unsampled_image</code> will use the <code>allocator</code> parameter provided when allocating memory on the host. The element size of the constructed SYCL <code>unsampled_image</code> will be derived from the <code>format</code> parameter. The range of the constructed SYCL <code>unsampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>unsampled_image</code> will be the default size determined by the <code>SYCL runtime</code>. Unless the member function <code>set_final_data()</code> is called with a valid non-null pointer, any memory allocated by the <code>SYCL runtime</code> is written back to <code>hostPointer</code>. Zero or more properties can be provided to the constructed SYCL <code>unsampled_image</code> via an instance of <code>property_list</code>.</p>

Constructor	Description
<pre data-bbox="161 185 900 327"> unsampled_image(void* hostPointer, image_format format,                  const range&lt;Dimensions&gt;&amp; rangeRef,                  const range&lt;Dimensions - 1&gt;&amp; pitch,                  const property_list&amp; proplist = {}) </pre>	<p data-bbox="1002 159 1457 219">Available only when: <code>Dimensions &gt; 1</code></p> <p data-bbox="1002 248 1457 1184">Construct a SYCL <code>unsampled_image</code> instance with the <code>hostPointer</code> parameter provided. The <code>unsampled_image</code> assumes exclusive access to this memory for the duration of its lifetime. The constructed SYCL <code>unsampled_image</code> will use a default constructed <code>AllocatorT</code> when allocating memory on the host. The element size of the constructed SYCL <code>unsampled_image</code> will be derived from the <code>format</code> parameter. The range of the constructed SYCL <code>unsampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>unsampled_image</code> will be the <code>pitch</code> parameter provided. Unless the member function <code>set_final_data()</code> is called with a valid non-null pointer, any memory allocated by the SYCL runtime is written back to <code>hostPointer</code>. Zero or more properties can be provided to the constructed SYCL <code>unsampled_image</code> via an instance of <code>property_list</code>.</p>

Constructor	Description
<pre data-bbox="161 185 900 360"> unsampled_image(void* hostPointer, image_format format,                  const range&lt;Dimensions&gt;&amp; rangeRef,                  const range&lt;Dimensions - 1&gt;&amp; pitch,                  AllocatorT allocator,                  const property_list&amp; proplist = {}) </pre>	<p data-bbox="1002 159 1457 226">Available only when: <code>Dimensions &gt; 1</code>.</p> <p data-bbox="1002 259 1457 1189">Construct a SYCL <code>unsampled_image</code> instance with the <code>hostPointer</code> parameter provided. The <code>unsampled_image</code> assumes exclusive access to this memory for the duration of its lifetime. The constructed SYCL <code>unsampled_image</code> will use the <code>allocator</code> parameter provided when allocating memory on the host. The element size of the constructed SYCL <code>unsampled_image</code> will be derived from the <code>format</code> parameter. The range of the constructed SYCL <code>unsampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>unsampled_image</code> will be the <code>pitch</code> parameter provided. Unless the member function <code>set_final_data()</code> is called with a valid non-null pointer, any memory allocated by the SYCL runtime is written back to <code>hostPointer</code>. Zero or more properties can be provided to the constructed SYCL <code>unsampled_image</code> via an instance of <code>property_list</code>.</p>

Constructor	Description
<pre data-bbox="161 185 847 327"> unsampled_image(std::shared_ptr&lt;void&gt;&amp; hostPointer,                  image_format format,                  const range&lt;Dimensions&gt;&amp; rangeRef,                  const property_list&amp; proplist = {}) </pre>	<p data-bbox="1002 159 1458 658">When <code>hostPointer</code> is not empty, construct a SYCL <code>unsampled_image</code> with the contents of its stored pointer. The <code>unsampled_image</code> assumes exclusive access to this memory for the duration of its life-time. The <code>unsampled_image</code> also creates its own internal copy of the <code>shared_ptr</code> that shares ownership of the <code>hostData</code> memory, which means the application can safely release ownership of this <code>shared_ptr</code> when the constructor returns.</p> <p data-bbox="1002 689 1458 797">When <code>hostPointer</code> is empty, construct a SYCL <code>unsampled_image</code> with uninitialized memory.</p> <p data-bbox="1002 828 1458 1619">The constructed SYCL <code>unsampled_image</code> will use a default constructed <code>AllocatorT</code> when allocating memory on the host. The element size of the constructed SYCL <code>unsampled_image</code> will be derived from the <code>format</code> parameter. The range of the constructed SYCL <code>unsampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>unsampled_image</code> will be the default size determined by the SYCL runtime. Unless the member function <code>set_final_data()</code> is called with a valid non-null pointer, any memory allocated by the SYCL runtime is written back to <code>hostPointer</code>. Zero or more properties can be provided to the constructed SYCL <code>unsampled_image</code> via an instance of <code>property_list</code>.</p>



Constructor	Description
<pre data-bbox="164 185 847 360"> unsampled_image(std::shared_ptr&lt;void&gt;&amp; hostPointer,                  image_format format,                  const range&lt;Dimensions&gt;&amp; rangeRef,                  AllocatorT allocator,                  const property_list&amp; proplist = {}) </pre>	<p data-bbox="1002 159 1460 658">When <code>hostPointer</code> is not empty, construct a SYCL <code>unsampled_image</code> with the contents of its stored pointer. The <code>unsampled_image</code> assumes exclusive access to this memory for the duration of its lifetime. The <code>unsampled_image</code> also creates its own internal copy of the <code>shared_ptr</code> that shares ownership of the <code>hostData</code> memory, which means the application can safely release ownership of this <code>shared_ptr</code> when the constructor returns.</p> <p data-bbox="1002 689 1460 792">When <code>hostPointer</code> is empty, construct a SYCL <code>unsampled_image</code> with uninitialized memory.</p> <p data-bbox="1002 824 1460 1615">The constructed SYCL <code>unsampled_image</code> will use the <code>allocator</code> parameter provided when allocating memory on the host. The element size of the constructed SYCL <code>unsampled_image</code> will be derived from the <code>format</code> parameter. The range of the constructed SYCL <code>unsampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>unsampled_image</code> will be the default size determined by the SYCL runtime. Unless the member function <code>set_final_data()</code> is called with a valid non-null pointer, any memory allocated by the SYCL runtime is written back to <code>hostPointer</code>. Zero or more properties can be provided to the constructed SYCL <code>unsampled_image</code> via an instance of <code>property_list</code>.</p>

Constructor	Description
<pre data-bbox="161 185 847 360"> unsampled_image(std::shared_ptr&lt;void&gt;&amp; hostPointer,                  image_format format,                  const range&lt;Dimensions&gt;&amp; rangeRef,                  const range&lt;Dimensions - 1&gt;&amp; pitch,                  const property_list&amp; proplist = {}) </pre>	<p data-bbox="1002 159 1458 658">When <code>hostPointer</code> is not empty, construct a SYCL <code>unsampled_image</code> with the contents of its stored pointer. The <code>unsampled_image</code> assumes exclusive access to this memory for the duration of its lifetime. The <code>unsampled_image</code> also creates its own internal copy of the <code>shared_ptr</code> that shares ownership of the <code>hostData</code> memory, which means the application can safely release ownership of this <code>shared_ptr</code> when the constructor returns.</p> <p data-bbox="1002 692 1458 797">When <code>hostPointer</code> is empty, construct a SYCL <code>unsampled_image</code> with uninitialized memory.</p> <p data-bbox="1002 831 1458 1576">The constructed SYCL <code>unsampled_image</code> will use a default constructed <code>AllocatorT</code> when allocating memory on the host. The element size of the constructed SYCL <code>unsampled_image</code> will be derived from the <code>format</code> parameter. The range of the constructed SYCL <code>unsampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>unsampled_image</code> will be the <code>pitch</code> parameter provided. Unless the member function <code>set_final_data()</code> is called with a valid non-null pointer, any memory allocated by the SYCL runtime is written back to <code>hostPointer</code>. Zero or more properties can be provided to the constructed SYCL <code>unsampled_image</code> via an instance of <code>property_list</code>.</p>

Constructor	Description
<pre> unsampled_image(std::shared_ptr&lt;void&gt;&amp; hostPointer,                 image_format format,                 const range&lt;Dimensions&gt;&amp; rangeRef,                 const range&lt;Dimensions - 1&gt;&amp; pitch,                 AllocatorT allocator,                 const property_list&amp; propList = {}) </pre>	<p>When <code>hostPointer</code> is not empty, construct a SYCL <code>unsampled_image</code> with the contents of its stored pointer. The <code>unsampled_image</code> assumes exclusive access to this memory for the duration of its lifetime. The <code>unsampled_image</code> also creates its own internal copy of the <code>shared_ptr</code> that shares ownership of the <code>hostData</code> memory, which means the application can safely release ownership of this <code>shared_ptr</code> when the constructor returns.</p> <p>When <code>hostPointer</code> is empty, construct a SYCL <code>unsampled_image</code> with uninitialized memory.</p> <p>The constructed SYCL <code>unsampled_image</code> will use the <code>allocator</code> parameter provided when allocating memory on the host. The element size of the constructed SYCL <code>unsampled_image</code> will be derived from the <code>format</code> parameter. The range of the constructed SYCL <code>unsampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>unsampled_image</code> will be the <code>pitch</code> parameter provided. Unless the member function <code>set_final_data()</code> is called with a valid non-null pointer, any memory allocated by the SYCL runtime is written back to <code>hostPointer</code>. Zero or more properties can be provided to the constructed SYCL <code>unsampled_image</code> via an instance of <code>property_list</code>.</p>

Table 27. Member functions of the `unsampled_image` class template

Member function	Description
<pre> range&lt;Dimensions&gt; get_range() const </pre>	Return a range object representing the size of the image in terms of the number of elements in each dimension as passed to the constructor.
<pre> range&lt;Dimensions - 1&gt; get_pitch() const </pre>	<p>Available only when: <code>Dimensions &gt; 1</code>.</p> <p>Return a range object representing the pitch of the image in bytes.</p>

Member function	Description
<code>size_t size() const noexcept</code>	Returns the total number of elements in the image. Equal to <code>get_range()[0] * ... * get_range()[Dimensions-1]</code> .
<code>size_t byte_size() const noexcept</code>	Returns the size of the image storage in bytes. The number of bytes may be greater than <code>size()*element_size</code> due to padding of elements, rows and slices of the image for efficient access.
<code>AllocatorT get_allocator() const</code>	Returns the allocator provided to the image.
<pre>template &lt;typename DataT,           access_mode Mode = (std::is_const_v&lt;DataT&gt;                               ? access_mode::read                               : access_mode:: read_write),           image_target Targ = image_target::device&gt; unsampled_image_accessor&lt;DataT, Dimensions, Mode, Targ&gt; get_access(handler&amp; commandGroupHandler)</pre>	Returns a valid <code>unsampled_image_accessor</code> to the unsampled image with the specified data type, access mode and target in the command group.
<pre>template &lt;typename DataT, access_mode Mode = (std ::is_const_v&lt;DataT&gt;                                      ? access_mode::read                                      : access_mode::read_write)&gt; host_unsampled_image_accessor&lt;DataT, Dimensions, Mode&gt; get_host_access();</pre>	Returns a valid <code>host_unsampled_image_accessor</code> to the unsampled image with the specified data type and access mode.
<pre>template &lt;typename Destination = std::nullptr_t&gt; void set_final_data(Destination finalData = nullptr)</pre>	<p>The <code>finalData</code> point to where the output of all the image processing is going to be copied to at destruction time, if the image was involved with a write accessor.</p> <p>Destination can be either an output iterator, or a <code>std::weak_ptr&lt;T&gt;</code>.</p> <p>Note that a raw pointer is a special case of output iterator and thus defines the host memory to which the result is to be copied.</p> <p>In the case of a weak pointer, the output is not copied if the weak pointer has expired.</p> <p>If <code>Destination</code> is <code>std::nullptr_t</code>, then the copy back will not happen.</p>

Member function	Description
<pre>void set_write_back(bool flag = true)</pre>	<p>This member function allows dynamically forcing or canceling the write-back of the data of an image on destruction according to the value of <code>flag</code>.</p> <p>Forcing the write-back is similar to what happens during a normal write-back as described in <a href="#">Section 4.7.3.4</a> and <a href="#">Section 4.7.4</a>.</p> <p>If there is nowhere to write-back, using this function does not have any effect.</p>

### 4.7.3.2. Sampled image interface

Each constructor of the `sampled_image` class requires a pointer to the host data the image will sample, an `image_format` to describe the data layout and an `image_sampler` ([Section 4.7.8](#)) to describe how to sample the image data.

Each constructor additionally takes as the last parameter an optional SYCL `property_list` to provide properties to the SYCL `sampled_image`.

```

1 namespace sycl {
2
3 enum class image_format : /* unspecified */ {
4     r8g8b8a8_unorm,
5     r16g16b16a16_unorm,
6     r8g8b8a8_sint,
7     r16g16b16a16_sint,
8     r32b32g32a32_sint,
9     r8g8b8a8_uint,
10    r16g16b16a16_uint,
11    r32b32g32a32_uint,
12    r16b16g16a16_sfloat,
13    r32g32b32a32_sfloat,
14    b8g8r8a8_unorm
15 };
16
17 template <int Dimensions = 1, typename AllocatorT = sycl::image_allocator>
18 class sampled_image {
19 public:
20     sampled_image(const void* hostPointer, image_format format,
21                 image_sampler sampler, const range<Dimensions>& rangeRef,
22                 const property_list& proplist = {});
23
24     /* Available only when: Dimensions > 1 */
25     sampled_image(const void* hostPointer, image_format format,
26                 image_sampler sampler, const range<Dimensions>& rangeRef,
27                 const range<Dimensions - 1>& pitch,
28                 const property_list& proplist = {});
29
30     sampled_image(std::shared_ptr<const void>& hostPointer, image_format format,
31                 image_sampler sampler, const range<Dimensions>& rangeRef,
```

```

32         const property_list& propList = {});
33
34     /* Available only when: Dimensions > 1 */
35     sampled_image(std::shared_ptr<const void>& hostPointer, image_format format,
36                 image_sampler sampler, const range<Dimensions>& rangeRef,
37                 const range<Dimensions - 1>& pitch,
38                 const property_list& propList = {});
39
40     /* -- common interface members -- */
41
42     /* -- property interface members -- */
43
44     range<Dimensions> get_range() const;
45
46     /* Available only when: Dimensions > 1 */
47     range<Dimensions - 1> get_pitch() const;
48
49     size_t byte_size() const noexcept;
50
51     size_t size() const noexcept;
52
53     template <typename DataT, image_target Targ = image_target::device>
54     sampled_image_accessor<DataT, Dimensions, Targ>
55     get_access(handler& commandGroupHandler, const property_list& propList = {});
56
57     template <typename DataT>
58     host_sampled_image_accessor<DataT, Dimensions>
59     get_host_access(const property_list& propList = {});
60 };
61
62 } // namespace sycl

```

Table 28. Constructors of the `sampled_image` class template

Constructor	Description
<pre data-bbox="161 185 954 327">sampled_image(const void* hostPointer, image_format format,                image_sampler sampler,                const range&lt;Dimensions&gt;&amp; rangeRef,                const property_list&amp; propList = {})</pre>	<p data-bbox="1002 159 1460 1135">Construct a SYCL <code>sampled_image</code> instance with the <code>hostPointer</code> parameter provided. The <code>sampled_image</code> assumes exclusive access to this memory for the duration of its lifetime. The host address is <code>const</code>, so the host accesses must be read-only. Since, the <code>hostPointer</code> is <code>const</code>, this image is only initialized with this memory and there is no write after its destruction. The element size of the constructed SYCL <code>sampled_image</code> will be derived from the <code>format</code> parameter. Accessors that read the constructed SYCL <code>sampled_image</code> will use the <code>sampler</code> parameter to sample the image. The range of the constructed SYCL <code>sampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>sampled_image</code> will be the default size determined by the SYCL runtime. Zero or more properties can be provided to the constructed SYCL <code>sampled_image</code> via an instance of <code>property_list</code>.</p>

Constructor	Description
<pre data-bbox="161 185 954 360">sampled_image(const void* hostPointer, image_format format,               image_sampler sampler,               const range&lt;Dimensions&gt;&amp; rangeRef,               const range&lt;Dimensions - 1&gt;&amp; pitch,               const property_list&amp; propList = {})</pre>	<p data-bbox="1002 159 1457 226">Available only when: <code>Dimensions &gt; 1</code>.</p> <p data-bbox="1002 259 1457 1189">Construct a SYCL <code>sampled_image</code> instance with the <code>hostPointer</code> parameter provided. The <code>sampled_image</code> assumes exclusive access to this memory for the duration of its lifetime. The host address is <code>const</code>, so the host accesses must be read-only. Since, the <code>hostPointer</code> is <code>const</code>, this image is only initialized with this memory and there is no write after destruction. The element size of the constructed SYCL <code>sampled_image</code> will be derived from the <code>format</code> parameter. Accessors that read the constructed SYCL <code>sampled_image</code> will use the <code>sampler</code> parameter to sample the image. The range of the constructed SYCL <code>sampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>sampled_image</code> will be the <code>pitch</code> parameter provided. Zero or more properties can be provided to the constructed SYCL <code>sampled_image</code> via an instance of <code>property_list</code>.</p>



Constructor	Description
<pre data-bbox="164 185 900 360">sampled_image(std::shared_ptr&lt;const void&gt;&amp; hostPointer,                image_format format,                image_sampler sampler,                const range&lt;Dimensions&gt;&amp; rangeRef,                const property_list&amp; propList = {})</pre>	<p data-bbox="1005 159 1458 622">When <code>hostPointer</code> is not empty, construct a SYCL <code>sampled_image</code> with the contents of its stored pointer. The <code>sampled_image</code> assumes exclusive access to this memory for the duration of its lifetime. The <code>sampled_image</code> also creates its own internal copy of the <code>shared_ptr</code> that shares ownership of the <code>hostData</code> memory, which means the application can safely release ownership of this <code>shared_ptr</code> when the constructor returns.</p> <p data-bbox="1005 656 1458 757">When <code>hostPointer</code> is empty, construct a SYCL <code>sampled_image</code> with uninitialized memory.</p> <p data-bbox="1005 790 1458 1541">The host address is <code>const</code>, so the host accesses must be read-only. Since, the <code>hostPointer</code> is <code>const</code>, this image is only initialized with this memory and there is no write after its destruction. The element size of the constructed SYCL <code>sampled_image</code> will be derived from the <code>format</code> parameter. Accessors that read the constructed SYCL <code>sampled_image</code> will use the <code>sampler</code> parameter to sample the image. The range of the constructed SYCL <code>sampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>sampled_image</code> will be the default size determined by the <a href="#">SYCL runtime</a>. Zero or more properties can be provided to the constructed SYCL <code>sampled_image</code> via an instance of <code>property_list</code>.</p>

Constructor	Description
<pre>sampld_image(std::shared_ptr&lt;const void&gt;&amp; hostPointer,               image_format format,               image_sampler sampler,               const range&lt;Dimensions&gt;&amp; rangeRef,               const range&lt;Dimensions - 1&gt;&amp; pitch,               const property_list&amp; propList = {})</pre>	<p>When <code>hostPointer</code> is not empty, construct a SYCL <code>sampld_image</code> with the contents of its stored pointer. The <code>sampld_image</code> assumes exclusive access to this memory for the duration of its lifetime. The <code>sampld_image</code> also creates its own internal copy of the <code>shared_ptr</code> that shares ownership of the <code>hostData</code> memory, which means the application can safely release ownership of this <code>shared_ptr</code> when the constructor returns.</p> <p>When <code>hostPointer</code> is empty, construct a SYCL <code>sampld_image</code> with uninitialized memory.</p> <p>The host address is <code>const</code>, so the host accesses can be read-only. Since, the <code>hostPointer</code> is <code>const</code>, this image is only initialized with this memory and there is no write after its destruction. The element size of the constructed SYCL <code>sampld_image</code> will be derived from the <code>format</code> parameter. Accessors that read the constructed SYCL <code>sampld_image</code> will use the <code>sampler</code> parameter to sample the image. The range of the constructed SYCL <code>sampld_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>sampld_image</code> will be the <code>pitch</code> parameter provided. Zero or more properties can be provided to the constructed SYCL <code>sampld_image</code> via an instance of <code>property_list</code>.</p>

Table 29. Member functions of the `sampld_image` class template

Member function	Description
<pre>range&lt;Dimensions&gt; get_range() const</pre>	Return a range object representing the size of the image in terms of the number of elements in each dimension as passed to the constructor.
<pre>range&lt;Dimensions - 1&gt; get_pitch() const</pre>	<p>Available only when: <code>Dimensions &gt; 1</code>.</p> <p>Return a range object representing the pitch of the image in bytes.</p>

Member function	Description
<code>size_t size() const noexcept</code>	Returns the total number of elements in the image. Equal to <code>get_range()[0] * ... * get_range()[Dimensions-1]</code> .
<code>size_t byte_size() const noexcept</code>	Returns the size of the image storage in bytes. The number of bytes may be greater than <code>size()*element size</code> due to padding of elements, rows and slices of the image for efficient access.
<pre>template &lt;typename DataT, image_target Targ = image_target::device&gt; sampled_image_accessor&lt;DataT, Dimensions, Targ&gt; get_access(handler&amp; commandGroupHandler)</pre>	Returns a valid <code>sampled_image_accessor</code> to the sampled image with the specified data type and target in the command group.
<pre>template &lt;typename DataT&gt; host_sampled_image_accessor&lt;DataT, Dimensions&gt; get_host_access()</pre>	Returns a valid <code>host_sampled_image_accessor</code> to the sampled image with the specified data type in the command group.

#### 4.7.3.3. Image properties

The properties that can be provided when constructing the SYCL `unsampled_image` and `sampled_image` classes are described in [Table 30](#).

```

1 namespace sycl {
2 namespace property {
3 namespace image {
4 class use_host_ptr {
5 public:
6   use_host_ptr() = default;
7 };
8
9 class use_mutex {
10 public:
11   use_mutex(std::mutex& mutexRef);
12
13   std::mutex* get_mutex_ptr() const;
14 };
15
16 class context_bound {
17 public:
18   context_bound(context boundContext);
19
20   context get_context() const;
21 };
22 } // namespace image
23 } // namespace property
24 } // namespace sycl
```

Table 30. Properties supported by the SYCL image classes

Property	Description
<code>property::image::use_host_ptr</code>	The <code>use_host_ptr</code> property adds the requirement that the SYCL runtime must not allocate any memory for the <code>image</code> and instead uses the provided host pointer directly. This prevents the SYCL runtime from allocating additional temporary storage on the host.
<code>property::image::use_mutex</code>	The property adds the requirement that the memory which is owned by the SYCL <code>image</code> can be shared with the application via a <code>std::mutex</code> provided to the property. The <code>std::mutex</code> is locked by the runtime whenever the data is in use and unlocked otherwise. The contents of <code>hostData</code> are guaranteed to reflect the contents of the image when the <code>std::mutex</code> is unlocked by the runtime.
<code>property::image::context_bound</code>	The <code>context_bound</code> property adds the requirement that the SYCL <code>image</code> can only be associated with a single SYCL <code>context</code> that is provided to the property.

The constructors and member functions of the image `property` classes are listed in [Table 31](#) and [Table 32](#)

Table 31. Constructors of the image `property` classes

Constructor	Description
<code>property::image::use_host_ptr::use_host_ptr()</code>	Constructs a SYCL <code>use_host_ptr</code> property instance.
<code>property::image::use_mutex::use_mutex(std::mutex&amp; mutexRef)</code>	Constructs a SYCL <code>use_mutex</code> property instance with a reference to <code>mutexRef</code> parameter provided.
<code>property::image::context_bound::context_bound(context boundContext)</code>	Constructs a SYCL <code>context_bound</code> property instance with a copy of a SYCL <code>context</code> .

Table 32. Member functions of the image `property` classes

Member function	Description
<code>std::mutex* property::image::use_mutex::get_mutex_ptr() const</code>	Returns the <code>std::mutex</code> which was specified when constructing this SYCL <code>use_mutex</code> property.
<code>context property::image::context_bound::get_context() const</code>	Returns the <code>context</code> which was specified when constructing this SYCL <code>context_bound</code> property.

#### 4.7.3.4. Image destruction rules

The rules are similar to those described in [Section 4.7.2.3](#).

For the lifetime of the image object, the associated host memory must be left available to the [SYCL runtime](#) and the contents of the associated host memory is unspecified until the image object is destroyed. If an image object value is copied, then only a reference to the underlying image object is copied. The underlying image object is reference-counted. Only after all image value references to the underlying image object have been destroyed is the actual image object itself destroyed.

If an image object is constructed with associated host memory, then its destructor blocks until all operations in all SYCL queues on that image object have completed. Any modifications to the image data will be copied back, if necessary, to the associated host memory. Any errors occurring during destruction are reported to any associated context's asynchronous error handler. If an image object is constructed with a storage object, then the storage object defines what blocking or copying behavior occurs on image object destruction.

#### 4.7.4. Sharing host memory with the SYCL data management classes

In order to allow the [SYCL runtime](#) to do memory management and allow for data dependencies, there are two classes defined, buffer and image. The default behavior for them is that a “raw” pointer is given during the construction of the data management class, with full ownership to use it until the destruction of the SYCL object.

In this section we go in greater detail on sharing or explicitly not sharing host memory with the SYCL data classes, and we will use the buffer class as an example. The same rules will apply to images as well.

##### 4.7.4.1. Default behavior

When using a SYCL buffer, the ownership of the pointer passed to the constructor of the class is, by default, passed to [SYCL runtime](#), and that pointer cannot be used on the host side until the buffer or image is destroyed. A SYCL application can access the contents of the memory managed by a SYCL buffer by using a [host\\_accessor](#) as defined in [Section 4.7.6](#). However, there is no guarantee that the host accessor will copy data back to the original host address used in its constructor.

The pointer passed in is the one used to copy data back to the host, if needed, before buffer destruction. The memory pointed by [host pointer](#) will not be de-allocated by the runtime, and the data is copied back from the device if there is a need for it.

##### 4.7.4.2. SYCL ownership of the host memory

In the case where there is host memory to be used for initialization of data but there is no intention of using that host memory after the buffer is destroyed, then the buffer can take full ownership of that host memory.

When a buffer owns the [host pointer](#) there is no copy back, by default. In this situation, the SYCL application may pass a unique pointer to the host data, which will be then used by the runtime internally to initialize the data in the device.

For example, the following could be used:

```
1 {
2   auto ptr = std::make_unique<int>(-1234);
3   buffer<int, 1> b { std::move(ptr), range { 1 } };
4   // ptr is not valid anymore.
5   // There is nowhere to copy data back
6 }
```

However, optionally the `buffer::set_final_data()` can be set to a `std::weak_ptr` to enable copying data back, to another host memory address that will be valid when the buffer is destroyed.

```

1 {
2   auto ptr = std::make_unique<int>(-42);
3   buffer<int, 1> b { std::move(ptr), range { 1 } };
4   // ptr is not valid anymore.
5   // There is nowhere to copy data back.
6   // To get copy back, a location can be specified:
7   b.set_final_data(std::weak_ptr<int> { .... })
8 }

```

#### 4.7.4.3. Shared SYCL ownership of the host memory

When an instance of `std::shared_ptr` is passed to the buffer constructor, then the buffer object and the developer's application share the memory region. If the shared pointer is still used on the application's side then the data will be copied back from the buffer or image and will be available to the application after the buffer or image is destroyed.

If the `shared_ptr` is not empty, the contents of the referenced memory are used to initialize the buffer. If the `shared_ptr` is empty, then the buffer is created with uninitialized memory.

When the buffer is destroyed and the data have potentially been updated, if the number of copies of the shared pointer outside the runtime is 0, there is no user-side shared pointer to read the data. Therefore the data is not copied out, and the buffer destructor does not need to wait for the data processes to be finished, as the outcome is not needed on the application's side.

This behavior can be overridden using the `set_final_data()` member function of the buffer class, which will by any means force the buffer destructor to wait until the data is copied to wherever the `set_final_data()` member function has put the data (or not wait nor copy if set final data is `nullptr`).

```

1 {
2   std::shared_ptr<int> ptr { data };
3   {
4     buffer<int, 1> b { ptr, range<2>{ 10, 10 } };
5     // update the data
6     [...]
7   } // Data is copied back because there is an user side shared_ptr
8 }

```

```

1 {
2   std::shared_ptr<int> ptr { data };
3   {
4     buffer<int, 1> b { ptr, range<2>{ 10, 10 } };
5     // update the data
6     [...]
7     ptr.reset();
8   } // Data is not copied back, there is no user side shared_ptr.
9 }

```

#### 4.7.5. Synchronization primitives

To prevent race conditions between accesses to the host memory owned by a `buffer` in the SYCL runtime

(e.g., by accessors) and in host code, it is necessary to use manual synchronization through a `host_accessor`, or by passing a `std::mutex` to the `buffer` constructor through a `property`.

When a `buffer` was constructed with a `std::mutex` property, the SYCL runtime is required to lock the mutex whenever the data is in use by the runtime, and unlock the mutex when the data is not in use by the SYCL runtime.

```

1 {
2   std::mutex m;
3   auto shD = std::make_shared<int>(42)
4   sycl::buffer b { shD, { sycl::property::buffer::use_mutex { m } } };
5   {
6     std::lock_guard lck { m };
7
8     // User accesses the data
9     do_something(shD);
10
11    /* m is unlocked when lck goes out of scope, either at normal ending of this
12       block or if an exception is thrown */
13  }
14 }

```

When the runtime releases the mutex, the user is guaranteed that the data has been copied back through the shared pointer --- unless the final data destination has been changed using the member function `set_final_data()`.

## 4.7.6. Accessors

**Accessors** provide three different capabilities: they provide access to the data managed by a `buffer` or `image`, they provide access to local memory on a `device`, and they define the **requirements** to memory objects which determine the scheduling of `kernels` (see [Section 3.8.1](#)).

A memory object requirement is created when an accessor is constructed, unless the accessor is a placeholder in which case the requirement is created when the accessor is bound to a `command` by calling `handler::require()`.

There are several different C++ classes that implement accessors:

- The `accessor` class provides access to data in a `buffer` from within a `command`.
- The `host_accessor` class provides access to data in a `buffer` from host code that is outside of a `command`. These accessors are typically used in `application scope`.
- The `local_accessor` class provides access to device local memory from within a `SYCL kernel function`.
- The `unsampled_image_accessor` and `sampled_image_accessor` classes provide access to data in an `unsampled_image` and `sampled_image` from within a `command`.
- The `host_unsampled_image_accessor` and `host_sampled_image_accessor` classes provide access to data in an `unsampled_image` and `sampled_image` from host code that is outside of a `command`. These accessors are typically used in `application scope`.

Accessor objects must always be constructed in host code, either in `command group scope` or in `application scope`. Whether the constructor blocks until data is available depends on the type of accessor. Those accessors which provide access to data within a `command` do not block. Instead, these accessors define a requirement which influences the scheduling of the `command`. Those accessors which provide access to data from host code do block until the data is available on the host.

For those accessors which provide access to data within a `command`, the member functions which access

data should only be called from within the `command`. Programs which call these member functions from outside of the `command` are ill formed. The sections below describe exactly which member functions fall into this category.

#### 4.7.6.1. Data type

All accessors have a `DataT` template parameter which specifies the type of each element that the accessor accesses. For `accessor` and `host_accessor`, this type must either match the type of each element in the underlying `buffer`, or it must be a `const` qualified version of that type.

For the image accessors (`unsampled_image_accessor`, `sampled_image_accessor`, `host_unsampled_image_accessor`, and `host_sampled_image_accessor`), `DataT` must be one of:

- `int4(vec<int32_t,4>)`,
- `uint4(vec<uint32_t,4>)`,
- `float4(vec<float,4>)`, or
- `half4(vec<half,4>)`

For `local_accessor` see [Section 4.7.6.11](#) for the allowable `DataT` types.

#### 4.7.6.2. Access modes

Most accessors have an `AccessMode` template parameter which specifies whether the accessor can read or write the underlying data. This information is used by the runtime when defining the requirements for the associated `command`, and it tells the runtime whether data needs to be transferred to or from a device before data can be accessed through the accessor.

The `access_mode` enumeration, shown in [Table 33](#), describes the potential modes of an accessor. However, not all accessor classes support all modes, so see the description of each class for more details.

```

1 namespace sycl {
2
3 enum class access_mode : /* unspecified */ {
4   read,
5   write,
6   read_write,
7   discard_write,      // Deprecated in SYCL 2020
8   discard_read_write, // Deprecated in SYCL 2020
9   atomic               // Deprecated in SYCL 2020
10 };
11
12 namespace access {
13 // The legacy type "access::mode" is deprecated.
14 using mode = sycl::access_mode;
15 } // namespace access
16
17 } // namespace sycl

```

Table 33. Enumeration of access modes available to accessors

access_mode	Description
<code>access_mode::read</code>	Read-only access.



access_mode	Description
<code>access_mode::write</code>	Write-only access.
<code>access_mode::read_write</code>	Read and write access.

#### 4.7.6.3. Deduction tags

Some accessor constructors take a `TagT` parameter, which is used to deduce template arguments for the constructor's class. Each of the access modes in [Table 33](#) has an associated tag, but there are additional tags which set other template parameters in addition to the access mode. The synopsis below shows the namespace scope variables that the implementation provides as possible values for the `TagT` parameter.

```

1 namespace sycl {
2
3 inline constexpr __unspecified__ read_only;
4 inline constexpr __unspecified__ read_write;
5 inline constexpr __unspecified__ write_only;
6 inline constexpr __unspecified__ read_only_host_task;
7 inline constexpr __unspecified__ read_write_host_task;
8 inline constexpr __unspecified__ write_only_host_task;
9
10 } // namespace sycl

```

The precise meaning of these tags depends on the specific accessor class that is being constructed, so they are described more fully below in the section that pertains to each of the accessor types.

#### 4.7.6.4. Properties

All accessor constructors accept a `property_list` parameter, which affects the semantics of the accessor. [Table 34](#) shows the set of all possible accessor properties and tells which properties are allowed when constructing each accessor class.

```

1 namespace sycl {
2 namespace property {
3 struct no_init {};
4 } // namespace property
5
6 inline constexpr property::no_init no_init;
7 } // namespace sycl

```

Table 34. Properties supported by accessors

Property	Allowed with	Description
<code>property::no_init</code>	<code>accessor</code> <code>host_accessor</code> <code>unsampled_image_accessor</code> <code>host_unsampled_image_accessor</code>	<p>This property is useful when an application expects to write new values to all of the accessor's elements without reading their previous values. The implementation can use this information to avoid copying the accessor's data in some cases. Following is a more formal description.</p> <p>This property is allowed only for accessors with <code>access_mode::write</code> or <code>access_mode::read_write</code> access modes. Attempting to construct an <code>access_mode::read</code> accessor with this property causes an <code>exception</code> with the <code>errc::invalid</code> error code to be thrown.</p> <p>The usage of this property is different depending on whether the accessor's underlying data type <code>DataT</code> is an implicit-lifetime type (as defined in the C++ core language). If it is an implicit-lifetime type, the accessor implicitly creates objects of that type with indeterminate values. The application is not required to write values to each element of the accessor, but unwritten elements of the accessor's buffer or image receive indeterminate values, even if those buffer or image elements previously had defined values. If this is a <code>ranged accessor</code>, this applies only to the elements within the accessor's range. The values of unwritten elements outside of this range are preserved.</p> <p>If <code>DataT</code> is not an implicit-lifetime type, the accessor merely allocates uninitialized memory, and the application is responsible for constructing objects in that memory (e.g. by calling <code>placement-new</code>). The application must create an object in each element of the accessor unless the corresponding element of the underlying buffer did not previously contain an object. If this is a <code>ranged accessor</code>, this applies only to the elements within the accessor's range. The content of objects in the buffer outside of this range is preserved.</p>



As stated above, the `property::no_init` property requires the application to construct an object for each accessor element when the element's type is not an implicit-lifetime type (except in the case when the corresponding buffer element did not previously contain an object). The reason for this requirement is to avoid the possibility of overwriting a valid object with indeterminate bytes, for example, when a `command` using the accessor completes. This means that the implementation can unconditionally copy memory from the device back to the host when the `command` completes, regardless of whether the `DataT` type is an implicit-lifetime type.

The constructors of the accessor property classes are listed in [Table 35](#).

Table 35. Constructors of the accessor property classes

Constructor	Description
<code>property::no_init::no_init()</code>	Constructs a <code>no_init</code> property instance.

#### 4.7.6.5. Read only accessors

Accessors which have an `AccessMode` template parameter can be declared as read-only by specifying `access_mode::read` for the template parameter. A read-only accessor provides read-only access to the underlying data and provides a "read" requirement for the memory object when it is constructed.

The `DataT` template parameter for a read-only accessor can optionally be `const` qualified, and the semantics of the accessor are unchanged. For example, an accessor declared with `const DataT` and `access_mode::read` has the same semantics as an accessor declared with `DataT` and `access_mode::read`.

As detailed in the sections below, some accessor types have a default value for `AccessMode`, which depends on whether the `DataT` parameter is `const` qualified. This provides a convenient way to declare a read-only accessor without explicitly specifying the access mode.

A `const` qualified `DataT` is only allowed for a read-only accessor. Programs which specify a `const` qualified `DataT` and any access mode other than `access_mode::read` are ill formed, and the implementation must issue a diagnostic in this case.

Each accessor class also provides implicit conversions between the two forms of read-only accessors. This makes it possible, for example, to assign an accessor whose type has `const DataT` and `access_mode::read` to an accessor whose type has `DataT` and `access_mode::read`, so long as the other template parameters are the same. There is also an implicit conversion from a read-write accessor to either of the forms of a read-only accessor. These implicit conversions are described in detail for each accessor class in the sections that follow.

#### 4.7.6.6. Accessing elements of an accessor

Accessors of type `accessor`, `host_accessor`, and `local_accessor` can have zero, one, two, or three Dimensions. A zero dimension accessor provides access to a single scalar element via an implicit conversion operator to the underlying type of that element and via an overloaded copy/move assignment operators from the underlying type of the element.

One, two, or three dimensional specializations of these accessors provide access to the elements they contain in two ways. The first way is through a subscript operator that takes an instance of an `id` class which has the same dimensionality as the accessor. The second way is by passing a single `size_t` value to multiple consecutive subscript operators as specified in [Section 3.11.2](#).

In all these cases, the reference to the contained element is of type `const DataT&` for read-only accessors and of type `DataT&` for other accessors.

Accessors of all types have a range that defines the set of indices that may be used to access elements. For buffer accessors, this is the range of the underlying buffer, unless it is a `ranged_accessor` in which case the range comes from the accessor's constructor. For image accessors, this is the range of the underlying image. Local accessors specify the range when the accessor is constructed. Any attempt to access an element via an index that is outside of this range produces undefined behavior.

#### 4.7.6.7. Container interface

Accessors of type `accessor`, `host_accessor`, and `local_accessor` meet the C++ requirement of `Reversible-Container`. The exception to this is that only `local_accessor` owns the underlying data, meaning that its

destructor destroys elements and frees the memory. The `accessor` and `host_accessor` types don't destroy any elements or free the memory on destruction. The iterator for the container interface meets the C++ requirement of `LegacyRandomAccessIterator` and the underlying pointers/references correspond to the address space specified by the accessor type. For multidimensional accessors the iterator linearizes the data according to [Section 3.11.1](#).

#### 4.7.6.8. Ranged accessors

Accessors of type `accessor` and `host_accessor` can be constructed from a sub-range of a `buffer` by providing a range and offset to the constructor. This limits the elements that can be accessed to the specified sub-range, which allows the implementation to perform certain optimizations such as reducing the amount of memory that needs to be copied to or from a device.

If the ranged accessor is multi-dimensional, the sub-range is allowed to describe a region of memory in the underlying buffer that is not contiguous in the linear address space. It is also legal to construct several ranged accessors for the same underlying buffer, either overlapping or non-overlapping.

A ranged accessor still creates a requisite for the entire underlying buffer, even for the portions not within the range. For example, if one command writes through a ranged accessor to one region of a buffer and a second command reads through a ranged accessor from a non-overlapping region of the same buffer, the second command must still be scheduled after the first because the requisites for the two commands are on the entire buffer, not on the sub-ranges of the ranged accessors.

Most of the accessor member functions which provide a reference to the underlying buffer elements are affected by a ranged accessor's offset and range. For example, calling `operator[](0)` on a one-dimensional ranged accessor returns a reference to the element at the position specified by the accessor's offset, which is not necessarily the first element in the buffer. In addition, the accessor's iterator functions iterate only over the elements that are within the sub-range.

The only exceptions are the `get_pointer` and `get_multi_ptr` member functions, which return a pointer to the beginning of the underlying buffer regardless of the accessor's offset. Applications using these functions must take care to manually add the offset before dereferencing the pointer because accessing an element that is outside of the accessor's range results in undefined behavior.



There is no change in behavior for ranged accessors with a range of zero. It still creates a requisite for the entire underlying buffer, and an attempt to access an element produces undefined behavior.

#### 4.7.6.9. Buffer accessor for commands

The `accessor` class provides access to data in a `buffer` in three different ways. It can be used to access the buffer's data from within a [SYCL kernel function](#) via the device's [global memory](#). It can also be used to access the buffer's data on host from within a host task. Finally, it can be used to get a native backend handle to the buffer from within a host task. The `AccessTarget` template parameter helps distinguish these three cases as shown in [Table 36](#).

Table 36. Description of access targets for buffer accessors

Access target	Meaning
<code>target::device</code>	Access a buffer from a <a href="#">SYCL kernel function</a> via device global memory. Also used to get a native backend handle to the buffer from within a host task.
<code>target::host_task</code>	Access a buffer's data on host from within a <a href="#">host task</a> .

When an `accessor` is used from within a [SYCL kernel function](#), the access target must be `target::device`, `target::constant_buffer`, or `target::local`; otherwise the behavior is undefined. See [Section 4.7.6.9.4.5](#) and [Section 4.7.6.9.4.7](#) for a description of the deprecated `target::constant_buffer` and `target::local` targets.

When an **accessor** is used from within a host task, the use of the accessor must correspond to the access target, otherwise the behavior is undefined. If the access target is **target::host\_task**, the accessor may only be used to access the buffer's data on host, from within the host task function. If the access target is **target::device**, the accessor may only be used to get a native backend handle for the buffer as described in [Section 4.10.2](#).

The dimensionality of the accessor must match the underlying buffer, however, there is a special case if the buffer is one-dimensional. In this case, the accessor may either be one-dimensional or it may be zero-dimensional. A zero-dimensional accessor has access to just the first element of the buffer, whereas a one-dimensional accessor has access to the entire buffer.

Certain **accessor** constructors create a "placeholder" accessor. Such an accessor is bound to a **buffer** and its semantics such as access target and access mode are defined. However, a placeholder accessor is not yet bound to a **command group**. Before such an accessor can be used in a **command**, it must be bound by calling **handler::require()**. Passing a placeholder accessor as an argument to a **command** without first being bound to a **command group** with **handler::require()** will result in undefined behavior.



Implementations are encouraged to throw either a synchronous or an asynchronous exception when a placeholder accessor, that has not been bound to the corresponding **command group** with **handler::require()**, is either passed as an argument to or is used inside a **command**.

#### 4.7.6.9.1. Interface for buffer command accessors

A synopsis of the **accessor** class is provided below, showing the interface when it is specialized with **target::device** or **target::host\_task**. Since some of the class types and member functions have the same name and meaning as other accessors, the common types and functions are described in [Section 4.7.6.12](#). The member types are listed in [Table 61](#) and [Table 37](#). The constructors are listed in [Table 38](#), and the member functions are listed in [Table 62](#) and [Table 39](#).

The additional common special member functions and common member functions are listed in [Section 4.5.2](#) in [Table 7](#) and [Table 8](#), respectively. For valid implicit conversions between accessor types refer to [Section 4.7.6.9.3](#). Additionally, accessors of the same type must be equality comparable both in the host application and also in [SYCL kernel functions](#).

```

1 namespace sycl {
2
3 enum class target : /* unspecified */ {
4     device,
5     host_task,
6     constant_buffer,      // Deprecated
7     local,                // Deprecated
8     host_buffer,          // Deprecated
9     global_buffer = device // Deprecated
10 };
11
12 namespace access {
13 // The legacy type "access::target" is deprecated.
14 using sycl::target;
15
16 enum class placeholder : /* unspecified */ { // Deprecated
17     false_t,
18     true_t
19 };
20
21 } // namespace access

```

```

22
23 template <typename DataT, int Dimensions = 1,
24           access_mode AccessMode =
25             (std::is_const_v<DataT> ? access_mode::read
26              : access_mode::read_write),
27           target AccessTarget = target::device,
28           access::placeholder isPlaceholder = access::placeholder::false_t>
29 class accessor {
30 public:
31     using value_type = // const DataT for read-only accessors, DataT otherwise
32         __value_type__;
33     using reference = value_type&;
34     using const_reference = const DataT&;
35     template <access::decorated IsDecorated>
36     using accessor_ptr = // multi_ptr to value_type with target address space,
37         __pointer_class__; // unspecified for access_mode::host_task
38     using iterator = __unspecified_iterator__<value_type>;
39     using const_iterator = __unspecified_iterator__<const value_type>;
40     using reverse_iterator = std::reverse_iterator<iterator>;
41     using const_reverse_iterator = std::reverse_iterator<const_iterator>;
42     using difference_type =
43         typename std::iterator_traits<iterator>::difference_type;
44     using size_type = size_t;
45
46     accessor();
47
48     /* Available only when: (Dimensions == 0) */
49     template <typename AllocatorT>
50     accessor(buffer<DataT, 1, AllocatorT>& bufferRef,
51             const property_list& propList = {});
52
53     /* Available only when: (Dimensions == 0) */
54     template <typename AllocatorT>
55     accessor(buffer<DataT, 1, AllocatorT>& bufferRef,
56             handler& commandGroupHandlerRef, const property_list& propList = {});
57
58     /* Available only when: (Dimensions > 0) */
59     template <typename AllocatorT>
60     accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
61             const property_list& propList = {});
62
63     /* Available only when: (Dimensions > 0) */
64     template <typename AllocatorT, typename TagT>
65     accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef, TagT tag,
66             const property_list& propList = {});
67
68     /* Available only when: (Dimensions > 0) */
69     template <typename AllocatorT>
70     accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
71             handler& commandGroupHandlerRef, const property_list& propList = {});
72
73     /* Available only when: (Dimensions > 0) */
74     template <typename AllocatorT, typename TagT>
75     accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
76             handler& commandGroupHandlerRef, TagT tag,
77             const property_list& propList = {});

```



```

78
79  /* Available only when: (Dimensions > 0) */
80  template <typename AllocatorT>
81  accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
82           range<Dimensions> accessRange, const property_list& propList = {});
83
84  /* Available only when: (Dimensions > 0) */
85  template <typename AllocatorT, typename TagT>
86  accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
87           range<Dimensions> accessRange, TagT tag,
88           const property_list& propList = {});
89
90  /* Available only when: (Dimensions > 0) */
91  template <typename AllocatorT>
92  accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
93           range<Dimensions> accessRange, id<Dimensions> accessOffset,
94           const property_list& propList = {});
95
96  /* Available only when: (Dimensions > 0) */
97  template <typename AllocatorT, typename TagT>
98  accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
99           range<Dimensions> accessRange, id<Dimensions> accessOffset, TagT tag,
100          const property_list& propList = {});
101
102  /* Available only when: (Dimensions > 0) */
103  template <typename AllocatorT>
104  accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
105           handler& commandGroupHandlerRef, range<Dimensions> accessRange,
106           const property_list& propList = {});
107
108  /* Available only when: (Dimensions > 0) */
109  template <typename AllocatorT, typename TagT>
110  accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
111           handler& commandGroupHandlerRef, range<Dimensions> accessRange,
112           TagT tag, const property_list& propList = {});
113
114  /* Available only when: (Dimensions > 0) */
115  template <typename AllocatorT>
116  accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
117           handler& commandGroupHandlerRef, range<Dimensions> accessRange,
118           id<Dimensions> accessOffset, const property_list& propList = {});
119
120  /* Available only when: (Dimensions > 0) */
121  template <typename AllocatorT, typename TagT>
122  accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
123           handler& commandGroupHandlerRef, range<Dimensions> accessRange,
124           id<Dimensions> accessOffset, TagT tag,
125           const property_list& propList = {});
126
127  /* -- common interface members -- */
128
129  void swap(accessor& other);
130
131  bool is_placeholder() const;
132
133  size_type byte_size() const noexcept;

```

```

134
135   size_type size() const noexcept;
136
137   size_type max_size() const noexcept;
138
139   // Deprecated
140   size_t get_size() const;
141
142   // Deprecated
143   size_t get_count() const;
144
145   bool empty() const noexcept;
146
147   /* Available only when: (Dimensions > 0) */
148   range<Dimensions> get_range() const;
149
150   /* Available only when: (Dimensions > 0) */
151   id<Dimensions> get_offset() const;
152
153   /* Available only when: (AccessMode != access_mode::atomic && Dimensions == 0) */
154   operator reference() const;
155
156   /* Available only when: (AccessMode != access_mode::atomic &&
157                          AccessMode != access_mode::read && Dimensions == 0) */
158   const accessor& operator=(const value_type& other) const;
159
160   /* Available only when: (AccessMode != access_mode::atomic &&
161                          AccessMode != access_mode::read && Dimensions == 0) */
162   const accessor& operator=(value_type&& other) const;
163
164   /* Available only when: (Dimensions > 0) */
165   reference operator[](id<Dimensions> index) const;
166
167   /* Available only when: (Dimensions > 1) */
168   __unspecified__ operator[](size_t index) const;
169
170   /* Available only when: (AccessMode != access_mode::atomic && Dimensions == 1)
171   */
172   reference operator[](size_t index) const;
173
174   /* Deprecated
175   Available only when: (AccessMode == access_mode::atomic && Dimensions == 0)
176   */
177   operator sycl::atomic<DataT, access::address_space::global_space>() const;
178
179   /* Deprecated
180   Available only when: (AccessMode == access_mode::atomic && Dimensions == 1) */
181   sycl::atomic<DataT, access::address_space::global_space>
182   operator[](id<Dimensions> index) const;
183
184   /* Deprecated in SYCL 2020
185   Available only when: (AccessTarget == target::device) */
186   global_ptr<value_type> get_pointer() const noexcept;
187
188   /* Available only when: (AccessTarget == target::host_task) */
189   std::add_pointer_t<value_type> get_pointer() const noexcept;

```



```

190
191  /* Available only when: (AccessTarget == target::device) */
192  template <access::decorated IsDecorated>
193  accessor_ptr<IsDecorated> get_multi_ptr() const noexcept;
194
195  iterator begin() const noexcept;
196
197  iterator end() const noexcept;
198
199  const_iterator cbegin() const noexcept;
200
201  const_iterator cend() const noexcept;
202
203  reverse_iterator rbegin() const noexcept;
204
205  reverse_iterator rend() const noexcept;
206
207  const_reverse_iterator crbegin() const noexcept;
208
209  const_reverse_iterator crend() const noexcept;
210 };
211
212 } // namespace sycl

```

Table 37. Member types of the `accessor` class

Member types	Description
<pre>template &lt;access::decorated IsDecorated&gt; accessor_ptr</pre>	<p>If <code>(AccessTarget == target::device)</code>: <code>multi_ptr&lt;value_type, access::address_space::global_space, IsDecorated&gt;</code>.</p> <p>The definition of this type is not specified when <code>(AccessTarget == target::host_task)</code>.</p>

Table 38. Constructors of the `accessor` class

Constructor	Description
<pre>accessor()</pre>	<p>Constructs an empty accessor which fulfills the following post-conditions:</p> <ul style="list-style-type: none"> <li>• <code>(empty() == true)</code></li> <li>• All size queries return <code>0</code>.</li> <li>• The return values of <code>get_pointer()</code> and <code>get_multi_ptr()</code> are unspecified.</li> <li>• A default constructed accessor can be passed to a <a href="#">SYCL kernel function</a>, but attempting to access data elements from it produces undefined behavior.</li> </ul>

Constructor	Description
<pre>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, 1, AllocatorT&gt;&amp; bufferRef,          const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions == 0</code>).</p> <p>Constructs a placeholder <code>accessor</code> for accessing the first element of a <code>buffer</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p>
<pre>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, 1, AllocatorT&gt;&amp; bufferRef,          handler&amp; commandGroupHandlerRef, const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions == 0</code>).</p> <p>Constructs an <code>accessor</code> for accessing the first element of a <code>buffer</code> within a SYCL kernel function on the <code>queue</code> associated with <code>commandGroupHandlerRef</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p>
<pre>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Constructs a placeholder <code>accessor</code> for accessing a <code>buffer</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p>
<pre>template &lt;typename AllocatorT, typename TagT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          TagT tag,          const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Constructs a placeholder <code>accessor</code> for accessing a <code>buffer</code>. The <code>tag</code> is used to deduce template arguments of the accessor as described in <a href="#">Section 4.7.6.9.2</a>. The optional <code>property_list</code> provides properties for the constructed accessor.</p>
<pre>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          handler&amp; commandGroupHandlerRef, const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Constructs an <code>accessor</code> for accessing a <code>buffer</code> within a SYCL kernel function on the <code>queue</code> associated with <code>commandGroupHandlerRef</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p>

Constructor	Description
<pre>template &lt;typename AllocatorT, typename TagT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          handler&amp; commandGroupHandlerRef, TagT tag,          const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Constructs an <code>accessor</code> for accessing a <code>buffer</code> within a <code>SYCL kernel function</code> on the <code>queue</code> associated with <code>commandGroupHandlerRef</code>. The <code>tag</code> is used to deduce template arguments of the accessor as described in <a href="#">Section 4.7.6.9.2</a>. The optional <code>property_list</code> provides properties for the constructed accessor.</p>
<pre>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          range&lt;Dimensions&gt; accessRange, const          property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Constructs a placeholder <code>accessor</code> that is a <code>ranged accessor</code>, where the range starts at the beginning of the <code>buffer</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if <code>accessRange</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>
<pre>template &lt;typename AllocatorT, typename TagT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          range&lt;Dimensions&gt; accessRange, TagT tag,          const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Constructs a placeholder <code>accessor</code> that is a <code>ranged accessor</code>, where the range starts at the beginning of the <code>buffer</code>. The <code>tag</code> is used to deduce template arguments of the accessor as described in <a href="#">Section 4.7.6.9.2</a>. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if <code>accessRange</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>

Constructor	Description
<pre>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          range&lt;Dimensions&gt; accessRange, id&lt;Dimensions&gt;          accessOffset,          const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Constructs a placeholder <code>accessor</code> that is a <code>ranged accessor</code>, where the range starts at an offset from the beginning of the <code>buffer</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if the sum of <code>accessRange</code> and <code>accessOffset</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>
<pre>template &lt;typename AllocatorT, typename TagT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          range&lt;Dimensions&gt; accessRange, id&lt;Dimensions&gt;          accessOffset, TagT tag,          const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Constructs a placeholder <code>accessor</code> that is a <code>ranged accessor</code>, where the range starts at an offset from the beginning of the <code>buffer</code>. The <code>tag</code> is used to deduce template arguments of the accessor as described in <a href="#">Section 4.7.6.9.2</a>. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if the sum of <code>accessRange</code> and <code>accessOffset</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>
<pre>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          handler&amp; commandGroupHandlerRef, range&lt;Dimensions&gt;          accessRange,          const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Constructs an <code>accessor</code> that is a <code>ranged accessor</code>, where the range starts at the beginning of the <code>buffer</code>. The accessor can only be used in a <code>SYCL kernel function</code> on the <code>queue</code> associated with <code>commandGroupHandlerRef</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if <code>accessRange</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>

Constructor	Description
<pre>template &lt;typename AllocatorT, typename TagT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          handler&amp; commandGroupHandlerRef, range&lt;Dimensions&gt; accessRange,          TagT tag, const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Constructs an <code>accessor</code> that is a <code>ranged accessor</code>, where the range starts at the beginning of the <code>buffer</code>. The accessor can only be used in a <code>SYCL kernel function</code> on the <code>queue</code> associated with <code>commandGroupHandlerRef</code>. The <code>tag</code> is used to deduce template arguments of the accessor as described in <a href="#">Section 4.7.6.9.2</a>. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if <code>accessRange</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>
<pre>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          handler&amp; commandGroupHandlerRef, range&lt;Dimensions&gt; accessRange,          id&lt;Dimensions&gt; accessOffset, const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Constructs an <code>accessor</code> that is a <code>ranged accessor</code>, where the range starts at an offset from the beginning of the <code>buffer</code>. The accessor can only be used in a <code>SYCL kernel function</code> on the <code>queue</code> associated with <code>commandGroupHandlerRef</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if the sum of <code>accessRange</code> and <code>accessOffset</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>

Constructor	Description
<pre>template &lt;typename AllocatorT, typename TagT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          handler&amp; commandGroupHandlerRef, range&lt;Dimensions&gt; accessRange,          id&lt;Dimensions&gt; accessOffset, TagT tag,          const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Constructs an <code>accessor</code> that is a <code>ranged accessor</code>, where the range starts at an offset from the beginning of the <code>buffer</code>. The accessor can only be used in a <code>SYCL kernel function</code> on the <code>queue</code> associated with <code>commandGroupHandlerRef</code>. The <code>tag</code> is used to deduce template arguments of the accessor as described in <a href="#">Section 4.7.6.9.2</a>. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if the sum of <code>accessRange</code> and <code>accessOffset</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>

Table 39. Member functions of the `accessor` class

Member function	Description
<pre>void swap(accessor&amp; other);</pre>	Swaps the contents of the current accessor with the contents of <code>other</code> .
<pre>bool is_placeholder() const</pre>	Returns <code>true</code> if the accessor is a placeholder. Otherwise returns <code>false</code> .
<pre>id&lt;Dimensions&gt; get_offset() const</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>If this is a <code>ranged accessor</code>, returns the offset that was specified when the accessor was constructed. For other accessors, returns the default constructed <code>id&lt;Dimensions&gt;{}</code>.</p>

Member function	Description
<code>global_ptr&lt;value_type&gt; get_pointer() const noexcept</code>	<p>Available only when (<code>AccessTarget == target::device</code>).</p> <p>Returns a <code>multi_ptr</code> to the start of this accessor's underlying buffer, even if this is a <code>ranged accessor</code> whose range does not start at the beginning of the buffer. The return value is unspecified if the accessor is empty.</p> <p>This function may only be called from within a <code>command</code>.</p> <p>Deprecated in SYCL 2020. Use <code>get_multi_ptr</code> instead.</p>
<code>std::add_pointer_t&lt;value_type&gt; get_pointer() const noexcept</code>	<p>Available only when (<code>AccessTarget == target::host_task</code>).</p> <p>Returns a pointer to the start of this accessor's underlying buffer, even if this is a <code>ranged accessor</code> whose range does not start at the beginning of the buffer. The return value is unspecified if the accessor is empty.</p> <p>This function may only be called from within a <code>command</code>.</p>
<code>template &lt;access::decorated IsDecorated&gt; accessor_ptr&lt;IsDecorated&gt; get_multi_ptr() const noexcept</code>	<p>Available only when (<code>AccessTarget == target::device</code>).</p> <p>Returns a <code>multi_ptr</code> to the start of this accessor's underlying buffer, even if this is a <code>ranged accessor</code> whose range does not start at the beginning of the buffer. The return value is unspecified if the accessor is empty.</p> <p>This function may only be called from within a <code>command</code>.</p>
<code>const accessor&amp; operator=(const value_type&amp; other) const</code>	<p>Available only when (<code>AccessMode != access_mode::atomic &amp;&amp; AccessMode != access_mode::read &amp;&amp; Dimensions == 0</code>).</p> <p>Assignment to the single element that is accessed by this accessor.</p> <p>This function may only be called from within a <code>command</code>.</p>

Member function	Description
<pre>const accessor&amp; operator=(value_type&amp;&amp; other) const</pre>	<p>Available only when <code>(AccessMode != access_mode::atomic &amp;&amp; AccessMode != access_mode::read &amp;&amp; Dimensions == 0)</code>.</p> <p>Assignment to the single element that is accessed by this accessor.</p> <p>This function may only be called from within a <a href="#">command</a>.</p>

#### 4.7.6.9.2. Deduction tags for buffer command accessors

Some `accessor` constructors take a `TagT` parameter, which is used to deduce template arguments. The permissible values for this parameter are listed in [Table 40](#) along with the access mode and accessor target that they imply.

Table 40. Enumeration of tags available for `accessor` construction

Tag value	Access mode	Accessor target
<code>read_write</code>	<code>access_mode::read_write</code>	<code>target::device</code>
<code>read_only</code>	<code>access_mode::read</code>	<code>target::device</code>
<code>write_only</code>	<code>access_mode::write</code>	<code>target::device</code>
<code>read_write_host_task</code>	<code>access_mode::read_write</code>	<code>target::host_task</code>
<code>read_only_host_task</code>	<code>access_mode::read</code>	<code>target::host_task</code>
<code>write_only_host_task</code>	<code>access_mode::write</code>	<code>target::host_task</code>

#### 4.7.6.9.3. Read only buffer command accessors and implicit conversions

[Table 41](#) shows the specializations of `accessor` with `target::device` or `target::host_task` that are read-only accessors. There is an implicit conversion between any of these specializations, provided that all other template parameters are the same.

Table 41. Specializations of `accessor` that are read-only

Data type	Access mode
not const-qualified	<code>access_mode::read</code>
const-qualified	<code>access_mode::read</code>

There is also an implicit conversion from the read-write specialization shown in [Table 42](#) to any of the read-only specializations shown in [Table 41](#), provided that all other template parameters are the same.

Table 42. Specializations of `accessor` that are read-write

Data type	Access mode
not const-qualified	<code>access_mode::read_write</code>

#### 4.7.6.9.4. Deprecated features of the `accessor` class

All of the features defined in this section are deprecated and will likely be removed from a future version of the specification.

##### 4.7.6.9.4.1. Aliased names

The enumerated value `target::global_buffer` is an alias for `target::device`. It has the same type and



value as its alias.

The enumerated type `access::target` is an alias for `target`, and the enumerated type `access::mode` is an alias for `access_mode`.

#### 4.7.6.9.4.2. Discard access modes

An `accessor` instance specialized with access mode `access_mode::discard_write` has the same behavior as an `accessor` instance of mode `access_mode::write` that is constructed with the property `property::no_init`.

An `accessor` instance specialized with access mode `access_mode::discard_read_write` has the same behavior as an `accessor` instance of mode `access_mode::read_write` that is constructed with the property `property::no_init`.

#### 4.7.6.9.4.3. Placeholder template parameter

The `accessor` template parameter `IsPlaceholder` is allowed to be specified, but it has no bearing on whether the `accessor` instance is a placeholder. This is determined solely by the constructor used to create the instance.

The associated type `access::placeholder` is also deprecated.

#### 4.7.6.9.4.4. Additional member functions for `target::device` specialization

Specializations of the `accessor` class with `target::device` have the additional member functions described in Table 43.

Table 43. Deprecated member functions of the `accessor` class

Member function	Description
<code>size_t get_size() const</code>	Returns the same value as <code>byte_size()</code> .
<code>size_t get_count() const</code>	Returns the same value as <code>size()</code> .

#### 4.7.6.9.4.5. Accessor specialization with `target::constant_buffer`

The `accessor` class may be specialized with target `target::constant_buffer`, which results in an accessor that can be used within a SYCL kernel function to access the contents of a buffer through the device's constant memory.

As with other `accessor` specializations, the dimensionality must match the underlying buffer, however there is a special case if the buffer is one-dimensional. In this case, the accessor may either be one-dimensional or it may be zero-dimensional. A zero-dimensional accessor has access to just the first element of the buffer, whereas a one-dimensional accessor has access to the entire buffer.

This specialization of `accessor` is available only for the access mode `access_mode::read`.

This accessor type can be constructed as a "placeholder" accessor. As with other `accessor` specializations that are placeholders, `handler::require()` must be called before passing a placeholder accessor to a `command`. Passing a placeholder accessor as an argument to a `command` without first being bound to a `command group` with `handler::require()` will result in undefined behavior.

A synopsis for this specialization of `accessor` is provided below. Since some of the class types and member functions have the same name and meaning as other accessors, the common types and functions are described in Section 4.7.6.9.4.8. The member types are listed in Table 50. The constructors are listed in Table 44, and the member functions are listed in Table 51 and Table 45.

The additional common special member functions and common member functions are listed in [Section 4.5.2](#) in [Table 7](#) and [Table 8](#), respectively. Additionally, accessors of the same type must be equality comparable.

```

1 namespace sycl {
2
3 template <typename DataT, int Dimensions, access_mode AccessMode,
4         target AccessTarget, access::placeholder IsPlaceholder>
5 class accessor {
6 public:
7     using value_type = const DataT;
8     using reference = const DataT&
9     using const_reference = const DataT&
10
11     /* Available only when: (Dimensions == 0) */
12     template <typename AllocatorT>
13     accessor(buffer<DataT, 1, AllocatorT>& bufferRef,
14             const property_list& propList = {});
15
16     /* Available only when: (Dimensions == 0) */
17     template <typename AllocatorT>
18     accessor(buffer<DataT, 1, AllocatorT>& bufferRef,
19             handler& commandGroupHandlerRef, const property_list& propList = {});
20
21     /* Available only when: (Dimensions > 0) */
22     template <typename AllocatorT>
23     accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
24             const property_list& propList = {});
25
26     /* Available only when: (Dimensions > 0) */
27     template <typename AllocatorT>
28     accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
29             handler& commandGroupHandlerRef, const property_list& propList = {});
30
31     /* Available only when: (Dimensions > 0) */
32     template <typename AllocatorT>
33     accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
34             range<Dimensions> accessRange, const property_list& propList = {});
35
36     /* Available only when: (Dimensions > 0) */
37     template <typename AllocatorT>
38     accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
39             range<Dimensions> accessRange, id<Dimensions> accessOffset,
40             const property_list& propList = {});
41
42     /* Available only when: (Dimensions > 0) */
43     template <typename AllocatorT>
44     accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
45             handler& commandGroupHandlerRef, range<Dimensions> accessRange,
46             const property_list& propList = {});
47
48     /* Available only when: (Dimensions > 0) */
49     template <typename AllocatorT>
50     accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
51             handler& commandGroupHandlerRef, range<Dimensions> accessRange,
52             id<Dimensions> accessOffset, const property_list& propList = {});

```

```

53
54  /* -- common interface members -- */
55
56  bool is_placeholder() const;
57
58  size_t get_size() const noexcept;
59
60  size_t get_count() const noexcept;
61
62  /* Available only when: (Dimensions > 0) */
63  range<Dimensions> get_range() const;
64
65  /* Available only when: (Dimensions > 0) */
66  id<Dimensions> get_offset() const;
67
68  /* Available only when: (Dimensions == 0) */
69  operator reference() const;
70
71  /* Available only when: (Dimensions > 0) */
72  reference operator[](id<Dimensions> index) const;
73
74  /* Available only when: (Dimensions > 1) */
75  __unspecified__ operator[](size_t index) const;
76
77  /* Available only when: (Dimensions == 1) */
78  reference operator[](size_t index) const;
79
80  constant_ptr<DataT> get_pointer() const noexcept;
81 };
82
83 } // namespace sycl

```

Table 44. Constructors of the deprecated constant accessor

Constructor	Description
<pre> template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, 1, AllocatorT&gt;&amp; bufferRef,          const property_list&amp; propList = {}) </pre>	<p>Available only when <code>(Dimensions == 0)</code>.</p> <p>Constructs a placeholder <code>accessor</code> for accessing the first element of a <code>buffer</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p>
<pre> template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, 1, AllocatorT&gt;&amp; bufferRef,          handler&amp; commandGroupHandlerRef, const          property_list&amp; propList = {}) </pre>	<p>Available only when <code>(Dimensions == 0)</code>.</p> <p>Constructs an <code>accessor</code> for accessing the first element of a <code>buffer</code> within a SYCL kernel function on the <code>queue</code> associated with <code>commandGroupHandlerRef</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p>

Constructor	Description
<pre>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          const property_list&amp; propList = {})</pre>	<p>Available only when <code>(Dimensions &gt; 0)</code>.</p> <p>Constructs a placeholder <code>accessor</code> for accessing a <code>buffer</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p>
<pre>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          handler&amp; commandGroupHandlerRef, const          property_list&amp; propList = {})</pre>	<p>Available only when <code>(Dimensions &gt; 0)</code>.</p> <p>Constructs an <code>accessor</code> for accessing a <code>buffer</code> within a SYCL kernel function on the <code>queue</code> associated with <code>commandGroupHandlerRef</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p>
<pre>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          range&lt;Dimensions&gt; accessRange, const          property_list&amp; propList = {})</pre>	<p>Available only when <code>(Dimensions &gt; 0)</code>.</p> <p>Constructs a placeholder <code>accessor</code> that is a <code>ranged accessor</code>, where the range starts at the beginning of the <code>buffer</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if <code>accessRange</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>
<pre>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          range&lt;Dimensions&gt; accessRange, id&lt;Dimensions&gt;          accessOffset,          const property_list&amp; propList = {})</pre>	<p>Available only when <code>(Dimensions &gt; 0)</code>.</p> <p>Constructs a placeholder <code>accessor</code> that is a <code>ranged accessor</code>, where the range starts at an offset from the beginning of the <code>buffer</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if the sum of <code>accessRange</code> and <code>accessOffset</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>

Constructor	Description
<pre>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          handler&amp; commandGroupHandlerRef, range&lt;Dimensions&gt;          accessRange,          const property_list&amp; propList = {})</pre>	<p>Available only when <code>(Dimensions &gt; 0)</code>.</p> <p>Constructs an <code>accessor</code> that is a <code>ranged accessor</code>, where the range starts at the beginning of the <code>buffer</code>. The accessor can only be used in a <code>SYCL kernel function</code> on the <code>queue</code> associated with <code>commandGroupHandlerRef</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if <code>accessRange</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>
<pre>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          handler&amp; commandGroupHandlerRef, range&lt;Dimensions&gt;          accessRange,          id&lt;Dimensions&gt; accessOffset, const property_list&amp;          propList = {})</pre>	<p>Available only when <code>(Dimensions &gt; 0)</code>.</p> <p>Constructs an <code>accessor</code> that is a <code>ranged accessor</code>, where the range starts at an offset from the beginning of the <code>buffer</code>. The accessor can only be used in a <code>SYCL kernel function</code> on the <code>queue</code> associated with <code>commandGroupHandlerRef</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if the sum of <code>accessRange</code> and <code>accessOffset</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>

Table 45. Member functions of the deprecated constant accessor

Member function	Description
<pre>bool is_placeholder() const</pre>	<p>Returns <code>true</code> if the accessor was constructed as a placeholder and returns <code>false</code> otherwise.</p>
<pre>id&lt;Dimensions&gt; get_offset() const</pre>	<p>Available only when <code>(Dimensions &gt; 0)</code>.</p> <p>If this is a <code>ranged accessor</code>, returns the offset that was specified when the accessor was constructed, otherwise returns the default constructed <code>id&lt;Dimensions&gt;{}</code>.</p>

Member function	Description
<code>constant_ptr&lt;DataT&gt; get_pointer() const noexcept</code>	<p>Returns a <code>multi_ptr</code> to the start of this accessor's underlying buffer, even if this is a <code>ranged accessor</code> whose range does not start at the beginning of the buffer. The return value is unspecified if the accessor is empty.</p> <p>This function may only be called from within a <code>command</code>.</p>

#### 4.7.6.9.4.6. Accessor specialization with `target::host_buffer`

The `accessor` class may be specialized with target `target::host_buffer`, which results in a host accessor similar to `host_accessor`. This specialization provides access to data in a `buffer` from host code that is outside of a `command`, and constructors of this specialization block until the requested data is available on the host.

As with other `accessor` specializations, the dimensionality must match the underlying buffer, however there is a special case if the buffer is one-dimensional. In this case, the accessor may either be one-dimensional or it may be zero-dimensional. A zero-dimensional accessor has access to just the first element of the buffer, whereas a one-dimensional accessor has access to the entire buffer.

This specialization of `accessor` is available for all access modes except for `access_mode::atomic`.

A synopsis for this specialization of `accessor` is provided below. Since some of the class types and member functions have the same name and meaning as other accessors, the common types and functions are described in [Section 4.7.6.9.4.8](#). The member types are listed in [Table 50](#). The constructors are listed in [Table 46](#), and the member functions are listed in [Table 51](#) and [Table 47](#).

The additional common special member functions and common member functions are listed in [Section 4.5.2](#) in [Table 7](#) and [Table 8](#), respectively. Additionally, accessors of the same type must be equality comparable.

```

1 namespace sycl {
2
3 template <typename DataT, int Dimensions, access_mode AccessMode,
4         target AccessTarget, access::placeholder IsPlaceholder>
5 class accessor {
6 public:
7     using value_type = // const DataT for access_mode::read, DataT otherwise
8         __value_type__;
9     using reference = value_type&;
10    using const_reference = const DataT&;
11
12    /* Available only when: (Dimensions == 0) */
13    template <typename AllocatorT>
14    accessor(buffer<DataT, 1, AllocatorT>& bufferRef,
15            const property_list& propList = {});
16
17    /* Available only when: (Dimensions > 0) */
18    template <typename AllocatorT>
19    accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
20            const property_list& propList = {});
21

```

```

22  /* Available only when: (Dimensions > 0) */
23  template <typename AllocatorT>
24  accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
25           range<Dimensions> accessRange, const property_list& propList = {});
26
27  /* Available only when: (Dimensions > 0) */
28  template <typename AllocatorT>
29  accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
30           range<Dimensions> accessRange, id<Dimensions> accessOffset,
31           const property_list& propList = {});
32
33  /* -- common interface members -- */
34
35  bool is_placeholder() const;
36
37  size_t get_size() const;
38
39  size_t get_count() const;
40
41  /* Available only when: (Dimensions > 0) */
42  range<Dimensions> get_range() const;
43
44  /* Available only when: (Dimensions > 0) */
45  id<Dimensions> get_offset() const;
46
47  /* Available only when: (Dimensions == 0) */
48  operator reference() const;
49
50  /* Available only when: (Dimensions > 0) */
51  reference operator[](id<Dimensions> index) const;
52
53  /* Available only when: (Dimensions > 1) */
54  __unspecified__ operator[](size_t index) const;
55
56  /* Available only when: (Dimensions == 1) */
57  reference operator[](size_t index) const;
58
59  std::add_pointer_t<value_type> get_pointer() const noexcept;
60 };
61
62 } // namespace sycl

```

Table 46. Constructors of the deprecated host buffer accessor

Constructor	Description
<pre> template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, 1, AllocatorT&gt;&amp; bufferRef,          const property_list&amp; propList = {}) </pre>	<p>Available only when <code>(Dimensions == 0)</code>.</p> <p>Constructs an <code>accessor</code> for accessing the first element of a <code>buffer</code> immediately on the host. The optional <code>property_list</code> provides properties for the constructed accessor.</p>



Constructor	Description
<pre>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          const property_list&amp; propList = {})</pre>	<p>Available only when <code>(Dimensions &gt; 0)</code>.</p> <p>Constructs an <code>accessor</code> for accessing a <code>buffer</code> immediately on the host. The optional <code>property_list</code> provides properties for the constructed accessor.</p>
<pre>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          range&lt;Dimensions&gt; accessRange, const          property_list&amp; propList = {})</pre>	<p>Available only when <code>(Dimensions &gt; 0)</code>.</p> <p>Constructs an <code>accessor</code> that is a <code>ranged accessor</code> which accesses a buffer immediately on the host, where the range starts at the beginning of the buffer. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if <code>accessRange</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>
<pre>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,          range&lt;Dimensions&gt; accessRange, id&lt;Dimensions&gt;          accessOffset,          const property_list&amp; propList = {})</pre>	<p>Available only when <code>(Dimensions &gt; 0)</code>.</p> <p>Constructs an <code>accessor</code> that is a <code>ranged accessor</code> which accesses a buffer immediately on the host, where the range starts at an offset from the beginning of the buffer. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if the sum of <code>accessRange</code> and <code>accessOffset</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>

Table 47. Member functions of the deprecated host buffer accessor

Member function	Description
<pre>bool is_placeholder() const</pre>	<p>Always returns <code>false</code>.</p>



Member function	Description
<code>id&lt;Dimensions&gt; get_offset() const</code>	Available only when <code>(Dimensions &gt; 0)</code> .  If this is a <a href="#">ranged accessor</a> , returns the offset that was specified when the accessor was constructed, otherwise returns the default constructed <code>id&lt;Dimensions&gt;{}</code> .
<code>std::add_pointer_t&lt;value_type&gt; get_pointer() const noexcept</code>	Returns a pointer to the start of this accessor's underlying buffer, even if this is a <a href="#">ranged accessor</a> whose range does not start at the beginning of the buffer. The return value is unspecified if the accessor is empty.

#### 4.7.6.9.4.7. Accessor specialization with `target::local`

The `accessor` class may be specialized with target `target::local`, which results in a local accessor that has the same semantics and restrictions as `local_accessor`.

This specialization of `accessor` is only available for access modes `access_mode::read_write` and `access_mode::atomic`.

A synopsis for this specialization of `accessor` is provided below. Since some of the class types and member functions have the same name and meaning as other accessors, the common types and functions are described in [Section 4.7.6.9.4.8](#). The member types are listed in [Table 50](#). The constructors are listed in [Table 48](#), and the member functions are listed in [Table 51](#) and [Table 49](#).

The additional common special member functions and common member functions are listed in [Section 4.5.2](#) in [Table 7](#) and [Table 8](#), respectively. Additionally, accessors of the same type must be equality comparable.

```

1 namespace sycl {
2
3 template <typename DataT, int Dimensions, access_mode AccessMode,
4         target AccessTarget, access::placeholder IsPlaceholder>
5 class accessor {
6 public:
7     using value_type = DataT;
8     using reference = DataT&
9     using const_reference = const DataT&
10
11     /* Available only when: (Dimensions == 0) */
12     accessor(handler& commandGroupHandlerRef, const property_list& propList = {});
13
14     /* Available only when: (Dimensions > 0) */
15     accessor(range<Dimensions> allocationSize, handler& commandGroupHandlerRef,
16             const property_list& propList = {});
17
18     /* -- common interface members -- */
19
20     size_t get_size() const;
21

```

```

22  size_t get_count() const;
23
24  /* Available only when: (Dimensions > 0) */
25  range<Dimensions> get_range() const;
26
27  /* Available only when: (AccessMode == access_mode::read_write && Dimensions
28   * == 0) */
29  operator reference() const;
30
31  /* Available only when: (AccessMode == access_mode::read_write && Dimensions >
32   * 0) */
33  reference operator[](id<Dimensions> index) const;
34
35  /* Available only when: (Dimensions > 1) */
36  __unspecified__ operator[](size_t index) const;
37
38  /* Available only when: (AccessMode == access_mode::read_write && Dimensions
39   * == 1) */
40  reference operator[](size_t index) const;
41
42  /* Available only when: (AccessMode == access_mode::atomic && Dimensions == 0)
43   */
44  operator atomic<DataT, access::address_space::local_space>() const;
45
46  /* Available only when: (AccessMode == access_mode::atomic && Dimensions > 0)
47   */
48  atomic<DataT, access::address_space::local_space>
49  operator[](id<Dimensions> index) const;
50
51  /* Available only when: (AccessMode == access_mode::atomic && Dimensions == 1)
52   */
53  atomic<DataT, access::address_space::local_space>
54  operator[](size_t index) const;
55
56  local_ptr<DataT> get_pointer() const noexcept;
57 };
58
59 } // namespace sycl

```

Table 48. Constructors of the deprecated local accessor

Constructor	Description
<pre> <b>accessor</b>(<b>handler</b>&amp; commandGroupHandlerRef, <b>const</b> <b>property_list</b>&amp; propList = {}) </pre>	<p>Available only when <b>(Dimensions == 0)</b>.</p> <p>Constructs an <b>accessor</b> instance for accessing <b>local memory</b> of a single <b>DataT</b> element within a <b>SYCL kernel function</b> on the queue associated with <b>commandGroupHandlerRef</b>. The optional <b>property_list</b> provides properties for the constructed accessor.</p>

Constructor	Description
<pre> <b>accessor</b>(<b>range</b>&lt;Dimensions&gt; allocationSize, <b>handler</b>&amp; commandGroupHandlerRef,           <b>const</b> <b>property_list</b>&amp; propList = {}) </pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Constructs an <code>accessor</code> instance for accessing <code>local memory</code> of an array of <code>DataT</code> elements within a <code>SYCL kernel function</code> on the queue associated with <code>commandGroupHandlerRef</code>. The number of elements in the array is defined by <code>allocationSize</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p>

Table 49. Member functions of the deprecated local accessor

Member function	Description
<pre> <b>operator</b> <b>atomic</b>&lt;DataT, <b>access</b>::address_space::local_space&gt;() <b>const</b> </pre>	<p>Available only when (<code>AccessMode == access_mode::atomic &amp;&amp; Dimensions == 0</code>).</p> <p>Returns an instance of <code>atomic</code> of type <code>DataT</code> providing atomic access to the element stored within the work-group's local memory allocation that this accessor is accessing.</p> <p>This function may only be called from within a <code>command</code>.</p>
<pre> <b>atomic</b>&lt;DataT, <b>access</b>::address_space::local_space&gt; <b>operator</b>[(<b>id</b>&lt;Dimensions&gt; index)] <b>const</b> </pre>	<p>Available only when (<code>AccessMode == access_mode::atomic &amp;&amp; Dimensions &gt; 0</code>).</p> <p>Returns an instance of <code>atomic</code> of type <code>DataT</code> providing atomic access to the element stored within the work-group's local memory allocation that this accessor is accessing, at the index specified by <code>index</code>.</p> <p>This function may only be called from within a <code>command</code>.</p>
<pre> <b>atomic</b>&lt;DataT, <b>access</b>::address_space::local_space&gt; <b>operator</b>[(<b>size_t</b> index)] <b>const</b> </pre>	<p>Available only when (<code>AccessMode == access_mode::atomic &amp;&amp; Dimensions == 1</code>).</p> <p>Returns an instance of <code>atomic</code> of type <code>DataT</code> providing atomic access to the element stored within the work-group's local memory allocation that this accessor is accessing, at the index specified by <code>index</code>.</p> <p>This function may only be called from within a <code>command</code>.</p>

Member function	Description
<code>local_ptr&lt;DataT&gt; get_pointer() const noexcept</code>	<p>Returns a <code>multi_ptr</code> to the work-group's local memory allocation that this accessor is accessing. The return value is unspecified if the accessor is empty.</p> <p>This function may only be called from within a <code>command</code>.</p>

#### 4.7.6.9.4.8. Common members for deprecated accessors

Specializations of the `accessor` class with `target::constant_buffer`, `target::host_buffer` and `target::local` have many member types and member functions with the same name and meaning. [Table 50](#) describes these common types and [Table 51](#) describes the common member functions.

Table 50. Common member types of the deprecated accessors

Member types	Description
<code>value_type</code>	If <code>(AccessMode == access_mode::read)</code> , equal to <code>const DataT</code> , otherwise equal to <code>DataT</code> .
<code>reference</code>	Equal to <code>value_type&amp;</code> .
<code>const_reference</code>	Equal to <code>const DataT&amp;</code> .

Table 51. Common member functions of the deprecated accessors

Member function	Description
<code>size_t get_size() const noexcept</code>	<p>Returns the size in bytes of the memory region this accessor may access.</p> <p>When <code>AccessTarget</code> is <code>target::constant_buffer</code> or <code>target::host_buffer</code>, the returned value is the size of the elements in the underlying buffer, unless this is a <code>ranged accessor</code> in which case it is the size of the elements within the accessor's range.</p> <p>When <code>AccessTarget</code> is <code>target::local</code>, the returned value is the size in bytes of the accessor's local memory allocation, per work-group.</p>

Member function	Description
<pre>size_t get_count() const noexcept</pre>	<p>Returns the number of <code>DataT</code> elements of the memory region this accessor may access.</p> <p>When <code>AccessTarget</code> is <code>target::constant_buffer</code> or <code>target::host_buffer</code>, the returned value is the number of elements in the underlying buffer, unless this is a <code>ranged accessor</code> in which case it is the number of elements within the accessor's range.</p> <p>When <code>AccessTarget</code> is <code>target::local</code>, the returned value is the number of elements in the accessor's local memory allocation, per work-group.</p>
<pre>range&lt;Dimensions&gt; get_range() const</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Returns a <code>range</code> object which represents the number of elements of <code>DataT</code> per dimension that this accessor may access.</p> <p>When <code>AccessTarget</code> is <code>target::constant_buffer</code> or <code>target::host_buffer</code>, the returned value is the range of the underlying buffer, unless this is a <code>ranged accessor</code> in which case it is the range that was specified when the accessor was constructed.</p> <p>When <code>AccessTarget</code> is <code>target::local</code>, the returned value is the range that was specified when the accessor was constructed.</p>

Member function	Description
<code>operator reference() const</code>	<p>When <code>AccessTarget</code> is <code>target::constant_buffer</code> or <code>target::host_buffer</code>, available only when <code>(Dimensions == 0)</code>.</p> <p>When <code>AccessTarget</code> is <code>target::local</code>, available only when <code>(AccessMode == access_mode::read_write &amp;&amp; Dimensions == 0)</code>.</p> <p>Returns a reference to the single element that is accessed by this accessor.</p> <p>When <code>AccessTarget</code> is <code>target::local</code> or <code>target::constant_buffer</code>, this function may only be called from within a <code>command</code>.</p>
<code>reference operator[](id&lt;Dimensions&gt; index) const</code>	<p>When <code>AccessTarget</code> is <code>target::constant_buffer</code> or <code>target::host_buffer</code>, available only when <code>(Dimensions &gt; 0)</code>.</p> <p>When <code>AccessTarget</code> is <code>target::local</code>, available only when <code>(AccessMode == access_mode::read_write &amp;&amp; Dimensions &gt; 0)</code>.</p> <p>Returns a reference to the element at the location specified by <code>index</code>. If this is a <code>ranged accessor</code>, the element is determined by adding <code>index</code> to the accessor's offset.</p> <p>When <code>AccessTarget</code> is <code>target::local</code> or <code>target::constant_buffer</code>, this function may only be called from within a <code>command</code>.</p>

Member function	Description
<code>__unspecified__ operator[](size_t index) const</code>	<p>Available only when (<code>Dimensions &gt; 1</code>).</p> <p>Returns an instance of an undefined intermediate type representing this accessor, with the dimensionality <code>Dimensions-1</code> and containing an implicit <code>id</code> with index <code>Dimensions</code> set to <code>index</code>. The intermediate type returned must provide all available subscript operators which take a <code>size_t</code> parameter defined by this accessor class that are appropriate for the type it represents (including this subscript operator).</p> <p>If this is a <code>ranged accessor</code>, the implicit <code>id</code> in the returned instance also includes the accessor's offset.</p> <p>When <code>AccessTarget</code> is <code>target::local</code> or <code>target::constant_buffer</code>, this function may only be called from within a <code>command</code>.</p>
<code>reference operator[](size_t index) const</code>	<p>When <code>AccessTarget</code> is <code>target::constant_buffer</code> or <code>target::host_buffer</code>, available only when (<code>Dimensions == 1</code>).</p> <p>When <code>AccessTarget</code> is <code>target::local</code>, available only when (<code>AccessMode == access_mode::read_write &amp;&amp; Dimensions == 1</code>).</p> <p>Returns a reference to the element at the location specified by <code>index</code>. If this is a <code>ranged accessor</code>, the element is determined by adding <code>index</code> to the accessor's offset.</p> <p>When <code>AccessTarget</code> is <code>target::local</code> or <code>target::constant_buffer</code>, this function may only be called from within a <code>command</code>.</p>

#### 4.7.6.9.4.9. Accessor specialization with `access_mode::atomic`

The `accessor` class may be specialized with target `target::device` and access mode `access_mode::atomic`. This specialization provides additional member functions beyond those that are provided for other `target::device` specializations as described in Table 52.

Table 52. Deprecated atomic member functions of the `accessor` class

Member function	Description
<pre>operator atomic&lt;DataT, access::address_space::global_space&gt;() const</pre>	<p>Available only when (<code>AccessMode == access_mode::atomic &amp;&amp; Dimensions == 0</code>).</p> <p>Returns an instance of <code>atomic</code> of type <code>DataT</code> providing atomic access to the single element that is accessed by this accessor.</p>
<pre>atomic&lt;DataT, access::address_space::global_space&gt; operator[](id&lt;Dimensions&gt; index) const</pre>	<p>Available only when (<code>AccessMode == access_mode::atomic &amp;&amp; Dimensions &gt; 0</code>).</p> <p>Returns an instance of <code>atomic</code> of type <code>DataT</code> providing atomic access to the element stored within the accessor's buffer at the index specified by <code>index</code>.</p> <p>If this is a <code>ranged accessor</code>, the returned <code>atomic</code> instance provides access to the buffer element whose location is determined by adding the accessor's offset to <code>index</code>.</p>
<pre>atomic&lt;DataT, access::address_space::global_space&gt; operator[](size_t index) const</pre>	<p>Available only when (<code>AccessMode == access_mode::atomic &amp;&amp; Dimensions == 1</code>).</p> <p>Returns an instance of <code>atomic</code> of type <code>DataT</code> providing atomic access to the element stored within the accessor's buffer at the index specified by <code>index</code>.</p> <p>If this is a <code>ranged accessor</code>, the returned <code>atomic</code> instance provides access to the buffer element whose location is determined by adding the accessor's offset to <code>index</code>.</p>

#### 4.7.6.10. Buffer accessor for host code

The `host_accessor` class provides access to data in a `buffer` from host code that is outside of a `command` (i.e. do not use this class to access a buffer inside a host task).

As with `accessor`, the dimensionality of `host_accessor` must match the underlying buffer, however, there is a special case if the buffer is one-dimensional. In this case, the accessor may either be one-dimensional or it may be zero-dimensional. A zero-dimensional accessor has access to just the first element of the buffer, whereas a one-dimensional accessor has access to the entire buffer.

The `host_accessor` class supports the following access modes: `access_mode::read`, `access_mode::write` and `access_mode::read_write`.

##### 4.7.6.10.1. Interface for buffer host accessors

A synopsis of the `host_accessor` class is provided below. Since some of the class types and member functions have the same name and meaning as other accessors, the common types and functions are



described in [Section 4.7.6.12](#). The member types are listed in [Table 61](#). The constructors are listed in [Table 53](#), and the member functions are listed in [Table 62](#) and [Table 54](#).

The additional common special member functions and common member functions are listed in [Section 4.5.2](#) in [Table 7](#) and [Table 8](#), respectively. For valid implicit conversions between accessor types refer to [Section 4.7.6.10.3](#). Additionally, accessors of the same type must be equality comparable.

```

1 namespace sycl {
2 template <typename DataT, int Dimensions = 1,
3         access_mode AccessMode =
4             (std::is_const_v<DataT> ? access_mode::read
5              : access_mode::read_write)>
6 class host_accessor {
7 public:
8     using value_type = // const DataT for read-only accessors, DataT otherwise
9         __value_type__;
10    using reference = value_type&;
11    using const_reference = const DataT&;
12    using iterator = __unspecified_iterator__<value_type>;
13    using const_iterator = __unspecified_iterator__<const value_type>;
14    using reverse_iterator = std::reverse_iterator<iterator>;
15    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
16    using difference_type =
17        typename std::iterator_traits<iterator>::difference_type;
18    using size_type = size_t;
19
20    host_accessor();
21
22    /* Available only when: (Dimensions == 0) */
23    template <typename AllocatorT>
24    host_accessor(buffer<DataT, 1, AllocatorT>& bufferRef,
25                const property_list& proplist = {});
26
27    /* Available only when: (Dimensions > 0) */
28    template <typename AllocatorT>
29    host_accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
30                const property_list& proplist = {});
31
32    /* Available only when: (Dimensions > 0) */
33    template <typename AllocatorT, typename TagT>
34    host_accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef, TagT tag,
35                const property_list& proplist = {});
36
37    /* Available only when: (Dimensions > 0) */
38    template <typename AllocatorT>
39    host_accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
40                range<Dimensions> accessRange,
41                const property_list& proplist = {});
42
43    /* Available only when: (Dimensions > 0) */
44    template <typename AllocatorT, typename TagT>
45    host_accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
46                range<Dimensions> accessRange, TagT tag,
47                const property_list& proplist = {});
48
49    /* Available only when: (Dimensions > 0) */

```

```

50  template <typename AllocatorT>
51  host_accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
52               range<Dimensions> accessRange, id<Dimensions> accessOffset,
53               const property_list& propList = {});
54
55  /* Available only when: (Dimensions > 0) */
56  template <typename AllocatorT, typename TagT>
57  host_accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
58               range<Dimensions> accessRange, id<Dimensions> accessOffset,
59               TagT tag, const property_list& propList = {});
60
61  /* -- common interface members -- */
62
63  void swap(host_accessor& other);
64
65  size_type byte_size() const noexcept;
66
67  size_type size() const noexcept;
68
69  size_type max_size() const noexcept;
70
71  bool empty() const noexcept;
72
73  /* Available only when: (Dimensions > 0) */
74  range<Dimensions> get_range() const;
75
76  /* Available only when: (Dimensions > 0) */
77  id<Dimensions> get_offset() const;
78
79  /* Available only when: (Dimensions == 0) */
80  operator reference() const;
81
82  /* Available only when: (AccessMode != access_mode::read && Dimensions == 0) */
83  const host_accessor& operator=(const value_type& other) const;
84
85  /* Available only when: (AccessMode != access_mode::read && Dimensions == 0) */
86  const host_accessor& operator=(value_type&& other) const;
87
88  /* Available only when: (Dimensions > 0) */
89  reference operator[](id<Dimensions> index) const;
90
91  /* Available only when: (Dimensions > 1) */
92  __unspecified__ operator[](size_t index) const;
93
94  /* Available only when: (Dimensions == 1) */
95  reference operator[](size_t index) const;
96
97  std::add_pointer_t<value_type> get_pointer() const noexcept;
98
99  iterator begin() const noexcept;
100
101  iterator end() const noexcept;
102
103  const_iterator cbegin() const noexcept;
104
105  const_iterator cend() const noexcept;

```

```

106
107 reverse_iterator rbegin() const noexcept;
108
109 reverse_iterator rend() const noexcept;
110
111 const_reverse_iterator crbegin() const noexcept;
112
113 const_reverse_iterator crend() const noexcept;
114 };
115 } // namespace sycl

```

Table 53. Constructors of the `host_accessor` class

Constructor	Description
<code>host_accessor()</code>	<p>Constructs an empty accessor which fulfills the following post-conditions:</p> <ul style="list-style-type: none"> <li>• <code>(empty() == true)</code></li> <li>• All size queries return <code>0</code>.</li> <li>• The return value of <code>get_pointer()</code> is unspecified.</li> <li>• Trying to access the underlying memory is undefined behavior.</li> </ul>
<pre> template &lt;typename AllocatorT&gt; host_accessor(buffer&lt;DataT, 1, AllocatorT&gt;&amp; bufferRef,               const property_list&amp; propList = {}) </pre>	<p>Available only when <code>(Dimensions == 0)</code>.</p> <p>Constructs a <code>host_accessor</code> for accessing the first element of a <code>buffer</code> immediately on the host. The optional <code>property_list</code> provides properties for the constructed accessor.</p>
<pre> template &lt;typename AllocatorT&gt; host_accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,               const property_list&amp; propList = {}) </pre>	<p>Available only when <code>(Dimensions &gt; 0)</code>.</p> <p>Constructs a <code>host_accessor</code> for accessing a <code>buffer</code> immediately on the host. The optional <code>property_list</code> provides properties for the constructed accessor.</p>
<pre> template &lt;typename AllocatorT, typename TagT&gt; host_accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef, TagT tag,               const property_list&amp; propList = {}) </pre>	<p>Available only when <code>(Dimensions &gt; 0)</code>.</p> <p>Constructs a <code>host_accessor</code> for accessing a <code>buffer</code> immediately on the host. The <code>tag</code> is used to deduce template arguments of the accessor as described in <a href="#">Section 4.7.6.10.2</a>. The optional <code>property_list</code> provides properties for the constructed accessor.</p>

Constructor	Description
<pre>template &lt;typename AllocatorT&gt; host_accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,              range&lt;Dimensions&gt; accessRange, const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Constructs a <code>host_accessor</code> that is a <code>ranged accessor</code> which accesses a buffer immediately on the host, where the range starts at the beginning of the <code>buffer</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if <code>accessRange</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>
<pre>template &lt;typename AllocatorT, typename TagT&gt; host_accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,              range&lt;Dimensions&gt; accessRange, TagT tag,              const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Constructs a <code>host_accessor</code> that is a <code>ranged accessor</code> which accesses a buffer immediately on the host, where the range starts at the beginning of the <code>buffer</code>. The <code>tag</code> is used to deduce template arguments of the accessor as described in <a href="#">Section 4.7.6.10.2</a>. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if <code>accessRange</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>
<pre>template &lt;typename AllocatorT&gt; host_accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,              range&lt;Dimensions&gt; accessRange, id&lt;Dimensions&gt; accessOffset,              const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Constructs a <code>host_accessor</code> that is a <code>ranged accessor</code> which accesses a buffer immediately on the host, where the range starts at an offset from the beginning of the buffer. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if the sum of <code>accessRange</code> and <code>accessOffset</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>

Constructor	Description
<pre>template &lt;typename AllocatorT, typename TagT&gt; host_accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,                range&lt;Dimensions&gt; accessRange, id&lt;Dimensions&gt; accessOffset,                TagT tag, const property_list&amp; propList = {})</pre>	<p>Available only when <code>(Dimensions &gt; 0)</code>.</p> <p>Constructs a <code>host_accessor</code> that is a <code>ranged accessor</code> which accesses a buffer immediately on the host, where the range starts at an offset from the beginning of the buffer. The <code>tag</code> is used to deduce template arguments of the accessor as described in <a href="#">Section 4.7.6.10.2</a>. The optional <code>property_list</code> provides properties for the constructed accessor.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if the sum of <code>accessRange</code> and <code>accessOffset</code> exceeds the range of <code>bufferRef</code> in any dimension.</p>

Table 54. Member functions of the `host_accessor` class

Member function	Description
<pre>void swap(host_accessor&amp; other);</pre>	Swaps the contents of the current accessor with the contents of <code>other</code> .
<pre>id&lt;Dimensions&gt; get_offset() const</pre>	<p>Available only when <code>(Dimensions &gt; 0)</code>.</p> <p>If this is a <code>ranged accessor</code>, returns the offset that was specified when the accessor was constructed. For other accessors, returns the default constructed <code>id&lt;Dimensions&gt;{}</code>.</p>
<pre>std::add_pointer_t&lt;value_type&gt; get_pointer() const noexcept</pre>	Returns a pointer to the start of this accessor's underlying buffer, even if this is a <code>ranged accessor</code> whose range does not start at the beginning of the buffer. The return value is unspecified if the accessor is empty.
<pre>const host_accessor&amp; operator=(const value_type&amp; other) const</pre>	<p>Available only when <code>(AccessMode != access_mode::read &amp;&amp; Dimensions == 0)</code>.</p> <p>Assignment to the single element that is accessed by this accessor.</p>
<pre>const host_accessor&amp; operator=(value_type&amp;&amp; other) const</pre>	<p>Available only when <code>(AccessMode != access_mode::read &amp;&amp; Dimensions == 0)</code>.</p> <p>Assignment to the single element that is accessed by this accessor.</p>

#### 4.7.6.10.2. Deduction tags for buffer host accessors

Some `host_accessor` constructors take a `TagT` parameter, which is used to deduce template arguments. The permissible values for this parameter are listed in Table 55 along with the access mode that they imply.

Table 55. Enumeration of tags available for `host_accessor` construction

Tag value	Access mode
<code>read_write</code>	<code>access_mode::read_write</code>
<code>read_only</code>	<code>access_mode::read</code>
<code>write_only</code>	<code>access_mode::write</code>

#### 4.7.6.10.3. Read only buffer host accessors and implicit conversions

Table 56 shows the specializations of `host_accessor` that are read-only accessors. There is an implicit conversion between any of these specializations, provided that all other template parameters are the same.

Table 56. Specializations of `host_accessor` that are read-only

Data type	Access mode
not const-qualified	<code>access_mode::read</code>
const-qualified	<code>access_mode::read</code>

There is also an implicit conversion from the read-write `host_accessor` type shown in Table 57 to any of the read-only accessors in Table 56, provided that all other template parameters are the same.

Table 57. Specializations of `host_accessor` that are read-write

Data type	Access mode
not const-qualified	<code>access_mode::read_write</code>

#### 4.7.6.11. Local accessor

The `local_accessor` class allocates device local memory and provides access to this memory from within a SYCL kernel function. The local memory that is allocated is shared between all work-items of a work-group. If multiple work-groups execute simultaneously in an implementation, each work-group receives its own independent copy of the allocated local memory.

The underlying `DataT` type can be any C++ type that the device supports. If `DataT` is an implicit-lifetime type (as defined in the C++ core language), the local accessor implicitly creates objects of that type with indeterminate values. For other types, the local accessor merely allocates uninitialized memory, and the application is responsible for constructing objects in that memory (e.g. by calling `placement-new`).

A local accessor must not be used in a SYCL kernel function that is invoked via `single_task` or via the simple form of `parallel_for` that takes a `range` parameter. In these cases submitting the kernel to a queue must throw a synchronous exception with the `errc::kernel_argument` error code.

##### 4.7.6.11.1. Interface for local accessors

A synopsis of the `local_accessor` class is provided below. Since some of the class types and member functions have the same name and meaning as other accessors, the common types and functions are described in Section 4.7.6.12. The member types are listed in Table 61 and Table 58. The constructors are listed in Table 59, and the member functions are listed in Table 62 and Table 60.

The additional common special member functions and common member functions are listed in Section 4.5.2 in Table 7 and Table 8, respectively. For valid implicit conversions between accessor types refer to Section 4.7.6.11.2. Additionally, accessors of the same type must be equality comparable.

```

1 namespace sycl {
2 template <typename DataT, int Dimensions = 1> class local_accessor {
3 public:
4     using value_type = DataT;
5     using reference = value_type&
6     using const_reference = const DataT&
7     template <access::decorated IsDecorated>
8     using accessor_ptr =
9         multi_ptr<value_type, access::address_space::local_space, IsDecorated>;
10    using iterator = __unspecified_iterator__<value_type>;
11    using const_iterator = __unspecified_iterator__<const value_type>;
12    using reverse_iterator = std::reverse_iterator<iterator>;
13    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
14    using difference_type =
15        typename std::iterator_traits<iterator>::difference_type;
16    using size_type = size_t;
17
18    local_accessor();
19
20    /* Available only when: (Dimensions == 0) */
21    local_accessor(handler& commandGroupHandlerRef,
22                  const property_list& propList = {});
23
24    /* Available only when: (Dimensions > 0) */
25    local_accessor(range<Dimensions> allocationSize,
26                  handler& commandGroupHandlerRef,
27                  const property_list& propList = {});
28
29    /* -- common interface members -- */
30
31    void swap(local_accessor& other);
32
33    size_type byte_size() const noexcept;
34
35    size_type size() const noexcept;
36
37    size_type max_size() const noexcept;
38
39    bool empty() const noexcept;
40
41    range<Dimensions> get_range() const;
42
43    /* Available only when: (Dimensions == 0) */
44    operator reference() const;
45
46    /* Available only when: (!std::is_const_v<DataT> && Dimensions == 0) */
47    const local_accessor& operator=(const value_type& other) const;
48
49    /* Available only when: (!std::is_const_v<DataT> && Dimensions == 0) */
50    const local_accessor& operator=(value_type&& other) const;
51
52    /* Available only when: (Dimensions > 0) */
53    reference operator[](id<Dimensions> index) const;
54
55    /* Available only when: (Dimensions > 1) */

```

```

56  __unspecified__ operator[](size_t index) const;
57
58  /* Available only when: (Dimensions == 1) */
59  reference operator[](size_t index) const;
60
61  /* Deprecated in SYCL 2020 */
62  local_ptr<value_type> get_pointer() const noexcept;
63
64  template <access::decorated IsDecorated>
65  accessor_ptr<IsDecorated> get_multi_ptr() const noexcept;
66
67  iterator begin() const noexcept;
68
69  iterator end() const noexcept;
70
71  const_iterator cbegin() const noexcept;
72
73  const_iterator cend() const noexcept;
74
75  reverse_iterator rbegin() const noexcept;
76
77  reverse_iterator rend() const noexcept;
78
79  const_reverse_iterator crbegin() const noexcept;
80
81  const_reverse_iterator crend() const noexcept;
82 };
83 } // namespace sycl

```

Table 58. Member types of the `local_accessor` class

Member types	Description
<code>template &lt;access::decorated IsDecorated&gt; accessor_ptr</code>	Equal to <code>multi_ptr&lt;value_type, access::address_space::local_space, IsDecorated&gt;</code> .

Table 59. Constructors of the `local_accessor` class

Constructor	Description
<code>local_accessor()</code>	<p>Constructs an empty local accessor which fulfills the following post-conditions:</p> <ul style="list-style-type: none"> <li>• <code>(empty() == true)</code></li> <li>• All size queries return <code>0</code>.</li> <li>• The return values of <code>get_pointer()</code> and <code>get_multi_ptr()</code> are unspecified.</li> <li>• A default constructed local accessor can be passed to a <a href="#">SYCL kernel function</a>, but attempting to access data elements from it produces undefined behavior.</li> </ul>



Constructor	Description
<pre>local_accessor(handler&amp; commandGroupHandlerRef,                const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions == 0</code>).</p> <p>Constructs a <code>local_accessor</code> for accessing <code>local memory</code> of a single <code>DataT</code> element within a <code>SYCL kernel function</code> on the queue associated with <code>commandGroupHandlerRef</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p>
<pre>local_accessor(range&lt;Dimensions&gt; allocationSize,                handler&amp; commandGroupHandlerRef,                const property_list&amp; propList = {})</pre>	<p>Available only when (<code>Dimensions &gt; 0</code>).</p> <p>Constructs a <code>local_accessor</code> for accessing <code>local memory</code> of an array of <code>DataT</code> elements within a <code>SYCL kernel function</code> on the queue associated with <code>commandGroupHandlerRef</code>. The number of elements in the array is defined by <code>allocationSize</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</p>

Table 60. Member functions of the `local_accessor` class

Member function	Description
<pre>void swap(local_accessor&amp; other);</pre>	<p>Swaps the contents of the current accessor with the contents of <code>other</code>.</p>
<pre>local_ptr&lt;value_type&gt; get_pointer() const noexcept</pre>	<p>Returns a <code>multi_ptr</code> to the start of this accessor's local memory region which corresponds to the calling work-group. The return value is unspecified if the accessor is empty.</p> <p>This function may only be called from within a <code>command</code>.</p> <p>Deprecated in SYCL 2020. Use <code>get_multi_ptr</code> instead.</p>
<pre>template &lt;access::decorated IsDecorated&gt; accessor_ptr&lt;IsDecorated&gt; get_multi_ptr() const noexcept</pre>	<p>Returns a <code>multi_ptr</code> to the start of the accessor's local memory region which corresponds to the calling work-group. The return value is unspecified if the accessor is empty.</p> <p>This function may only be called from within a <code>SYCL kernel function</code>.</p>

Member function	Description
<pre>const local_accessor&amp; operator=(const value_type&amp; other) const</pre>	<p>Available only when <code>(!std::is_const_v&lt;DataT&gt; &amp;&amp; Dimensions == 0)</code>.</p> <p>Assignment to the single element that is accessed by this accessor.</p> <p>This function may only be called from within a <a href="#">command</a>.</p>
<pre>const local_accessor&amp; operator=(const value_type&amp;&amp; other) const</pre>	<p>Available only when <code>(!std::is_const_v&lt;DataT&gt; &amp;&amp; Dimensions == 0)</code>.</p> <p>Assignment to the single element that is accessed by this accessor.</p> <p>This function may only be called from within a <a href="#">command</a>.</p>

#### 4.7.6.11.2. Read only local accessors and implicit conversions

Since `local_accessor` has no template parameter for the access mode, the only specialization for a read-only local accessor is by providing a `const` qualified `DataT` parameter. Specializations with a non-`const` qualified `DataT` parameter are read-write. There is an implicit conversion from the read-write specialization to the read-only specialization, provided that all other template parameters are the same.

#### 4.7.6.12. Common members for buffer and local accessors

The `accessor`, `host_accessor`, and `local_accessor` classes have many member types and member functions with the same name and meaning. [Table 61](#) describes these common types and [Table 62](#) describes the common member functions.

Table 61. Common buffer and local accessor member types

Member types	Description
value_type	<p>If the accessor is read-only, equal to <code>const DataT</code>, otherwise equal to <code>DataT</code>.</p> <p>See <a href="#">Section 4.7.6.9.3</a>, <a href="#">Section 4.7.6.10.3</a> and <a href="#">Section 4.7.6.11.2</a> for which accessors are considered read-only.</p>
reference	Equal to <code>value_type&amp;</code> .
const_reference	Equal to <code>const DataT&amp;</code> .

Member types	Description
<code>iterator</code>	Iterator that can provide ranged access. Cannot be written to if the accessor is read-only. The underlying pointer is address space qualified for <code>accessor</code> specializations with <code>target::device</code> and for <code>local_accessor</code> .
<code>const_iterator</code>	Iterator that can provide ranged access. Cannot be written to. The underlying pointer is address space qualified for <code>accessor</code> specializations with <code>target::device</code> and for <code>local_accessor</code> .
<code>reverse_iterator</code>	Iterator adaptor that reverses the direction of <code>iterator</code> .
<code>const_reverse_iterator</code>	Iterator adaptor that reverses the direction of <code>const_iterator</code> .
<code>difference_type</code>	Equal to <code>typename std::iterator_traits&lt;iterator&gt;::difference_type</code> .
<code>size_type</code>	Equal to <code>size_t</code> .

Table 62. Common buffer and local accessor member functions

Member function	Description
<code>size_type byte_size() const noexcept</code>	<p>Returns the size in bytes of the memory region this accessor may access.</p> <p>For a buffer accessor this is the size of the underlying buffer, unless it is a <code>ranged_accessor</code> in which case it is the size of the elements within the accessor's range.</p> <p>For a local accessor this is the size of the accessor's local memory allocation, per work-group.</p>

Member function	Description
<code>size_type size() const noexcept</code>	<p>Returns the number of <code>DataT</code> elements of the memory region this accessor may access.</p> <p>For a buffer accessor this is the number of elements in the underlying buffer, unless it is a <code>ranged accessor</code> in which case it is the number of elements within the accessor's range.</p> <p>For a local accessor this is the number of elements in the accessor's local memory allocation, per work-group.</p>
<code>size_type max_size() const noexcept</code>	Returns the maximum number of elements any accessor of this type would be able to access.
<code>bool empty() const noexcept</code>	Returns <code>true</code> if <code>(size() == 0)</code> .
<code>range&lt;Dimensions&gt; get_range() const</code>	<p>Available only when <code>(Dimensions &gt; 0)</code>.</p> <p>Returns a <code>range</code> object which represents the number of elements of <code>DataT</code> per dimension that this accessor may access.</p> <p>For a buffer accessor this is the range of the underlying buffer, unless it is a <code>ranged accessor</code> in which case it is the range that was specified when the accessor was constructed.</p>
<code>operator reference() const</code>	<p>For <code>accessor</code> available only when <code>(AccessMode != access_mode::atomic &amp;&amp; Dimensions == 0)</code>.</p> <p>For <code>host_accessor</code> and <code>local_accessor</code> available only when <code>(Dimensions == 0)</code>.</p> <p>Returns a reference to the single element that is accessed by this accessor.</p> <p>For <code>accessor</code> and <code>local_accessor</code>, this function may only be called from within a <code>command</code>.</p>

Member function	Description
reference <code>operator[]</code> ( <code>id</code> < <code>Dimensions</code> > <code>index</code> ) <code>const</code>	<p>For <code>accessor</code> available only when <code>(AccessMode != access_mode::atomic &amp;&amp; Dimensions &gt; 0)</code>.</p> <p>For <code>host_accessor</code> and <code>local_accessor</code> available only when <code>(Dimensions &gt; 0)</code>.</p> <p>Returns a reference to the element at the location specified by <code>index</code>. If this is a <code>ranged_accessor</code>, the element is determined by adding <code>index</code> to the accessor's offset.</p> <p>For <code>accessor</code> and <code>local_accessor</code>, this function may only be called from within a <code>command</code>.</p>
<code>__unspecified__ operator[]</code> ( <code>size_t index</code> ) <code>const</code>	<p>Available only when <code>(Dimensions &gt; 1)</code>.</p> <p>Returns an instance of an undefined intermediate type representing this accessor, with the dimensionality <code>Dimensions-1</code> and containing an implicit <code>id</code> with index <code>Dimensions</code> set to <code>index</code>. The intermediate type returned must provide all available subscript operators which take a <code>size_t</code> parameter defined by this accessor class that are appropriate for the type it represents (including this subscript operator).</p> <p>If this is a <code>ranged_accessor</code>, the implicit <code>id</code> in the returned instance also includes the accessor's offset.</p> <p>For <code>accessor</code> and <code>local_accessor</code>, this function may only be called from within a <code>command</code>.</p>

Member function	Description
reference <code>operator[](size_t index) const</code>	<p>For <code>accessor</code> available only when <code>(AccessMode != access_mode::atomic &amp;&amp; Dimensions == 1)</code>.</p> <p>For <code>host_accessor</code> and <code>local_accessor</code> available only when <code>(Dimensions == 1)</code>.</p> <p>Returns a reference to the element at the location specified by <code>index</code>. If this is a <code>ranged_accessor</code>, the element is determined by adding <code>index</code> to the accessor's offset.</p> <p>For <code>accessor</code> and <code>local_accessor</code>, this function may only be called from within a <code>command</code>.</p>
iterator <code>begin() const noexcept</code>	<p>Returns an iterator to the first element of the memory this accessor may access.</p> <p>For a buffer accessor this is an iterator to the first element of the underlying buffer, unless this is a <code>ranged_accessor</code> in which case it is an iterator to first element within the accessor's range.</p> <p>For <code>accessor</code> and <code>local_accessor</code>, this function may only be called from within a <code>command</code>.</p>
iterator <code>end() const noexcept</code>	<p>Returns an iterator to one element past the last element of the memory this accessor may access.</p> <p>For a buffer accessor this is an iterator to one element past the last element in the underlying buffer, unless this is a <code>ranged_accessor</code> in which case it is an iterator to one element past the last element within the accessor's range.</p> <p>For <code>accessor</code> and <code>local_accessor</code>, this function may only be called from within a <code>command</code>.</p>

Member function	Description
<div data-bbox="137 159 987 244" style="border: 1px solid #ccc; border-radius: 5px; padding: 10px;"> <code>const_iterator cbegin() const noexcept</code> </div>	<p>Returns a const iterator to the first element of the memory this accessor may access.</p> <p>For a buffer accessor this is a const iterator to the first element of the underlying buffer, unless this is a <a href="#">ranged accessor</a> in which case it is a const iterator to first element within the accessor's range.</p> <p>For <code>accessor</code> and <code>local_accessor</code>, this function may only be called from within a <a href="#">command</a>.</p>
<div data-bbox="137 663 987 748" style="border: 1px solid #ccc; border-radius: 5px; padding: 10px;"> <code>const_iterator cend() const noexcept</code> </div>	<p>Returns a const iterator to one element past the last element of the memory this accessor may access.</p> <p>For a buffer accessor this is a const iterator to one element past the last element in the underlying buffer, unless this is a <a href="#">ranged accessor</a> in which case it is a const iterator to one element past the last element within the accessor's range.</p> <p>For <code>accessor</code> and <code>local_accessor</code>, this function may only be called from within a <a href="#">command</a>.</p>
<div data-bbox="137 1238 987 1323" style="border: 1px solid #ccc; border-radius: 5px; padding: 10px;"> <code>reverse_iterator rbegin() const noexcept</code> </div>	<p>Returns an iterator adaptor to the last element of the memory this accessor may access.</p> <p>For a buffer accessor this is an iterator adaptor to the last element of the underlying buffer, unless this is a <a href="#">ranged accessor</a> in which case it is an iterator adaptor to the last element within the accessor's range.</p> <p>For <code>accessor</code> and <code>local_accessor</code>, this function may only be called from within a <a href="#">command</a>.</p>

Member function	Description
<code>reverse_iterator rend() const noexcept</code>	<p>Returns an iterator adaptor to one element before the first element of the memory this accessor may access.</p> <p>For a buffer accessor this is an iterator adaptor to one element before the first element in the underlying buffer, unless this is a <a href="#">ranged accessor</a> in which case it is an iterator adaptor to one element before the first element within the accessor's range.</p> <p>For <code>accessor</code> and <code>local_accessor</code>, this function may only be called from within a <a href="#">command</a>.</p>
<code>const_reverse_iterator crbegin() const noexcept</code>	<p>Returns a const iterator adaptor to the last element of the memory this accessor may access.</p> <p>For a buffer accessor this is a const iterator adaptor to the last element of the underlying buffer, unless this is a <a href="#">ranged accessor</a> in which case it is a const iterator adaptor to last element within the accessor's range.</p> <p>For <code>accessor</code> and <code>local_accessor</code>, this function may only be called from within a <a href="#">command</a>.</p>
<code>const_reverse_iterator crend() const noexcept</code>	<p>Returns a const iterator adaptor to one element before the first element of the memory this accessor may access.</p> <p>For a buffer accessor this is a const iterator adaptor to one element before the first element in the underlying buffer, unless this is a <a href="#">ranged accessor</a> in which case it is a const iterator adaptor to one element before the first element within the accessor's range.</p> <p>For <code>accessor</code> and <code>local_accessor</code>, this function may only be called from within a <a href="#">command</a>.</p>

#### 4.7.6.13. Unsampld image accessors

There are two classes which implement accessors for unsampled images, `unsampled_image_accessor` and `host_unsampled_image_accessor`. The former provides access from within a [SYCL kernel function](#) or from



within a [host task](#). The latter provides access from host code that is outside of a [host task](#).

The dimensionality of an unsampled image accessor must match the dimensionality of the underlying image to which it provides access. Both unsampled image accessor classes support the `access_mode::read` and `access_mode::write` access modes. In addition, the `host_unsampled_image_accessor` class supports `access_mode::read_write`.

The `AccessTarget` template parameter dictates how the `unsampled_image_accessor` can be used: `image_target::device` means the accessor can be used in a [SYCL kernel function](#) while `image_target::host_task` means the accessor can be used in a [host task](#). Programs which specify this template parameter as `image_target::device` and then use the `unsampled_image_accessor` from a [host task](#) are ill formed. Likewise, programs which specify this template parameter as `image_target::host_task` and then use the `unsampled_image_accessor` from a [SYCL kernel function](#) are ill formed.

#### 4.7.6.13.1. Interface for unsampled image accessors

A synopsis of the two unsampled image accessor classes is provided below. Both classes have member types with the same name, which are described in [Table 63](#). The constructors for the two classes are described in [Table 64](#) and [Table 65](#). Both classes also have member functions with the same name, which are described in [Table 66](#).

The additional common special member functions and common member functions are listed in [Section 4.5.2](#) in [Table 7](#) and [Table 8](#), respectively. For valid implicit conversions between unsampled accessor types refer to [Section 4.7.6.13.2](#).

Two `unsampled_image_accessor` objects of the same type must be equality comparable in both the host code and in SYCL kernel functions. Two `host_unsampled_image_accessor` objects of the same type must be equality comparable in the host code.

```

1 namespace sycl {
2
3 enum class image_target : /* unspecified */ { device, host_task };
4
5 template <typename DataT, int Dimensions, access_mode AccessMode,
6         image_target AccessTarget = image_target::device>
7 class unsampled_image_accessor {
8 public:
9     using value_type = // const DataT for read-only accessors, DataT otherwise
10         __value_type__;
11     using reference = value_type&;
12     using const_reference = const DataT&;
13
14     template <typename AllocatorT>
15     unsampled_image_accessor(unsampled_image<Dimensions, AllocatorT>& imageRef,
16                             handler& commandGroupHandlerRef,
17                             const property_list& propList = {});
18
19     /* -- common interface members -- */
20
21     /* -- property interface members -- */
22
23     size_t size() const noexcept;
24
25     /* Available only when: AccessMode == access_mode::read
26     if Dimensions == 1, CoordT = int
27     if Dimensions == 2, CoordT = int2
28     if Dimensions == 3, CoordT = int4 */

```

```

29  template <typename CoordT> DataT read(const CoordT& coords) const noexcept;
30
31  /* Available only when: AccessMode == access_mode::write
32  if Dimensions == 1, CoordT = int
33  if Dimensions == 2, CoordT = int2
34  if Dimensions == 3, CoordT = int4 */
35  template <typename CoordT>
36  void write(const CoordT& coords, const DataT& color) const;
37 };
38
39 template <typename DataT, int Dimensions = 1,
40         access_mode AccessMode =
41         (std::is_const_v<DataT> ? access_mode::read
42          : access_mode::read_write)>
43 class host_unsampled_image_accessor {
44 public:
45     using value_type = // const DataT for read-only accessors, DataT otherwise
46         __value_type__;
47     using reference = value_type&;
48     using const_reference = const DataT&;
49
50     template <typename AllocatorT>
51     host_unsampled_image_accessor(
52         unsampled_image<Dimensions, AllocatorT>& imageRef,
53         const property_list& propList = {});
54
55     /* -- common interface members -- */
56
57     /* -- property interface members -- */
58
59     size_t size() const noexcept;
60
61     /* Available only when: (AccessMode == access_mode::read ||
62        AccessMode == access_mode::read_write)
63     if Dimensions == 1, CoordT = int
64     if Dimensions == 2, CoordT = int2
65     if Dimensions == 3, CoordT = int4 */
66     template <typename CoordT> DataT read(const CoordT& coords) const noexcept;
67
68     /* Available only when: (AccessMode == access_mode::write ||
69        AccessMode == access_mode::read_write)
70     if Dimensions == 1, CoordT = int
71     if Dimensions == 2, CoordT = int2
72     if Dimensions == 3, CoordT = int4 */
73     template <typename CoordT>
74     void write(const CoordT& coords, const DataT& color) const;
75 };
76
77 } // namespace sycl

```

Table 63. Member types of the unsampled image classes

Member types	Description
value_type	If the accessor is read-only, equal to <code>const DataT</code> , otherwise equal to <code>DataT</code> .  See <a href="#">Section 4.7.6.13.2</a> for which accessors are considered read-only.
reference	Equal to <code>value_type&amp;</code> .
const_reference	Equal to <code>const DataT&amp;</code> .

Table 64. Constructors of the `unsampled_image_accessor` class

Constructor	Description
<pre>template &lt;typename AllocatorT&gt; unsampled_image_accessor(unsampled_image&lt;Dimensions, AllocatorT&gt;&amp; imageRef,                         handler&amp; commandGroupHandlerRef,                         const property_list&amp; propList =                         {})</pre>	<p>Constructs an <code>unsampled_image_accessor</code> for accessing an <code>unsampled_image</code> within a <code>command</code> on the <code>queue</code> associated with <code>commandGroupHandlerRef</code>. The optional <code>property_list</code> provides properties for the constructed object.</p> <p>If <code>AccessTarget</code> is <code>image_target::device</code>, throws an <code>exception</code> with the <code>errc::feature_not_supported</code> error code if the device associated with <code>commandGroupHandlerRef</code> does not have <code>aspect::image</code>.</p>

Table 65. Constructors of the `host_unsampled_image_accessor` class

Constructor	Description
<pre>template &lt;typename AllocatorT&gt; host_unsampled_image_accessor(unsampled_image&lt;Dimensions, AllocatorT&gt;&amp; imageRef,                              const property_list&amp; propList                              = {})</pre>	Constructs a <code>host_unsampled_image_accessor</code> for accessing an <code>unsampled_image</code> immediately on the host. The optional <code>property_list</code> provides properties for the constructed object.

Table 66. Member functions of the `unsampled image` classes

Member function	Description
<pre>size_t size() const noexcept</pre>	Returns the number of elements of the underlying <code>unsampled_image</code> that this accessor is accessing.

Member function	Description
<pre>template &lt;typename CoordT&gt; DataT read(const CoordT&amp; coords) const</pre>	<p>Available only when <code>(AccessMode == access_mode::read    AccessMode == access_mode::read_write)</code>.</p> <p>Reads and returns an element of the <code>unsampled_image</code> at the coordinates specified by <code>coords</code>. Permitted types for <code>CoordT</code> are <code>int</code> when <code>Dimensions == 1</code>, <code>int2</code> when <code>Dimensions == 2</code> and <code>int4</code> when <code>Dimensions == 3</code>.</p> <p>For <code>unsampled_image_accessor</code>, this function may only be called from within a <code>command</code>.</p>
<pre>template &lt;typename CoordT&gt; void write(const CoordT&amp; coords, const DataT&amp; color) const</pre>	<p>Available only when <code>(AccessMode == access_mode::write    AccessMode == access_mode::read_write)</code>.</p> <p>Writes the value specified by <code>color</code> to the element of the image at the coordinates specified by <code>coords</code>. Permitted types for <code>CoordT</code> are <code>int</code> when <code>Dimensions == 1</code>, <code>int2</code> when <code>Dimensions == 2</code> and <code>int4</code> when <code>Dimensions == 3</code>.</p> <p>For <code>unsampled_image_accessor</code>, this function may only be called from within a <code>command</code>.</p>

#### 4.7.6.13.2. Read only unsampled image accessors and implicit conversions

All specializations of unsampled image accessors with `access_mode::read` are read-only regardless of whether `DataT` is `const` qualified. There is an implicit conversion between the `const` qualified and non-`const` qualified specializations, provided that all other template parameters are the same.

#### 4.7.6.14. Sampled image accessors

There are two classes which implement accessors for sampled images, `sampled_image_accessor` and `host_sampled_image_accessor`. The former provides access from within a `SYCL kernel function` or from within a `host task`. The latter provides access from host code that is outside of a `host task`.

The dimensionality of a sampled image accessor must match the dimensionality of the underlying image to which it provides access. Sampled image accessors are always read-only.

The `AccessTarget` template parameter dictates how the `sampled_image_accessor` can be used: `image_target::device` means the accessor can be used in a `SYCL kernel function` while `image_target::host_task` means the accessor can be used in a `host task`. Programs which specify this template parameter as `image_target::device` and then use the `sampled_image_accessor` from a `host task` are ill formed. Likewise, programs which specify this template parameter as `image_target::host_task` and then use the `sampled_image_accessor` from a `SYCL kernel function` are ill formed.

##### 4.7.6.14.1. Interface for sampled image accessors

A synopsis of the two sampled image accessor classes is provided below. Both classes have member

types with the same name, which are described in [Table 67](#). The constructors for the two classes are described in [Table 68](#) and [Table 69](#). Both classes also have member functions with the same name, which are described in [Table 70](#).

The additional common special member functions and common member functions are listed in [Section 4.5.2](#) in [Table 7](#) and [Table 8](#), respectively. For valid implicit conversions between sampled accessor types refer to [Section 4.7.6.14.2](#).

Two `sampld_image_accessor` objects of the same type must be equality comparable in both the host code and in SYCL kernel functions. Two `host_sampld_image_accessor` objects of the same type must be equality comparable in the host code.

```

1 namespace sycl {
2
3 enum class image_target : /* unspecified */ { device, host_task };
4
5 template <typename DataT, int Dimensions,
6         image_target AccessTarget = image_target::device>
7 class sampld_image_accessor {
8 public:
9     using value_type = const DataT;
10    using reference = const DataT&
11    using const_reference = const DataT&
12
13    template <typename AllocatorT>
14    sampld_image_accessor(sampld_image<Dimensions, AllocatorT>& imageRef,
15                        handler& commandGroupHandlerRef,
16                        const property_list& propList = {});
17
18
19    /* -- common interface members -- */
20
21    /* -- property interface members -- */
22
23    size_t size() const noexcept;
24
25    /* if Dimensions == 1, CoordT = float
26       if Dimensions == 2, CoordT = float2
27       if Dimensions == 3, CoordT = float4 */
28    template <typename CoordT> DataT read(const CoordT& coords) const noexcept;
29 };
30
31 template <typename DataT, int Dimensions> class host_sampld_image_accessor {
32 public:
33     using value_type = const DataT;
34     using reference = const DataT&
35     using const_reference = const DataT&
36
37     template <typename AllocatorT>
38     host_sampld_image_accessor(sampld_image<Dimensions, AllocatorT>& imageRef,
39                             const property_list& propList = {});
40
41
42    /* -- common interface members -- */
43
44    /* -- property interface members -- */
45

```

```

45  size_t size() const noexcept;
46
47  /* if Dimensions == 1, CoordT = float
48     if Dimensions == 2, CoordT = float2
49     if Dimensions == 3, CoordT = float4 */
50  template <typename CoordT> DataT read(const CoordT& coords) const noexcept;
51 };
52
53 } // namespace sycl

```

Table 67. Member types of the sampled image classes

Member types	Description
value_type	Equal to <code>const DataT</code> .
reference	Equal to <code>const DataT&amp;</code> .
const_reference	Equal to <code>const DataT&amp;</code> .

Table 68. Constructors of the `sampld_image_accessor` class

Constructor	Description
<pre> template &lt;typename AllocatorT&gt; sampld_image_accessor(sampld_image&lt;Dimensions, AllocatorT&gt;&amp; imageRef,                       handler&amp; commandGroupHandlerRef,                       const property_list&amp; proplist = {}) </pre>	<p>Constructs a <code>sampld_image_accessor</code> for accessing a <code>sampld_image</code> within a <code>command</code> on the <code>queue</code> associated with <code>commandGroupHandlerRef</code>. The optional <code>property_list</code> provides properties for the constructed object.</p> <p>If <code>AccessTarget</code> is <code>image_target::device</code>, throws an exception with the <code>errc::feature_not_supported</code> error code if the device associated with <code>commandGroupHandlerRef</code> does not have <code>aspect::image</code>.</p>

Table 69. Constructors of the `host_sampld_image_accessor` class

Constructor	Description
<pre> template &lt;typename AllocatorT&gt; host_sampld_image_accessor(sampld_image&lt;Dimensions, AllocatorT&gt;&amp; imageRef,                            const property_list&amp; proplist =                            {}) </pre>	<p>Constructs a <code>host_sampld_image_accessor</code> for accessing a <code>sampld_image</code> immediately on the host. The optional <code>property_list</code> provides properties for the constructed object.</p>

Table 70. Member functions of the sampled image classes

Member function	Description
<code>size_t size() const noexcept</code>	Returns the number of elements of the underlying <code>sampled_image</code> that this accessor is accessing.
<code>template &lt;typename CoordT&gt; DataT read(const CoordT&amp; coords) const</code>	<p>Reads and returns a sampled element of the <code>sampled_image</code> at the coordinates specified by <code>coords</code>. Permitted types for <code>CoordT</code> are <code>float</code> when <code>Dimensions == 1</code>, <code>float2</code> when <code>Dimensions == 2</code> and <code>float4</code> when <code>Dimensions == 3</code>.</p> <p>For <code>sampled_image_accessor</code>, this function may only be called from within a <code>command</code>.</p>

#### 4.7.6.14.2. Read only sampled image accessors and implicit conversions

All specializations of sampled image accessors are read-only regardless of whether `DataT` is `const` qualified. There is an implicit conversion between the `const` qualified and non-`const` qualified specializations, provided that all other template parameters are the same.

### 4.7.7. Address space classes

In SYCL, there are five different address spaces: global, local, constant, private and generic. In a SYCL generic implementation, types are not affected by the address spaces. However, there are situations where users need to explicitly carry address spaces in the type. For example:

- For performance tuning and genericness. Even if the platform supports the representation of the generic address space, this may come at some performance sacrifice. In order to help the target compiler, it can be useful to track specifically which address space a pointer is addressing.
- When linking SYCL kernels with `SYCL backend`-specific functions. In this case, it might be necessary to specify the address space for any pointer parameters.

Direct declaration of pointers with address spaces is discouraged as the definition is implementation-defined. Users must rely on the `multi_ptr` class to handle address space boundaries and interoperability.

#### 4.7.7.1. Multi-pointer class

The multi-pointer class is the common interface for the explicit pointer classes, defined in [Section 4.7.7.2](#).

There are situations where a user may want to make their type address space dependent. This allows performing generic programming that depends on the address space associated with their data. An example might be wrapping a pointer inside a class, where a user may need to template the class according to the address space of the pointer the class is initialized with. In this case, the `multi_ptr` class enables users to do this in a portable and stable way.

The `multi_ptr` class exposes 3 flavors of the same interface. If the value of `access::decorated` is `access::decorated::no`, the interface exposes pointers and references type that are not decorated by an address space. If the value of `access::decorated` is `access::decorated::yes`, the interface exposes pointers and references type that are decorated by an address space. The decoration is implementation dependent and relies on device compiler extensions. The decorated type may be distinct from the non-decorated one. For interoperability with the `SYCL backend`, users should rely on types exposed by the decorated version. If the value of `access::decorated` is `access::decorated::legacy`, the 1.2.1 interface is exposed.



The template traits `remove_decoration` and type alias `remove_decoration_t` retrieve the non-decorated pointer or reference from a decorated one. Using this template trait with a non-decorated type is safe and returns the same type.

It is possible to use the `void` type for the `multi_ptr` class, but in that case some functionality is disabled. `multi_ptr<void>` does not provide the `reference` or `const_reference` types, the access operators (`operator*()`, `operator->()`), the arithmetic operators or `prefetch` member function. Conversions from `multi_ptr` to `multi_ptr<void>` of the same address space are allowed, and will occur implicitly. Conversions from `multi_ptr<void>` to any other `multi_ptr` type of the same address space are allowed, but must be explicit. The same rules apply to `multi_ptr<const void>`.

An overview of the interface provided for the `multi_ptr` class follows.

```

1 namespace sycl {
2 namespace access {
3
4 enum class address_space : /* unspecified */ {
5     global_space,
6     local_space,
7     constant_space, // Deprecated in SYCL 2020
8     private_space,
9     generic_space
10 };
11
12 enum class decorated : /* unspecified */ {
13     no,
14     yes,
15     legacy
16 };
17
18 } // namespace access
19
20 template <typename T> struct remove_decoration {
21     using type = /* ... */;
22 };
23
24 template <typename T> using remove_decoration_t = remove_decoration<T>::type;
25
26 template <typename ElementType, access::address_space Space,
27         access::decorated DecorateAddress = access::decorated::legacy>
28 class multi_ptr {
29 public:
30     static constexpr bool is_decorated =
31         DecorateAddress == access::decorated::yes;
32     static constexpr access::address_space address_space = Space;
33
34     using value_type = ElementType;
35     using pointer = std::conditional_t<is_decorated, __unspecified_*,
36                                     std::add_pointer_t<value_type>>;
37     using reference = std::conditional_t<is_decorated, __unspecified_&,
38                                     std::add_lvalue_reference_t<value_type>>;
39     using iterator_category = std::random_access_iterator_tag;
40     using difference_type = std::ptrdiff_t;
41
42     static_assert(std::is_same_v<remove_decoration_t<pointer>,
43                             std::add_pointer_t<value_type>>);

```



```

44 static_assert(std::is_same_v<remove_decoration_t<reference>,
45               std::add_lvalue_reference_t<value_type>>);
46 // Legacy has a different interface.
47 static_assert(DecorateAddress != access::decorated::legacy);
48
49 // Constructors
50 multi_ptr();
51 multi_ptr(const multi_ptr&);
52 multi_ptr(multi_ptr&&);
53 explicit multi_ptr(
54     typename multi_ptr<ElementType, Space, access::decorated::yes>::pointer);
55 multi_ptr(std::nullptr_t);
56
57 // Available only when:
58 // (Space == access::address_space::global_space ||
59 //  Space == access::address_space::generic_space) &&
60 // (std::is_same_v<std::remove_const_t<ElementType>, std::remove_const_t<AccDataT>>) &&
61 // (std::is_const_v<ElementType> ||
62 //  !std::is_const_v<accessor<AccDataT, Dimensions, Mode, target::device,
63 //                      IsPlaceholder>::value_type>)
64 template <typename AccDataT, int Dimensions, access_mode Mode,
65           access::placeholder IsPlaceholder>
66 multi_ptr(
67     accessor<AccDataT, Dimensions, Mode, target::device, IsPlaceholder>);
68
69 // Available only when:
70 // (Space == access::address_space::local_space ||
71 //  Space == access::address_space::generic_space) &&
72 // (std::is_same_v<std::remove_const_t<ElementType>, std::remove_const_t<AccDataT>>) &&
73 // (std::is_const_v<ElementType> || !std::is_const_v<AccDataT>)
74 template <typename AccDataT, int Dimensions>
75 multi_ptr(local_accessor<AccDataT, Dimensions>);
76
77 // Deprecated
78 // Available only when:
79 // (Space == access::address_space::local_space ||
80 //  Space == access::address_space::generic_space) &&
81 // (std::is_same_v<std::remove_const_t<ElementType>, std::remove_const_t<AccDataT>>) &&
82 // (std::is_const_v<ElementType> || !std::is_const_v<AccDataT>)
83 template <typename AccDataT, int Dimensions, access_mode Mode,
84           access::placeholder IsPlaceholder>
85 multi_ptr(
86     accessor<AccDataT, Dimensions, Mode, target::local, IsPlaceholder>);
87
88 // Deprecated
89 // Available only when:
90 // Space == access::address_space::constant_space &&
91 // (std::is_same_v<std::remove_const_t<ElementType>, std::remove_const_t<AccDataT>>) &&
92 // (std::is_const_v<ElementType> || !std::is_const_v<AccDataT>)
93 template <typename AccDataT, int Dimensions, access::placeholder IsPlaceholder>
94 multi_ptr(
95     accessor<AccDataT, Dimensions, access_mode::read, target::constant_buffer,
96     IsPlaceholder>);
97 // Assignment and access operators
98 multi_ptr& operator=(const multi_ptr&);

```

```

99  multi_ptr& operator=(multi_ptr&&);
100 multi_ptr& operator=(std::nullptr_t);
101
102 // Available only when:
103 //   (Space == access::address_space::generic_space &&
104 //    AS != access::address_space::constant_space)
105 template <access::address_space AS, access::decorated IsDecorated>
106 multi_ptr& operator=(const multi_ptr<value_type, AS, IsDecorated>&);
107
108 // Available only when:
109 //   (Space == access::address_space::generic_space &&
110 //    AS != access::address_space::constant_space)
111 template <access::address_space AS, access::decorated IsDecorated>
112 multi_ptr& operator=(multi_ptr<value_type, AS, IsDecorated>&&);
113
114 reference operator[](std::ptrdiff_t) const;
115
116 reference operator*() const;
117 pointer operator->() const;
118
119 pointer get() const;
120 std::add_pointer_t<value_type> get_raw() const;
121 __unspecified__* get_decorated() const;
122
123 // Conversion to the underlying pointer type
124 // Deprecated, get() should be used instead.
125 operator pointer() const;
126
127 // Cast to private_ptr
128 // Available only when: (Space == access::address_space::generic_space)
129 template <access::decorated IsDecorated>
130 explicit operator multi_ptr<value_type, access::address_space::private_space,
131                             IsDecorated>() const;
132
133 // Cast to private_ptr of const data
134 // Available only when: (Space == access::address_space::generic_space)
135 template <access::decorated IsDecorated>
136 explicit operator multi_ptr<const value_type, access::address_space::private_space,
137                             IsDecorated>() const;
138
139 // Cast to global_ptr
140 // Available only when: (Space == access::address_space::generic_space)
141 template <access::decorated IsDecorated>
142 explicit operator multi_ptr<value_type, access::address_space::global_space,
143                             IsDecorated>() const;
144
145 // Cast to global_ptr of const data
146 // Available only when: (Space == access::address_space::generic_space)
147 template <access::decorated IsDecorated>
148 explicit operator multi_ptr<const value_type, access::address_space::global_space,
149                             IsDecorated>() const;
150
151 // Cast to local_ptr
152 // Available only when: (Space == access::address_space::generic_space)
153 template <access::decorated IsDecorated>
154 explicit operator multi_ptr<value_type, access::address_space::local_space,

```

```

155         IsDecorated>() const;
156
157     // Cast to local_ptr of const data
158     // Available only when: (Space == access::address_space::generic_space)
159     template <access::decorated IsDecorated>
160     explicit operator multi_ptr<const value_type, access::address_space::local_space,
161         IsDecorated>() const;
162
163     // Implicit conversion to a multi_ptr<void>.
164     // Available only when: (!std::is_const_v<value_type>)
165     template <access::decorated IsDecorated>
166     operator multi_ptr<void, Space, IsDecorated>() const;
167
168     // Implicit conversion to a multi_ptr<const void>.
169     // Available only when: (std::is_const_v<value_type>)
170     template <access::decorated IsDecorated>
171     operator multi_ptr<const void, Space, IsDecorated>() const;
172
173     // Implicit conversion to multi_ptr<const value_type, Space>.
174     template <access::decorated IsDecorated>
175     operator multi_ptr<const value_type, Space, IsDecorated>() const;
176
177     // Implicit conversion to the non-decorated version of multi_ptr.
178     // Available only when: (is_decorated == true)
179     operator multi_ptr<value_type, Space, access::decorated::no>() const;
180
181     // Implicit conversion to the decorated version of multi_ptr.
182     // Available only when: (is_decorated == false)
183     operator multi_ptr<value_type, Space, access::decorated::yes>() const;
184
185     // Available only when: (Space == address_space::global_space)
186     void prefetch(size_t numElements) const;
187
188     // Arithmetic operators
189     friend multi_ptr& operator++(multi_ptr& mp) { /* ... */
190     }
191     friend multi_ptr operator++(multi_ptr& mp, int) { /* ... */
192     }
193     friend multi_ptr& operator--(multi_ptr& mp) { /* ... */
194     }
195     friend multi_ptr operator--(multi_ptr& mp, int) { /* ... */
196     }
197     friend multi_ptr& operator+=(multi_ptr& lhs, difference_type r) { /* ... */
198     }
199     friend multi_ptr& operator-=(multi_ptr& lhs, difference_type r) { /* ... */
200     }
201     friend multi_ptr operator+(const multi_ptr& lhs,
202         difference_type r) { /* ... */
203     }
204     friend multi_ptr operator-(const multi_ptr& lhs,
205         difference_type r) { /* ... */
206     }
207     friend reference operator*(const multi_ptr& lhs) { /* ... */
208     }
209
210     friend bool operator==(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */

```

```

211 }
212 friend bool operator!=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
213 }
214 friend bool operator<(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
215 }
216 friend bool operator>(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
217 }
218 friend bool operator<=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
219 }
220 friend bool operator>=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
221 }
222
223 friend bool operator==(const multi_ptr& lhs, std::nullptr_t) { /* ... */
224 }
225 friend bool operator!=(const multi_ptr& lhs, std::nullptr_t) { /* ... */
226 }
227 friend bool operator<(const multi_ptr& lhs, std::nullptr_t) { /* ... */
228 }
229 friend bool operator>(const multi_ptr& lhs, std::nullptr_t) { /* ... */
230 }
231 friend bool operator<=(const multi_ptr& lhs, std::nullptr_t) { /* ... */
232 }
233 friend bool operator>=(const multi_ptr& lhs, std::nullptr_t) { /* ... */
234 }
235
236 friend bool operator==(std::nullptr_t, const multi_ptr& rhs) { /* ... */
237 }
238 friend bool operator!=(std::nullptr_t, const multi_ptr& rhs) { /* ... */
239 }
240 friend bool operator<(std::nullptr_t, const multi_ptr& rhs) { /* ... */
241 }
242 friend bool operator>(std::nullptr_t, const multi_ptr& rhs) { /* ... */
243 }
244 friend bool operator<=(std::nullptr_t, const multi_ptr& rhs) { /* ... */
245 }
246 friend bool operator>=(std::nullptr_t, const multi_ptr& rhs) { /* ... */
247 }
248 };
249
250 // Specialization of multi_ptr for void and const void
251 // VoidType can be either void or const void
252 template <access::address_space Space, access::decorated DecorateAddress>
253 class multi_ptr<VoidType, Space, DecorateAddress> {
254 public:
255     static constexpr bool is_decorated =
256         DecorateAddress == access::decorated::yes;
257     static constexpr access::address_space address_space = Space;
258
259     using value_type = VoidType;
260     using pointer = std::conditional_t<is_decorated, __unspecified__*,
261                                         std::add_pointer_t<value_type>>;
262     using difference_type = std::ptrdiff_t;
263
264     static_assert(std::is_same_v<remove_decoration_t<pointer>,
265                               std::add_pointer_t<value_type>>);
266     // Legacy has a different interface.

```

```

267 static_assert(DecorateAddress != access::decorated::legacy);
268
269 // Constructors
270 multi_ptr();
271 multi_ptr(const multi_ptr&);
272 multi_ptr(multi_ptr&&);
273 explicit multi_ptr(
274     typename multi_ptr<VoidType, Space, access::decorated::yes>::pointer);
275 multi_ptr(std::nullptr_t);
276
277 // Available only when:
278 // (Space == access::address_space::global_space ||
279 //  Space == access::address_space::generic_space) &&
280 // (std::is_const_v<VoidType> ||
281 //  !std::is_const_v<accessor<ElementType, Dimensions, Mode, target::device,
282 //  IsPlaceholder>::value_type>)
283 template <typename ElementType, int Dimensions, access_mode Mode,
284           access::placeholder IsPlaceholder>
285 multi_ptr(
286     accessor<ElementType, Dimensions, Mode, target::device, IsPlaceholder>);
287
288 // Available only when:
289 // (Space == access::address_space::local_space ||
290 //  Space == access::address_space::generic_space) &&
291 // (std::is_const_v<VoidType> || !std::is_const_v<ElementType>)
292 template <typename ElementType, int Dimensions>
293 multi_ptr(local_accessor<ElementType, Dimensions>);
294
295 // Deprecated
296 // Available only when:
297 // (Space == access::address_space::local_space ||
298 //  Space == access::address_space::generic_space) &&
299 // (std::is_const_v<VoidType> || !std::is_const_v<ElementType>)
300 template <typename ElementType, int Dimensions, access_mode Mode,
301           access::placeholder IsPlaceholder>
302 multi_ptr(
303     accessor<ElementType, Dimensions, Mode, target::local, IsPlaceholder>);
304
305 // Deprecated
306 // Available only when:
307 // Space == access::address_space::constant_space &&
308 // (std::is_const_v<VoidType> || !std::is_const_v<ElementType>)
309 template <typename ElementType, int Dimensions, access::placeholder IsPlaceholder>
310 multi_ptr(
311     accessor<ElementType, Dimensions, access_mode::read, target::constant_buffer,
312     IsPlaceholder>);
313
314 // Assignment operators
315 multi_ptr& operator=(const multi_ptr&);
316 multi_ptr& operator=(multi_ptr&&);
317 multi_ptr& operator=(std::nullptr_t);
318
319 pointer get() const;
320
321 // Conversion to the underlying pointer type
322 operator pointer() const;

```

```

322
323 // Explicit conversion to a multi_ptr<ElementType>
324 // Available only when: (std::is_const_v<ElementType> || !std::is_const_v<VoidType>)
325 template <typename ElementType>
326 explicit operator multi_ptr<ElementType, Space, DecorateAddress>() const;
327
328 // Implicit conversion to the non-decorated version of multi_ptr.
329 // Available only when: (is_decorated == true)
330 operator multi_ptr<value_type, Space, access::decorated::no>() const;
331
332 // Implicit conversion to the decorated version of multi_ptr.
333 // Available only when: (is_decorated == false)
334 operator multi_ptr<value_type, Space, access::decorated::yes>() const;
335
336 // Implicit conversion to multi_ptr<const void, Space>
337 operator multi_ptr<const void, Space, DecorateAddress>() const;
338
339 friend bool operator==(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
340 }
341 friend bool operator!=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
342 }
343 friend bool operator<(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
344 }
345 friend bool operator>(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
346 }
347 friend bool operator<=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
348 }
349 friend bool operator>=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
350 }
351
352 friend bool operator==(const multi_ptr& lhs, std::nullptr_t) { /* ... */
353 }
354 friend bool operator!=(const multi_ptr& lhs, std::nullptr_t) { /* ... */
355 }
356 friend bool operator<(const multi_ptr& lhs, std::nullptr_t) { /* ... */
357 }
358 friend bool operator>(const multi_ptr& lhs, std::nullptr_t) { /* ... */
359 }
360 friend bool operator<=(const multi_ptr& lhs, std::nullptr_t) { /* ... */
361 }
362 friend bool operator>=(const multi_ptr& lhs, std::nullptr_t) { /* ... */
363 }
364
365 friend bool operator==(std::nullptr_t, const multi_ptr& rhs) { /* ... */
366 }
367 friend bool operator!=(std::nullptr_t, const multi_ptr& rhs) { /* ... */
368 }
369 friend bool operator<(std::nullptr_t, const multi_ptr& rhs) { /* ... */
370 }
371 friend bool operator>(std::nullptr_t, const multi_ptr& rhs) { /* ... */
372 }
373 friend bool operator<=(std::nullptr_t, const multi_ptr& rhs) { /* ... */
374 }
375 friend bool operator>=(std::nullptr_t, const multi_ptr& rhs) { /* ... */
376 }
377 };

```

```

378
379 // Deprecated, address_space_cast should be used instead.
380 template <typename ElementType, access::address_space Space,
381          access::decorated DecorateAddress>
382 multi_ptr<ElementType, Space, DecorateAddress> make_ptr(ElementType*);
383
384 template <access::address_space Space, access::decorated DecorateAddress,
385          typename ElementType>
386 multi_ptr<ElementType, Space, DecorateAddress> address_space_cast(ElementType*);
387
388 // Deduction guides
389 template <typename T, int Dimensions, access::placeholder IsPlaceholder>
390 multi_ptr(accessor<T, Dimensions, access_mode::read, target::device, IsPlaceholder>)
391     -> multi_ptr<const T, access::address_space::global_space, access::decorated::no>;
392
393 template <typename T, int Dimensions, access::placeholder IsPlaceholder>
394 multi_ptr(accessor<T, Dimensions, access_mode::write, target::device, IsPlaceholder>)
395     -> multi_ptr<T, access::address_space::global_space, access::decorated::no>;
396
397 template <typename T, int Dimensions, access::placeholder IsPlaceholder>
398 multi_ptr(accessor<T, Dimensions, access_mode::read_write, target::device, IsPlaceholder>)
399     -> multi_ptr<T, access::address_space::global_space, access::decorated::no>;
400
401 template <typename T, int Dimensions, access::placeholder IsPlaceholder>
402 multi_ptr(accessor<T, Dimensions, access_mode::read, target::constant_buffer,
403           IsPlaceholder>)
404     -> multi_ptr<const T, access::address_space::constant_space, access::decorated::no>;
405
406 template <typename T, int Dimensions, access_mode Mode, access::placeholder IsPlaceholder>
407 multi_ptr(accessor<T, Dimensions, Mode, target::local, IsPlaceholder>)
408     -> multi_ptr<T, access::address_space::local_space, access::decorated::no>;
409
410 template <typename T, int Dimensions>
411 multi_ptr(local_accessor<T, Dimensions>)
412     -> multi_ptr<T, access::address_space::local_space, access::decorated::no>;
413 } // namespace sycl

```

Table 71. Constructors of the SYCL `multi_ptr` class template

Constructor	Description
<code>multi_ptr()</code>	Default constructor.
<code>multi_ptr(const multi_ptr&amp;)</code>	Copy constructor.
<code>multi_ptr(multi_ptr&amp;&amp;)</code>	Move constructor.
<code>explicit multi_ptr(multi_ptr&lt;ElementType, Space,           access::decorated::yes&gt;::pointer)</code>	Constructor that takes as an argument a decorated pointer.

Constructor	Description
<code>multi_ptr(std::nullptr_t)</code>	Constructor from a <code>nullptr</code> .
<pre>template &lt;typename AccDataT, int Dimensions,           access_mode Mode,           access::placeholder IsPlaceholder&gt; multi_ptr(accessor&lt;AccDataT, Dimensions, Mode,                   target::device, IsPlaceholder&gt;)</pre>	<p>Available only when: <code>(Space == access::address_space::global_space    Space == access::address_space::generic_space) &amp;&amp; (std::is_void_v&lt;ElementType&gt;    std::is_same_v&lt;std::remove_const_t&lt;ElementType&gt;, std::remove_const_t&lt;AccDataT&gt;&gt;) &amp;&amp; (std::is_const_v&lt;ElementType&gt;    !std::is_const_v&lt;accessor&lt;AccDataT, Dimensions, Mode, target::device, IsPlaceholder&gt;::value_type)</code>.</p> <p>Constructs a <code>multi_ptr</code> from an accessor of <code>target::device</code>.</p> <p>This constructor may only be called from within a <code>command</code>.</p>
<pre>template &lt;typename AccDataT, int Dimensions&gt; multi_ptr(local_accessor&lt;AccDataT, Dimensions&gt;)</pre>	<p>Available only when: <code>(Space == access::address_space::local_space    Space == access::address_space::generic_space) &amp;&amp; (std::is_void_v&lt;ElementType&gt;    std::is_same_v&lt;std::remove_const_t&lt;ElementType&gt;, std::remove_const_t&lt;AccDataT&gt;&gt;) &amp;&amp; (std::is_const_v&lt;ElementType&gt;    !std::is_const_v&lt;AccDataT&gt;)</code>.</p> <p>Constructs a <code>multi_ptr</code> from a <code>local_accessor</code>.</p> <p>This constructor may only be called from within a <code>command</code>.</p>



Constructor	Description
<pre>template &lt;typename AccDataT, int Dimensions,           access_mode Mode,           access::placeholder IsPlaceholder&gt; multi_ptr(accessor&lt;AccDataT, Dimensions, Mode,                   target::local, IsPlaceholder&gt;)</pre>	<p>Deprecated in SYCL 2020. Use the overload with <code>local_accessor</code> instead.</p> <p>Available only when: <code>(Space == access::address_space::local_space    Space == access::address_space::generic_space) &amp;&amp; (std::is_void_v&lt;ElementType&gt;    std::is_same_v&lt;std::remove_const_t&lt;ElementType&gt;, std::remove_const_t&lt;AccDataT&gt;&gt;) &amp;&amp; (std::is_const_v&lt;ElementType&gt;    !std::is_const_v&lt;AccDataT&gt;)</code>.</p> <p>Constructs a <code>multi_ptr</code> from an accessor of <code>target::local</code>.</p> <p>This constructor may only be called from within a <code>command</code>.</p>
<pre>template &lt;typename ElementType,           access::address_space Space,           access::decorated DecorateAddress&gt; multi_ptr&lt;ElementType, Space, DecorateAddress&gt; make_ptr(ElementType* pointer)</pre>	<p>Deprecated in SYCL 2020. Use <code>address_space_cast</code> instead.</p> <p>Global function to create a <code>multi_ptr</code> instance depending on the address space of the <code>pointer</code> argument. An implementation must return <code>nullptr</code> if the run-time value of <code>pointer</code> is not compatible with <code>Space</code>, and must issue a compile-time diagnostic if the deduced address space is not compatible with <code>Space</code>.</p>
<pre>template &lt;access::address_space Space,           access::decorated DecorateAddress,           typename ElementType&gt; multi_ptr&lt;ElementType, Space, DecorateAddress&gt; address_space_cast(ElementType* pointer)</pre>	<p>Global function to create a <code>multi_ptr</code> instance from <code>pointer</code>, using the address space and decoration specified via the <code>Space</code> and <code>DecorateAddress</code> template arguments.</p> <p>An implementation must return <code>nullptr</code> if the run-time value of <code>pointer</code> is not compatible with <code>Space</code>, and must issue a compile-time diagnostic if the deduced address space for <code>pointer</code> is not compatible with <code>Space</code>.</p>

Table 72. Operators of `multi_ptr` class

Operators	Description
<pre>multi_ptr&amp; operator=(const multi_ptr&amp;)</pre>	Copy assignment operator.

Operators	Description
<code>multi_ptr&amp; operator=(multi_ptr&amp;&amp;)</code>	Move assignment operator.
<code>multi_ptr&amp; operator=(std::nullptr_t)</code>	Assigns <code>nullptr</code> to the <code>multi_ptr</code> .
<pre>template &lt;access::address_space AS,           access::decorated IsDecorated&gt; multi_ptr&amp; operator=(const multi_ptr&lt;value_type, AS, IsDecorated&gt;&amp;)</pre>	<p>Available only when: <code>(Space == access::address_space::generic_space &amp;&amp; AS != access::address_space::constant_space)</code>.</p> <p>Assigns the value of the right hand side <code>multi_ptr</code> into the <code>generic_ptr</code>.</p>
<pre>template&lt;access::address_space AS,          access::decorated IsDecorated&gt; multi_ptr&amp; operator=(multi_ptr&lt;value_type, AS, IsDecorated&gt;&amp;&amp;)</pre>	<p>Available only when: <code>(Space == access::address_space::generic_space &amp;&amp; AS != access::address_space::constant_space)</code>.</p> <p>Move the value of the right hand side <code>multi_ptr</code> into the <code>generic_ptr</code>.</p>
<code>reference operator[](std::ptrdiff_t i) const</code>	<p>Available only when: <code>(!std::is_void_v&lt;value_type&gt;)</code>.</p> <p>Returns a reference to the <i>i</i>-th pointed value. The value <i>i</i> can be negative.</p>
<code>pointer operator-&gt;() const</code>	<p>Available only when: <code>(!std::is_void_v&lt;value_type&gt;)</code>.</p> <p>Returns the underlying pointer.</p>
<code>reference operator*() const</code>	<p>Available only when: <code>(!std::is_void_v&lt;value_type&gt;)</code>.</p> <p>Returns a reference to the pointed value.</p>
<code>operator pointer() const</code>	<p>Implicit conversion to the underlying pointer type. <b>Deprecated:</b> The member function <code>get</code> should be used instead</p>
<pre>template &lt;access::decorated IsDecorated&gt; explicit operator multi_ptr&lt;value_type,                   access::address_space::private_space,                   IsDecorated&gt;() const</pre>	<p>Available only when: <code>(Space == access::address_space::generic_space)</code>.</p> <p>Conversion from <code>generic_ptr</code> to <code>private_ptr</code>. The result is undefined if the pointer does not address the private address space.</p>

Operators	Description
<pre>template &lt;access::decorated IsDecorated&gt; explicit operator multi_ptr&lt;const value_type,                   access::address_space::private_space,                   IsDecorated&gt;() const</pre>	<p>Available only when: (<code>Space == access::address_space::generic_space</code>).</p> <p>Conversion from <code>generic_ptr</code> to <code>private_ptr</code> of const data. The result is undefined if the pointer does not address the private address space.</p>
<pre>template &lt;access::decorated IsDecorated&gt; explicit operator multi_ptr&lt;value_type,                   access::address_space::global_space,                   IsDecorated&gt;() const</pre>	<p>Available only when: (<code>Space == access::address_space::generic_space</code>).</p> <p>Conversion from <code>generic_ptr</code> to <code>global_ptr</code>. The result is undefined if the pointer does not address the global address space.</p>
<pre>template &lt;access::decorated IsDecorated&gt; explicit operator multi_ptr&lt;const value_type,                   access::address_space::global_space,                   IsDecorated&gt;() const</pre>	<p>Available only when: (<code>Space == access::address_space::generic_space</code>).</p> <p>Conversion from <code>generic_ptr</code> to <code>global_ptr</code> of const data. The result is undefined if the pointer does not address the global address space.</p>
<pre>template &lt;access::decorated IsDecorated&gt; explicit operator multi_ptr&lt;value_type,                   access::address_space::local_space,                   IsDecorated&gt;() const</pre>	<p>Available only when: (<code>Space == access::address_space::generic_space</code>).</p> <p>Conversion from <code>generic_ptr</code> to <code>local_ptr</code>. The result is undefined if the pointer does not address the local address space.</p>
<pre>template &lt;access::decorated IsDecorated&gt; explicit operator multi_ptr&lt;const value_type,                   access::address_space::local_space,                   IsDecorated&gt;() const</pre>	<p>Available only when: (<code>Space == access::address_space::generic_space</code>).</p> <p>Conversion from <code>generic_ptr</code> to <code>local_ptr</code> of const data. The result is undefined if the pointer does not address the local address space.</p>
<pre>template &lt;access::decorated IsDecorated&gt; operator multi_ptr&lt;void, Space, IsDecorated&gt;() const</pre>	<p>Available only when: (<code>!std::is_void_v&lt;value_type&gt; &amp;&amp; !std::is_const_v&lt;value_type&gt;</code>).</p> <p>Implicit conversion to a <code>multi_ptr</code> of type <code>void</code>.</p>
<pre>template &lt;access::decorated IsDecorated&gt; operator multi_ptr&lt;const void, Space, IsDecorated&gt;() const</pre>	<p>Available only when: (<code>!std::is_void_v&lt;value_type&gt; &amp;&amp; std::is_const_v&lt;value_type&gt;</code>).</p> <p>Implicit conversion to a <code>multi_ptr</code> of type <code>const void</code>.</p>

Operators	Description
<pre>template &lt;access::decorated IsDecorated&gt; operator multi_ptr&lt;const value_type, Space,                   IsDecorated&gt;() const</pre>	Implicit conversion to a <code>multi_ptr</code> of type <code>const value_type</code> .
<pre>operator multi_ptr&lt;value_type, Space,                   access::decorated::no&gt;() const</pre>	Available only when: ( <code>is_decorated == true</code> ).  Implicit conversion to the equivalent <code>multi_ptr</code> object that does not expose decorated pointers or references.
<pre>operator multi_ptr&lt;value_type, Space,                   access::decorated::yes&gt;() const</pre>	Available only when: ( <code>is_decorated == false</code> ).  Implicit conversion to the equivalent <code>multi_ptr</code> object that exposes decorated pointers and references.

Table 73. Member functions of `multi_ptr` class

Member function	Description
<pre>pointer get() const</pre>	Returns the underlying pointer. Whether the pointer is decorated depends on the value of <code>DecorateAddress</code> .
<pre>__unspecified__* get_decorated() const</pre>	Returns the underlying pointer decorated by the address space that it addresses. Note that the support involves implementation-defined device compiler extensions.
<pre>std::add_pointer_t&lt;value_type&gt; get_raw() const</pre>	Returns the underlying pointer, always undecorated.
<pre>void prefetch(size_t numElements) const</pre>	Available only when: <code>Space == access::address_space::global_space</code> .  Prefetches a number of elements specified by <code>numElements</code> into the <code>global memory</code> cache. This operation is an implementation-defined optimization and does not effect the functional behavior of the SYCL kernel function.

Table 74. Hidden friend functions of the `multi_ptr` class

Hidden friend function	Description
reference <code>operator*(const multi_ptr&amp; mp)</code>	Available only when: <code>(!std::is_void_v&lt;ElementType&gt;)</code> . Operator that returns a reference to the <code>value_type</code> of <code>mp</code> .
<code>multi_ptr&amp; operator++(multi_ptr&amp; mp)</code>	Available only when: <code>(!std::is_void_v&lt;ElementType&gt;)</code> . Increments <code>mp</code> by 1 and returns <code>mp</code> .
<code>multi_ptr operator++(multi_ptr&amp; mp, int)</code>	Available only when: <code>(!std::is_void_v&lt;ElementType&gt;)</code> . Increments <code>mp</code> by 1 and returns a new <code>multi_ptr</code> with the value of the original <code>mp</code> .
<code>multi_ptr&amp; operator--(multi_ptr&amp; mp)</code>	Available only when: <code>(!std::is_void_v&lt;ElementType&gt;)</code> . Decrements <code>mp</code> by 1 and returns <code>mp</code> .
<code>multi_ptr operator--(multi_ptr&amp; mp, int)</code>	Available only when: <code>(!std::is_void_v&lt;ElementType&gt;)</code> . Decrements <code>mp</code> by 1 and returns a new <code>multi_ptr</code> with the value of the original <code>mp</code> .
<code>multi_ptr&amp; operator+=(multi_ptr&amp; lhs, difference_type r)</code>	Available only when: <code>(!std::is_void_v&lt;ElementType&gt;)</code> . Moves <code>mp</code> forward by <code>r</code> and returns <code>lhs</code> .
<code>multi_ptr&amp; operator-=(multi_ptr&amp; lhs, difference_type r)</code>	Available only when: <code>(!std::is_void_v&lt;ElementType&gt;)</code> . Moves <code>mp</code> backward by <code>r</code> and returns <code>lhs</code> .
<code>multi_ptr operator+(const multi_ptr&amp; lhs, difference_type r)</code>	Available only when: <code>(!std::is_void_v&lt;ElementType&gt;)</code> . Creates a new <code>multi_ptr</code> that points <code>r</code> forward compared to <code>lhs</code> .
<code>multi_ptr operator-(const multi_ptr&amp; lhs, difference_type r)</code>	Available only when: <code>(!std::is_void_v&lt;ElementType&gt;)</code> . Creates a new <code>multi_ptr</code> that points <code>r</code> backward compared to <code>lhs</code> .
<code>bool operator==(const multi_ptr&amp; lhs, const multi_ptr&amp; rhs)</code>	Comparison operator <code>==</code> for <code>multi_ptr</code> class.
<code>bool operator!=(const multi_ptr&amp; lhs, const multi_ptr&amp; rhs)</code>	Comparison operator <code>!=</code> for <code>multi_ptr</code> class.

Hidden friend function	Description
<code>bool operator&lt;(const multi_ptr&amp; lhs, const multi_ptr&amp; rhs)</code>	Comparison operator <code>&lt;</code> for <code>multi_ptr</code> class.
<code>bool operator&gt;(const multi_ptr&amp; lhs, const multi_ptr&amp; rhs)</code>	Comparison operator <code>&gt;</code> for <code>multi_ptr</code> class.
<code>bool operator&lt;=(const multi_ptr&amp; lhs, const multi_ptr&amp; rhs)</code>	Comparison operator <code>&lt;=</code> for <code>multi_ptr</code> class.
<code>bool operator&gt;=(const multi_ptr&amp; lhs, const multi_ptr&amp; rhs)</code>	Comparison operator <code>&gt;=</code> for <code>multi_ptr</code> class.
<code>bool operator==(const multi_ptr&amp; lhs, std::nullptr_t)</code>	Comparison operator <code>==</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code> .
<code>bool operator!=(const multi_ptr&amp; lhs, std::nullptr_t)</code>	Comparison operator <code>!=</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code> .
<code>bool operator&lt;(const multi_ptr&amp; lhs, std::nullptr_t)</code>	Comparison operator <code>&lt;</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code> .
<code>bool operator&gt;(const multi_ptr&amp; lhs, std::nullptr_t)</code>	Comparison operator <code>&gt;</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code> .
<code>bool operator&lt;=(const multi_ptr&amp; lhs, std::nullptr_t)</code>	Comparison operator <code>&lt;=</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code> .
<code>bool operator&gt;=(const multi_ptr&amp; lhs, std::nullptr_t)</code>	Comparison operator <code>&gt;=</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code> .
<code>bool operator==(std::nullptr_t, const multi_ptr&amp; rhs)</code>	Comparison operator <code>==</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code> .
<code>bool operator!=(std::nullptr_t, const multi_ptr&amp; rhs)</code>	Comparison operator <code>!=</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code> .
<code>bool operator&lt;(std::nullptr_t, const multi_ptr&amp; rhs)</code>	Comparison operator <code>&lt;</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code> .
<code>bool operator&gt;(std::nullptr_t, const multi_ptr&amp; rhs)</code>	Comparison operator <code>&gt;</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code> .
<code>bool operator&lt;=(std::nullptr_t, const multi_ptr&amp; rhs)</code>	Comparison operator <code>&lt;=</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code> .
<code>bool operator&gt;=(std::nullptr_t, const multi_ptr&amp; rhs)</code>	Comparison operator <code>&gt;=</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code> .

The following is the overview of the legacy interface from 1.2.1 provided for the `multi_ptr` class.

```

1 namespace sycl {
2
3 // Legacy interface, inherited from 1.2.1.
4 template <typename ElementType, access::address_space Space>

```

```

5 class [[deprecated]] multi_ptr<ElementType, Space, access::decorated::legacy> {
6 public:
7     using value_type = ElementType;
8     using element_type = ElementType;
9     using difference_type = std::ptrdiff_t;
10
11     // Implementation defined pointer and reference types that correspond to
12     // SYCL/OpenCL interoperability types for OpenCL C functions.
13     using pointer_t =
14         multi_ptr<ElementType, Space, access::decorated::yes::pointer;
15     using const_pointer_t =
16         multi_ptr<const ElementType, Space, access::decorated::yes::pointer;
17     using reference_t =
18         multi_ptr<ElementType, Space, access::decorated::yes::reference;
19     using const_reference_t =
20         multi_ptr<const ElementType, Space, access::decorated::yes::reference;
21
22     static constexpr access::address_space address_space = Space;
23
24     // Constructors
25     multi_ptr();
26     multi_ptr(const multi_ptr&);
27     multi_ptr(multi_ptr&&);
28     multi_ptr(pointer_t);
29     multi_ptr(ElementType*);
30     multi_ptr(std::nullptr_t);
31     ~multi_ptr();
32
33     // Assignment and access operators
34     multi_ptr& operator=(const multi_ptr&);
35     multi_ptr& operator=(multi_ptr&&);
36     multi_ptr& operator=(pointer_t);
37     multi_ptr& operator=(ElementType*);
38     multi_ptr& operator=(std::nullptr_t);
39     friend ElementType& operator*(const multi_ptr& mp) { /* ... */
40     }
41     ElementType* operator->() const;
42
43     // Available only when:
44     // (Space == access::address_space::global_space ||
45     //  Space == access::address_space::generic_space) &&
46     // (std::is_same_v<std::remove_const_t<ElementType>, std::remove_const_t<AccDataT>>) &&
47     // (std::is_const_v<ElementType> ||
48     //  !std::is_const_v<accessor<AccDataT, Dimensions, Mode, target::device,
49     //  IsPlaceholder>::value_type>)
50     template <int Dimensions, access_mode Mode, access::placeholder IsPlaceholder>
51     multi_ptr(
52         accessor<ElementType, Dimensions, Mode, target::device, IsPlaceholder>);
53
54     // Available only when:
55     // (Space == access::address_space::local_space ||
56     //  Space == access::address_space::generic_space) &&
57     // (std::is_same_v<std::remove_const_t<ElementType>, std::remove_const_t<AccDataT>>) &&
58     // (std::is_const_v<ElementType> || !std::is_const_v<AccDataT>)
59     template <int Dimensions, access_mode Mode, access::placeholder IsPlaceholder>
60     multi_ptr(

```

```

61     accessor<ElementType, Dimensions, Mode, target::local, IsPlaceholder>;
62
63     // Available only when:
64     //   (Space == access::address_space::local_space ||
65     //    Space == access::address_space::generic_space) &&
66     //    (std::is_same_v<std::remove_const_t<ElementType>, std::remove_const_t<AccDataT>>) &&
67     //    (std::is_const_v<ElementType> || !std::is_const_v<AccDataT>)
68     template <typename AccDataT, int Dimensions>
69     multi_ptr(local_accessor<AccDataT, Dimensions>);
70
71     // Only if Space == constant_space
72     template <int Dimensions, access_mode Mode, access::placeholder IsPlaceholder>
73     multi_ptr(accessor<ElementType, Dimensions, Mode, target::constant_buffer,
74               IsPlaceholder>);
75
76     // Returns the underlying OpenCL C pointer
77     pointer_t get() const;
78
79     std::add_pointer_t<value_type> get_raw() const;
80
81     pointer_t get_decorated() const;
82
83     // Implicit conversion to the underlying pointer type
84     operator ElementType*() const;
85
86     // Implicit conversion to a multi_ptr<void>
87     // Available only when ElementType is not const-qualified
88     operator multi_ptr<void, Space, access::decorated::legacy>() const;
89
90     // Implicit conversion to a multi_ptr<const void>
91     // Available only when ElementType is const-qualified
92     operator multi_ptr<const void, Space, access::decorated::legacy>() const;
93
94     // Implicit conversion to multi_ptr<const ElementType, Space>
95     operator multi_ptr<const ElementType, Space, access::decorated::legacy>()
96         const;
97
98     // Arithmetic operators
99     friend multi_ptr& operator++(multi_ptr& mp) { /* ... */
100 }
101     friend multi_ptr operator++(multi_ptr& mp, int) { /* ... */
102 }
103     friend multi_ptr& operator--(multi_ptr& mp) { /* ... */
104 }
105     friend multi_ptr operator--(multi_ptr& mp, int) { /* ... */
106 }
107     friend multi_ptr& operator+=(multi_ptr& lhs, difference_type r) { /* ... */
108 }
109     friend multi_ptr& operator-=(multi_ptr& lhs, difference_type r) { /* ... */
110 }
111     friend multi_ptr operator+(const multi_ptr& lhs,
112                               difference_type r) { /* ... */
113 }
114     friend multi_ptr operator-(const multi_ptr& lhs,
115                               difference_type r) { /* ... */
116 }

```



```

117
118 void prefetch(size_t numElements) const;
119
120 friend bool operator==(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
121 }
122 friend bool operator!=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
123 }
124 friend bool operator<(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
125 }
126 friend bool operator>(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
127 }
128 friend bool operator<=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
129 }
130 friend bool operator>=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
131 }
132
133 friend bool operator==(const multi_ptr& lhs, std::nullptr_t) { /* ... */
134 }
135 friend bool operator!=(const multi_ptr& lhs, std::nullptr_t) { /* ... */
136 }
137 friend bool operator<(const multi_ptr& lhs, std::nullptr_t) { /* ... */
138 }
139 friend bool operator>(const multi_ptr& lhs, std::nullptr_t) { /* ... */
140 }
141 friend bool operator<=(const multi_ptr& lhs, std::nullptr_t) { /* ... */
142 }
143 friend bool operator>=(const multi_ptr& lhs, std::nullptr_t) { /* ... */
144 }
145
146 friend bool operator==(std::nullptr_t, const multi_ptr& rhs) { /* ... */
147 }
148 friend bool operator!=(std::nullptr_t, const multi_ptr& rhs) { /* ... */
149 }
150 friend bool operator<(std::nullptr_t, const multi_ptr& rhs) { /* ... */
151 }
152 friend bool operator>(std::nullptr_t, const multi_ptr& rhs) { /* ... */
153 }
154 friend bool operator<=(std::nullptr_t, const multi_ptr& rhs) { /* ... */
155 }
156 friend bool operator>=(std::nullptr_t, const multi_ptr& rhs) { /* ... */
157 }
158 };
159
160 // Legacy interface, inherited from 1.2.1.
161 // Specialization of multi_ptr for void and const void
162 // VoidType can be either void or const void
163 template <access::address_space Space>
164 class [[deprecated]] multi_ptr<VoidType, Space, access::decorated::legacy> {
165 public:
166     using value_type = VoidType;
167     using element_type = VoidType;
168     using difference_type = std::ptrdiff_t;
169
170     // Implementation defined pointer types that correspond to
171     // SYCL/OpenCL interoperability types for OpenCL C functions
172     using pointer_t = multi_ptr<VoidType, Space, access::decorated::yes>::pointer;

```

```

173 using const_pointer_t =
174     multi_ptr<const VoidType, Space, access::decorated::yes>::pointer;
175
176 static constexpr access::address_space address_space = Space;
177
178 // Constructors
179 multi_ptr();
180 multi_ptr(const multi_ptr&);
181 multi_ptr(multi_ptr&&);
182 multi_ptr(pointer_t);
183 multi_ptr(VoidType*);
184 multi_ptr(std::nullptr_t);
185 ~multi_ptr();
186
187 // Assignment operators
188 multi_ptr& operator=(const multi_ptr&);
189 multi_ptr& operator=(multi_ptr&&);
190 multi_ptr& operator=(pointer_t);
191 multi_ptr& operator=(VoidType*);
192 multi_ptr& operator=(std::nullptr_t);
193
194 // Available only when:
195 // (Space == access::address_space::global_space ||
196 //  Space == access::address_space::generic_space) &&
197 // (std::is_const_v<VoidType> ||
198 //  !std::is_const_v<accessor<ElementType, Dimensions, Mode, target::device,
199 //  IsPlaceholder>::value_type>)
200 template <typename ElementType, int Dimensions, access_mode Mode>
201 multi_ptr(accessor<ElementType, Dimensions, Mode, target::device>);
202
203 // Available only when:
204 // (Space == access::address_space::local_space ||
205 //  Space == access::address_space::generic_space) &&
206 // (std::is_const_v<VoidType> || !std::is_const_v<ElementType>)
207 template <typename ElementType, int Dimensions, access_mode Mode>
208 multi_ptr(accessor<ElementType, Dimensions, Mode, target::local>);
209
210 // Available only when:
211 // (Space == access::address_space::local_space ||
212 //  Space == access::address_space::generic_space) &&
213 // (std::is_const_v<VoidType> || !std::is_const_v<ElementType>)
214 template <typename AccDataT, int Dimensions>
215 multi_ptr(local_accessor<AccDataT, Dimensions>);
216
217 // Only if Space == access::address_space::constant_space
218 template <typename ElementType, int Dimensions, access_mode Mode>
219 multi_ptr(accessor<ElementType, Dimensions, Mode, target::constant_buffer>);
220
221 // Returns the underlying OpenCL C pointer
222 pointer_t get() const;
223
224 std::add_pointer_t<value_type> get_raw() const;
225
226 pointer_t get_decorated() const;
227
228 // Implicit conversion to the underlying pointer type

```

```

229 operator VoidType*() const;
230
231 // Explicit conversion to a multi_ptr<ElementType>
232 // If VoidType is const, ElementType must be as well
233 template <typename ElementType>
234 explicit
235 operator multi_ptr<ElementType, Space, access::decorated::legacy>() const;
236
237 // Implicit conversion to multi_ptr<const void, Space>
238 operator multi_ptr<const void, Space, access::decorated::legacy>() const;
239
240 friend bool operator==(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
241 }
242 friend bool operator!=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
243 }
244 friend bool operator<(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
245 }
246 friend bool operator>(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
247 }
248 friend bool operator<=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
249 }
250 friend bool operator>=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
251 }
252
253 friend bool operator==(const multi_ptr& lhs, std::nullptr_t) { /* ... */
254 }
255 friend bool operator!=(const multi_ptr& lhs, std::nullptr_t) { /* ... */
256 }
257 friend bool operator<(const multi_ptr& lhs, std::nullptr_t) { /* ... */
258 }
259 friend bool operator>(const multi_ptr& lhs, std::nullptr_t) { /* ... */
260 }
261 friend bool operator<=(const multi_ptr& lhs, std::nullptr_t) { /* ... */
262 }
263 friend bool operator>=(const multi_ptr& lhs, std::nullptr_t) { /* ... */
264 }
265
266 friend bool operator==(std::nullptr_t, const multi_ptr& rhs) { /* ... */
267 }
268 friend bool operator!=(std::nullptr_t, const multi_ptr& rhs) { /* ... */
269 }
270 friend bool operator<(std::nullptr_t, const multi_ptr& rhs) { /* ... */
271 }
272 friend bool operator>(std::nullptr_t, const multi_ptr& rhs) { /* ... */
273 }
274 friend bool operator<=(std::nullptr_t, const multi_ptr& rhs) { /* ... */
275 }
276 friend bool operator>=(std::nullptr_t, const multi_ptr& rhs) { /* ... */
277 }
278 };
279
280 } // namespace sycl

```

### 4.7.7.2. Explicit pointer aliases

SYCL provides aliases to the `multi_ptr` class template (see [Section 4.7.7.1](#)) for each specialization of `access::address_space`.

A synopsis of the SYCL `multi_ptr` class template aliases is provided below.

```

1 namespace sycl {
2
3 template <typename ElementType, access::address_space Space,
4         access::decorated IsDecorated>
5 class multi_ptr;
6
7 // Template specialization aliases for different pointer address spaces
8
9 template <typename ElementType,
10         access::decorated IsDecorated = access::decorated::legacy>
11 using global_ptr =
12     multi_ptr<ElementType, access::address_space::global_space, IsDecorated>;
13
14 template <typename ElementType,
15         access::decorated IsDecorated = access::decorated::legacy>
16 using local_ptr =
17     multi_ptr<ElementType, access::address_space::local_space, IsDecorated>;
18
19 // Deprecated in SYCL 2020
20 template <typename ElementType>
21 using constant_ptr =
22     multi_ptr<ElementType, access::address_space::constant_space,
23             access::decorated::legacy>;
24
25 template <typename ElementType,
26         access::decorated IsDecorated = access::decorated::legacy>
27 using private_ptr =
28     multi_ptr<ElementType, access::address_space::private_space, IsDecorated>;
29
30 template <typename ElementType,
31         access::decorated IsDecorated = access::decorated::legacy>
32 using generic_ptr =
33     multi_ptr<ElementType, access::address_space::generic_space, IsDecorated>;
34
35 // Template specialization aliases for different pointer address spaces.
36 // The interface exposes non-decorated pointer while keeping the
37 // address space information internally.
38
39 template <typename ElementType>
40 using raw_global_ptr =
41     multi_ptr<ElementType, access::address_space::global_space,
42             access::decorated::no>;
43
44 template <typename ElementType>
45 using raw_local_ptr = multi_ptr<ElementType, access::address_space::local_space,
46                               access::decorated::no>;
47
48 template <typename ElementType>
49 using raw_private_ptr =

```

```

50     multi_ptr<ElementType, access::address_space::private_space,
51         access::decorated::no>;
52
53 template <typename ElementType>
54 using raw_generic_ptr =
55     multi_ptr<ElementType, access::address_space::generic_space,
56         access::decorated::no>;
57
58 // Template specialization aliases for different pointer address spaces.
59 // The interface exposes decorated pointer.
60
61 template <typename ElementType>
62 using decorated_global_ptr =
63     multi_ptr<ElementType, access::address_space::global_space,
64         access::decorated::yes>;
65
66 template <typename ElementType>
67 using decorated_local_ptr =
68     multi_ptr<ElementType, access::address_space::local_space,
69         access::decorated::yes>;
70
71 template <typename ElementType>
72 using decorated_private_ptr =
73     multi_ptr<ElementType, access::address_space::private_space,
74         access::decorated::yes>;
75
76 template <typename ElementType>
77 using decorated_generic_ptr =
78     multi_ptr<ElementType, access::address_space::generic_space,
79         access::decorated::yes>;
80
81 } // namespace sycl

```

### 4.7.8. Image samplers

The SYCL `image_sampler` struct contains a configuration for sampling a `sampled_image`. The members of this struct are defined by the following tables.

```

1 namespace sycl {
2
3 enum class addressing_mode : /* unspecified */ {
4     mirrored_repeat,
5     repeat,
6     clamp_to_edge,
7     clamp,
8     none
9 };
10
11 enum class filtering_mode : /* unspecified */ { nearest, linear };
12
13 enum class coordinate_normalization_mode : /* unspecified */ {
14     normalized,
15     unnormalized
16 };
17

```

```

18 struct image_sampler {
19     addressing_mode addressing;
20     coordinate_normalization_mode coordinate;
21     filtering_mode filtering;
22 };
23
24 } // namespace sycl

```

Table 75. Addressing modes description

addressing_mode	Description
mirrored_repeat	Out of range coordinates will be flipped at every integer junction. This addressing mode can only be used with normalized coordinates. If normalized coordinates are not used, this addressing mode may generate image coordinates that are undefined.
repeat	Out of range image coordinates are wrapped to the valid range. This addressing mode can only be used with normalized coordinates. If normalized coordinates are not used, this addressing mode may generate image coordinates that are undefined.
clamp_to_edge	Out of range image coordinates are clamped to the extent.
clamp	Out of range image coordinates will return a border color.
none	For this addressing mode the programmer guarantees that the image coordinates used to sample elements of the image refer to a location inside the image; otherwise the results are undefined.

Table 76. Filtering modes description

filtering_mode	Description
nearest	Chooses a color of nearest pixel.
linear	Performs a linear sampling of adjacent pixels.

Table 77. Coordinate normalization modes description

coordinate_normalization_mode	Description
normalized	Normalizes image coordinates.

coordinate_normalization_mode	Description
unnormalized	Does not normalize image coordinates.

## 4.8. Unified shared memory (USM)

This section describes properties and routines for pointer-based memory management interfaces in SYCL. These routines augment, rather than replace, the buffer-based interfaces in SYCL.

Unified Shared Memory (USM) provides a pointer-based alternative to the buffer programming model. USM enables:

- Easier integration into existing code bases by representing allocations as pointers rather than buffers, with full support for pointer arithmetic into allocations.
- Fine-grain control over ownership and accessibility of allocations, to optimally choose between performance and programmer convenience.
- A simpler programming model, by automatically migrating some allocations between SYCL devices and the host.

To show the differences with the example from [Section 3.2](#), the following source code example shows how shared memory can be used between host and device:

```

1 #include <iostream>
2 #include <sycl/sycl.hpp>
3 using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names
4
5 int main() {
6     // Create a default queue to enqueue work to the default device
7     queue myQueue;
8
9     // Allocate shared memory bound to the device and context associated to the
10    // queue Replacing malloc_shared with malloc_host would yield a correct
11    // program that allocated device-visible memory on the host.
12    int* data = sycl::malloc_shared<int>(1024, myQueue);
13
14    myQueue.parallel_for(1024, [=](id<1> idx) {
15        // Initialize each buffer element with its own rank number starting at 0
16        data[idx] = idx;
17    }); // End of the kernel function
18
19    // Explicitly wait for kernel execution since there is no accessor involved
20    myQueue.wait();
21
22    // Print result
23    for (int i = 0; i < 1024; i++)
24        std::cout << "data[" << i << "] = " << data[i] << std::endl;
25
26    return 0;
27 }
```

By comparison, the following source code example uses less capable device memory, which requires an explicit copy between the device and the host:

```

1 #include <iostream>
2 #include <sycl/sycl.hpp>
3 using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names
4
5 int main() {
6     // Create a default queue to enqueue work to the default device
7     queue myQueue;
8
9     // Allocate device USM, using the device and context associated with the queue
10    int* data = sycl::malloc_device<int>(1024, myQueue);
11
12    myQueue.parallel_for(1024, [=](id<1> idx) {
13        // Initialize each buffer element with its own rank number starting at 0
14        data[idx] = idx;
15    }); // End of the kernel function
16
17    // Explicitly wait for kernel execution since there is no accessor involved
18    myQueue.wait();
19
20    // Create an array to receive the device content
21    int hostData[1024];
22    // Receive the content from the device
23    myQueue.memcpy(hostData, data, 1024 * sizeof(int));
24    // Wait for the copy to complete
25    myQueue.wait();
26
27    // Print result
28    for (int i = 0; i < 1024; i++)
29        std::cout << "hostData[" << i << "] = " << hostData[i] << std::endl;
30
31    return 0;
32 }

```

## 4.8.1. Unified addressing

Unified Addressing guarantees that all devices will use a unified address space. Pointer values in the unified address space will always refer to the same location in memory. The unified address space encompasses the host and one or more devices. Note that this does not require addresses in the unified address space to be accessible on all devices, just that pointer values will be consistent.

## 4.8.2. Kinds of unified shared memory

USM is a capability that, when available, provides the ability to create allocations that are visible to both host and device(s). USM builds upon Unified Addressing to define a shared address space where pointer values in this space always refer to the same location in memory. USM defines three types of memory allocations described in [Table 78](#).

Table 78. Type of USM allocations

USM allocation type	Description
host	Allocations in host memory that are accessible by a device
device	Allocations in device memory that are <b>not</b> accessible by the host
shared	Allocations in shared memory that are accessible by both host and device



The following `enum` is used to refer to the different types of allocations inside of a SYCL program:

```

1 namespace sycl {
2 namespace usm {
3
4 enum class alloc : /* unspecified */ {
5     host,
6     device,
7     shared,
8     unknown
9 };
10
11 }
12 }
```

USM is an optional feature which may not be supported by all devices, and devices that support USM may not support all types of USM allocation. A SYCL application can use the `device::has()` function to determine the level of USM support for a device. See [Section 4.6.4.5](#) for more details.

The characteristics of USM allocations are summarized in [Table 79](#).

Table 79. Characteristics of the different kinds of USM allocation

Allocation Type	Initial Location	Accessible By		Migratable To	
device	device	host	No	host	No
		device	Yes	device	N/A
		Another device	Optional (P2P)	Another device	No
host	host	host	Yes	host	N/A
		Any device	Yes	device	No
shared	Unspecified	host	Yes	host	Yes
		device	Yes	device	Yes
		Another device	Optional	Another device	Optional

Each USM allocation has an associated SYCL [context](#), and any access to that memory must use the same context. Specifically, any [SYCL kernel function](#) that dereferences a pointer to a USM allocation must be submitted to a [queue](#) that was constructed with the same context that was used to allocate that memory. The explicit memory operation [commands](#) that take USM pointers have a similar restriction. (See [Section 4.9.4.3](#) for details.) Violations of these requirements result in undefined behavior.



There are no similar restrictions for dereferencing a USM pointer in a [host task](#). This is legal regardless of which [queue](#) the host task was submitted to so long as the USM pointer is accessible on the host.

Each type of USM allocation has different rules for where that memory is accessible. Attempting to dereference a USM pointer on the host or on a device in violation of these rules results in undefined behavior. Passing a USM pointer to one of the explicit memory functions where the pointer is not accessible to the device generally results in undefined behavior. See [Section 4.9.4.3](#) for the exact rules.

Device allocations are used for explicitly managing device memory. Programmers directly allocate device memory and explicitly copy data between host memory and a device allocation. Device allocations are obtained through SYCL device USM allocation routines instead of system allocation routines

like `std::malloc` or C++ `new`. Device allocations are not accessible on the host, but the pointer values remain consistent on account of Unified Addressing. The size of device allocations will be limited by the amount of memory in a device. Support for device allocations on a specific device can be queried through `aspect::usm_device_allocations`.

Device allocations must be explicitly copied between the host and a device. The member functions to copy and initialize data are found in [Section 4.6.5.3](#) and [Table 113](#), and these functions may be used on device allocations if a device supports `aspect::usm_device_allocations`.

Host allocations allow devices to directly read and write host memory inside of a kernel. This can be useful for several reasons, such as when the overhead of moving a small amount of data is not worth paying over the cost of a remote access or when the size of a data set exceeds the size of a device's memory. Host allocations must also be obtained using SYCL routines instead of system allocation routines. While a device may remotely read and write a host allocation, the allocation does not migrate to the device - it remains in host memory. Users should take care to properly synchronize access to host allocations between host execution and kernels. The total size of host allocations will be limited by the amount of pinnable-memory on the host on most systems. Support for host allocations on a specific device can be queried through `aspect::usm_host_allocations`. Support for atomic modification of host allocations on a specific device can be queried through `aspect::usm_atomic_host_allocations`.

Shared allocations implicitly share data between the host and devices. Data may move to where it is being used without the programmer explicitly informing the runtime. It is up to the runtime and backends to make sure that a shared allocation is available where it is used. Shared allocations must also be obtained using SYCL allocation routines instead of the system allocator. The maximum size of a shared allocation on a specific device, and the total size of all shared allocations in a context, are implementation-defined. Support for shared allocations on a specific device can be queried through `aspect::usm_shared_allocations`.

Not all devices may support concurrent access of a shared allocation with the host. If a device does not support this, host execution and device code must take turns accessing the allocation, so the host must not access a shared allocation while a kernel is executing. Host access to a shared allocation which is also accessed by an executing kernel on a device that does not support concurrent access results in undefined behavior. If a device does support concurrent access, both the host and the device may atomically modify the same data inside an allocation. Allocations, or pieces of allocations, are now free to migrate to different devices in the same context that also support this capability. Additionally, many devices that support concurrent access may support a working set of shared allocations larger than device memory. Users may query whether a device supports concurrent access with atomic modification of shared allocations through the aspect `aspect::usm_atomic_shared_allocations`. See [Section 4.6.4.5](#) for more details.

Performance hints for shared allocations may be specified by the user by enqueueing `prefetch` operations on a device. These operations inform the SYCL runtime that the specified shared allocation is likely to be accessed on the device in the future, and that it is free to migrate the allocation to the device. More about `prefetch` is found in [Section 4.6.5.3](#) and [Table 113](#). If a device supports concurrent access to shared allocations, then `prefetch` operations may be overlapped with kernel execution.

Additionally, users may use the `mem_advise` member function to annotate shared allocations with `advice`. Valid `advice` is defined by the device and its associated backend. See [Section 4.6.5.3](#) and [Table 113](#) for more information.

In the most capable systems, users do not need to use SYCL USM allocation functions to create shared allocations. The system allocator (`malloc/new`) may instead be used. Likewise, `std::free` and `delete` are used instead of `sycl::free`. Note that host and device allocations are unaffected by this change and must still be allocated using their respective USM functions in order to guarantee their behavior. Users may query the device to determine if system allocations are supported for use on the device, through `aspect::usm_system_allocations`.

### 4.8.3. USM allocations

USM provides several allocation functions. These functions accept a `property_list` parameter, which is provided for future extensibility. The [core SYCL specification](#) does not yet define any USM allocation properties.

Some of the allocation functions take an explicit alignment parameter. Like `std::aligned_alloc`, these functions return `nullptr` if the alignment is not supported by the implementation. Some of the allocation functions are templated on the allocated type `T` and some are not. The following table specifies the alignment guarantees for each category.

Table 80. Alignment guarantees of USM allocation functions

Category	Alignment guarantee
No alignment parameter Not templated on allocation type	Pointer is suitably aligned for any object with fundamental alignment whose size is less than or equal to the requested allocation size.
No alignment parameter Templated on allocation type <code>T</code>	Pointer is suitably aligned for an object of type <code>T</code> .
Alignment parameter <code>alignment</code> specified Not templated on allocation type	Pointer is suitably aligned for any object with fundamental alignment whose size is less than or equal to the requested allocation size or it is aligned to the specified <code>alignment</code> , whichever is greater.
Alignment parameter <code>alignment</code> specified Templated on allocation type <code>T</code>	Pointer is suitably aligned for an object of type <code>T</code> or it is aligned to the specified <code>alignment</code> , whichever is greater.

#### 4.8.3.1. C++ allocator interface

SYCL defines an allocator class named `usm_allocator` that satisfies the C++ named requirement `Allocator`. The `AllocKind` template parameter can be either `usm::alloc::host` or `usm::alloc::shared`, causing the allocator to make either host USM allocations or shared USM allocations.



There is no specialization for `usm::alloc::device` because an `Allocator` is required to allocate memory that is accessible on the host.

The `usm_allocator` class has a template argument `Alignment`, which specifies the minimum alignment for memory that it allocates. This alignment is used even if the allocator is rebound to a different type. Memory allocated by this allocator is suitably aligned for objects of its underlying `value_type` or at the alignment specified by `Alignment`, whichever is greater.

A synopsis of the `usm_allocator` class is provided below. The constructors are listed in [Table 81](#).

```

1 template <typename T, usm::alloc AllocKind, size_t Alignment = 0>
2 class usm_allocator {
3 public:
4     using value_type = T;
5     using propagate_on_container_copy_assignment = std::true_type;
6     using propagate_on_container_move_assignment = std::true_type;
7     using propagate_on_container_swap = std::true_type;
8
9 public:
10     template <typename U> struct rebound {
11         typedef usm_allocator<U, AllocKind, Alignment> other;
12     };
13 
```

```

14  usm_allocator() = delete;
15  usm_allocator(const context& syclContext,
16                const device& syclDevice,
17                const property_list& propList = {});
18  usm_allocator(const queue& syclQueue,
19                const property_list& propList = {});
20  usm_allocator(const usm_allocator& other);
21  usm_allocator(usm_allocator&&) noexcept;
22  usm_allocator& operator=(const usm_allocator&);
23  usm_allocator& operator=(usm_allocator&&);
24
25  template <class U>
26  usm_allocator(usm_allocator<U, AllocKind, Alignment> const&) noexcept;
27
28  /// Allocate memory
29  T* allocate(size_t count);
30
31  /// Deallocate memory
32  void deallocate(T* Ptr, size_t count);
33
34  /// Equality Comparison
35  ///
36  /// Allocators only compare equal if they are of the same USM kind, alignment,
37  /// context, and device
38  template <class U, usm::alloc AllocKindU, size_t AlignmentU>
39  friend bool operator==(const usm_allocator<T, AllocKind, Alignment>&,
40                        const usm_allocator<U, AllocKindU, AlignmentU>&);
41
42  /// Inequality Comparison
43  /// Allocators only compare unequal if they are not of the same USM kind, alignment,
44  /// context, or device
45  template <class U, usm::alloc AllocKindU, size_t AlignmentU>
46  friend bool operator!=(const usm_allocator<T, AllocKind, Alignment>&,
47                        const usm_allocator<U, AllocKindU, AlignmentU>&);
48  };

```

Table 81. Constructors of the `usm_allocator` class

Constructor	Description
<pre>usm_allocator(const context&amp; syclContext, const device&amp; syclDevice,                const property_list&amp; propList = {})</pre>	<p>Constructs a <code>usm_allocator</code> instance that allocates USM for the provided context and device.</p> <p>If <code>AllocKind</code> is <code>usm::alloc::host</code>, this constructor throws a synchronous <code>exception</code> with the <code>errc::feature_not_supported</code> error code if no device in <code>syclContext</code> has <code>aspect::usm_host_allocations</code>. The <code>syclDevice</code> is ignored for this allocation kind.</p> <p>If <code>AllocKind</code> is <code>usm::alloc::shared</code>, this constructor throws a synchronous <code>exception</code> with the <code>errc::feature_not_supported</code> error code if the <code>syclDevice</code> does not have <code>aspect::usm_shared_allocations</code>. The <code>syclDevice</code> must either be contained by <code>syclContext</code> or it must be a <code>descendent device</code> of some device that is contained by that context, otherwise this constructor throws a synchronous <code>exception</code> with the <code>errc::invalid</code> error code.</p>
<pre>usm_allocator(const queue&amp; syclQueue, const property_list&amp; propList = {})</pre>	<p>Simplified constructor form where <code>syclQueue</code> provides the <code>device</code> and <code>context</code>.</p>

#### 4.8.3.2. Device allocation functions

The functions in [Table 82](#) allocate device USM. On success, these functions return a pointer to the newly allocated memory, which must eventually be deallocated with `sycl::free` in order to avoid a memory leak. If there are not enough resources to allocate the requested memory, these functions return `nullptr`.

When the allocation size is zero bytes (`numBytes` or `count` is zero), these functions behave in a manner consistent with C++ `std::malloc`. The value returned is unspecified in this case, and the returned pointer may not be used to access storage. If this pointer is not null, it must be passed to `sycl::free` to avoid a memory leak.

Table 82. Device USM Allocation Functions

Function	Description
<pre>void* <b>sycl::malloc_device</b>(size_t numBytes, const <b>device</b>&amp; <b>syclDevice</b>,                                 const <b>context</b>&amp; <b>syclContext</b>,                                 const <b>property_list</b>&amp; <b>propList</b> =     {})</pre>	<p>Returns a pointer to the newly allocated memory, which is allocated on <b>syclDevice</b>. The allocation size is specified in bytes. Throws a synchronous <b>exception</b> with the <b>errc::feature_not_supported</b> error code if the <b>syclDevice</b> does not have <b>aspect::usm_device_allocations</b>. The <b>syclDevice</b> must either be contained by <b>syclContext</b> or it must be a <b>descendent device</b> of some device that is contained by that context, otherwise this function throws a synchronous <b>exception</b> with the <b>errc::invalid</b> error code.</p>
<pre>template &lt;typename T&gt; T* <b>sycl::malloc_device</b>(size_t count, const <b>device</b>&amp; <b>syclDevice</b>,                                 const <b>context</b>&amp; <b>syclContext</b>,                                 const <b>property_list</b>&amp; <b>propList</b> = {})</pre>	<p>Returns a pointer to the newly allocated memory, which is allocated on <b>syclDevice</b>. The allocation size is specified in number of elements of type <b>T</b>. Throws a synchronous <b>exception</b> with the <b>errc::feature_not_supported</b> error code if the <b>syclDevice</b> does not have <b>aspect::usm_device_allocations</b>. The <b>syclDevice</b> must either be contained by <b>syclContext</b> or it must be a <b>descendent device</b> of some device that is contained by that context, otherwise this function throws a synchronous <b>exception</b> with the <b>errc::invalid</b> error code.</p>
<pre>void* <b>sycl::malloc_device</b>(size_t numBytes, const <b>queue</b>&amp; <b>syclQueue</b>,                                 const <b>property_list</b>&amp; <b>propList</b> =     {})</pre>	<p>Simplified form where <b>syclQueue</b> provides the <b>device</b> and <b>context</b>.</p>
<pre>template &lt;typename T&gt; T* <b>sycl::malloc_device</b>(size_t count, const <b>queue</b>&amp; <b>syclQueue</b>,                                 const <b>property_list</b>&amp; <b>propList</b> = {})</pre>	<p>Simplified form where <b>syclQueue</b> provides the <b>device</b> and <b>context</b>.</p>

Function	Description
<pre> void* <b>sycl::aligned_alloc_device</b>(size_t alignment, size_t numBytes,                                 const <b>device&amp;</b> syclDevice,                                 const <b>context&amp;</b> syclContext,                                 const <b>property_list&amp;</b> propList = {}) </pre>	<p>Returns a pointer to the newly allocated memory, which is allocated on <b>syclDevice</b>. The allocation is specified in bytes and aligned according to <b>alignment</b>. Throws a synchronous <b>exception</b> with the <b>errc::feature_not_supported</b> error code if the <b>syclDevice</b> does not have <b>aspect::usm_device_allocations</b>. The <b>syclDevice</b> must either be contained by <b>syclContext</b> or it must be a <b>descendent device</b> of some device that is contained by that context, otherwise this function throws a synchronous <b>exception</b> with the <b>errc::invalid</b> error code.</p>
<pre> template &lt;typename T&gt; T* <b>sycl::aligned_alloc_device</b>(size_t alignment, size_t count,                                 const <b>device&amp;</b> syclDevice,                                 const <b>context&amp;</b> syclContext,                                 const <b>property_list&amp;</b> propList = {}) </pre>	<p>Returns a pointer to the newly allocated memory, which is allocated on <b>syclDevice</b>. The allocation is specified in number of elements of type <b>T</b> and aligned according to <b>alignment</b>. Throws a synchronous <b>exception</b> with the <b>errc::feature_not_supported</b> error code if the <b>syclDevice</b> does not have <b>aspect::usm_device_allocations</b>. The <b>syclDevice</b> must either be contained by <b>syclContext</b> or it must be a <b>descendent device</b> of some device that is contained by that context, otherwise this function throws a synchronous <b>exception</b> with the <b>errc::invalid</b> error code.</p>
<pre> void* <b>sycl::aligned_alloc_device</b>(size_t alignment, size_t numBytes,                                 const <b>queue&amp;</b> syclQueue,                                 const <b>property_list&amp;</b> propList = {}) </pre>	<p>Simplified form where <b>syclQueue</b> provides the <b>device</b> and <b>context</b>.</p>
<pre> template &lt;typename T&gt; T* <b>sycl::aligned_alloc_device</b>(size_t alignment, size_t count,                                 const <b>queue&amp;</b> syclQueue,                                 const <b>property_list&amp;</b> propList = {}) </pre>	<p>Simplified form where <b>syclQueue</b> provides the <b>device</b> and <b>context</b>.</p>

#### 4.8.3.3. Host allocation functions

The functions in [Table 83](#) allocate host USM. On success, these functions return a pointer to the newly allocated memory, which must eventually be deallocated with **sycl::free** in order to avoid a memory



leak. If there are not enough resources to allocate the requested memory, these functions return `nullptr`.

When the allocation size is zero bytes (`numBytes` or `count` is zero), these functions behave in a manner consistent with C++ `std::malloc`. The value returned is unspecified in this case, and the returned pointer may not be used to access storage. If this pointer is not null, it must be passed to `sycl::free` to avoid a memory leak.

Table 83. Host USM Allocation Functions

Function	Description
<pre>void* sycl::malloc_host(size_t numBytes, const context&amp;                         syclContext,                         const property_list&amp; propList = {})</pre>	Returns a pointer to the newly allocated memory. This allocation is specified in bytes. Throws a synchronous <code>exception</code> with the <code>errc::feature_not_supported</code> error code if no device in <code>syclContext</code> has <code>aspect::usm_host_allocations</code> .
<pre>template &lt;typename T&gt; T* sycl::malloc_host(size_t count, const context&amp;                     syclContext,                     const property_list&amp; propList = {})</pre>	Returns a pointer to the newly allocated memory. This allocation is specified in number of elements of type <code>T</code> . Throws a synchronous <code>exception</code> with the <code>errc::feature_not_supported</code> error code if no device in <code>syclContext</code> has <code>aspect::usm_host_allocations</code> .
<pre>void* sycl::malloc_host(size_t numBytes, const queue&amp;                         syclQueue,                         const property_list&amp; propList = {})</pre>	Simplified form where <code>syclQueue</code> provides the <code>context</code> .
<pre>template &lt;typename T&gt; T* sycl::malloc_host(size_t count, const queue&amp; syclQueue,                     const property_list&amp; propList = {})</pre>	Simplified form where <code>syclQueue</code> provides the <code>context</code> .
<pre>void* sycl::aligned_alloc_host(size_t alignment, size_t                               numBytes,                               const context&amp; syclContext,                               const property_list&amp;                               propList = {})</pre>	Returns a pointer to the newly allocated memory. This allocation is specified in bytes and aligned according to <code>alignment</code> . Throws a synchronous <code>exception</code> with the <code>errc::feature_not_supported</code> error code if no device in <code>syclContext</code> has <code>aspect::usm_host_allocations</code> .
<pre>template &lt;typename T&gt; T* sycl::aligned_alloc_host(size_t alignment, size_t count,                             const context&amp; syclContext,                             const property_list&amp; propList =                             {})</pre>	Returns a pointer to the newly allocated memory. This allocation is specified in elements of type <code>T</code> and aligned according to <code>alignment</code> . Throws a synchronous <code>exception</code> with the <code>errc::feature_not_supported</code> error code if no device in <code>syclContext</code> has <code>aspect::usm_host_allocations</code> .



Function	Description
<pre>void* <b>sycl::aligned_alloc_host</b>(size_t alignment, size_t numBytes,                                 const <b>queue&amp;</b> syclQueue,                                 const <b>property_list&amp;</b> propList = {})</pre>	Simplified form where <b>syclQueue</b> provides the <b>context</b> .
<pre>template &lt;typename T&gt; T* <b>sycl::aligned_alloc_host</b>(size_t alignment, size_t count,                                 const <b>queue&amp;</b> syclQueue,                                 const <b>property_list&amp;</b> propList = {})</pre>	Simplified form where <b>syclQueue</b> provides the <b>context</b> .

#### 4.8.3.4. Shared allocation functions

The functions in [Table 84](#) allocate shared USM. On success, these functions return a pointer to the newly allocated memory, which must eventually be deallocated with **sycl::free** in order to avoid a memory leak. If there are not enough resources to allocate the requested memory, these functions return **nullptr**.

When the allocation size is zero bytes (**numBytes** or **count** is zero), these functions behave in a manner consistent with C++ **std::malloc**. The value returned is unspecified in this case, and the returned pointer may not be used to access storage. If this pointer is not null, it must be passed to **sycl::free** to avoid a memory leak.

Table 84. Shared USM Allocation Functions

Function	Description
<pre>void* <b>sycl::malloc_shared</b>(size_t numBytes, const <b>device&amp;</b> syclDevice,                                 const <b>context&amp;</b> syclContext,                                 const <b>property_list&amp;</b> propList = {})</pre>	Returns a pointer to the newly allocated memory, which is associated with <b>syclDevice</b> . This allocation is specified in bytes. Throws a synchronous <b>exception</b> with the <b>errc::feature_not_supported</b> error code if the <b>syclDevice</b> does not have <b>aspect::usm_shared_allocations</b> . The <b>syclDevice</b> must either be contained by <b>syclContext</b> or it must be a <b>descendent device</b> of some device that is contained by that context, otherwise this function throws a synchronous <b>exception</b> with the <b>errc::invalid</b> error code.

Function	Description
<pre>template &lt;typename T&gt; T* <b>sycl::malloc_shared</b>(size_t count, const <b>device</b>&amp; <b>syclDevice</b>,                         const <b>context</b>&amp; <b>syclContext</b>,                         const <b>property_list</b>&amp; <b>propList</b> = {})</pre>	<p>Returns a pointer to the newly allocated memory, which is associated with <b>syclDevice</b>. This allocation is specified in number of elements of type <b>T</b>. Throws a synchronous <b>exception</b> with the <b>errc::feature_not_supported</b> error code if the <b>syclDevice</b> does not have <b>aspect::usm_shared_allocations</b>. The <b>syclDevice</b> must either be contained by <b>syclContext</b> or it must be a <b>descendent device</b> of some device that is contained by that context, otherwise this function throws a synchronous <b>exception</b> with the <b>errc::invalid</b> error code.</p>
<pre>void* <b>sycl::malloc_shared</b>(size_t numBytes, const <b>queue</b>&amp; <b>syclQueue</b>,                         const <b>property_list</b>&amp; <b>propList</b> =                         {})</pre>	<p>Simplified form where <b>syclQueue</b> provides the <b>device</b> and <b>context</b>.</p>
<pre>template &lt;typename T&gt; T* <b>sycl::malloc_shared</b>(size_t count, const <b>queue</b>&amp; <b>syclQueue</b>,                         const <b>property_list</b>&amp; <b>propList</b> = {})</pre>	<p>Simplified form where <b>syclQueue</b> provides the <b>device</b> and <b>context</b>.</p>
<pre>void* <b>sycl::aligned_alloc_shared</b>(size_t alignment, size_t numBytes,                                 const <b>device</b>&amp; <b>syclDevice</b>,                                 const <b>context</b>&amp; <b>syclContext</b>,                                 const <b>property_list</b>&amp; <b>propList</b> = {})</pre>	<p>Returns a pointer to the newly allocated memory, which is associated with <b>syclDevice</b>. This allocation is specified in bytes and aligned according to <b>alignment</b>. Throws a synchronous <b>exception</b> with the <b>errc::feature_not_supported</b> error code if the <b>syclDevice</b> does not have <b>aspect::usm_shared_allocations</b>. The <b>syclDevice</b> must either be contained by <b>syclContext</b> or it must be a <b>descendent device</b> of some device that is contained by that context, otherwise this function throws a synchronous <b>exception</b> with the <b>errc::invalid</b> error code.</p>

Function	Description
<pre> template &lt;typename T&gt; T* <b>sycl::aligned_alloc_shared</b>(size_t alignment, size_t count,                                 const <b>device</b>&amp; syclDevice,                                 const <b>context</b>&amp; syclContext,                                 const <b>property_list</b>&amp; propList = {}) </pre>	<p>Returns a pointer to the newly allocated memory, which is associated with <b>syclDevice</b>. This allocation is specified in number of elements of type <b>T</b> and aligned aligned according to <b>alignment</b>. Throws a synchronous <b>exception</b> with the <b>errc::feature_not_supported</b> error code if the <b>syclDevice</b> does not have <b>aspect::usm_shared_allocations</b>. The <b>syclDevice</b> must either be contained by <b>syclContext</b> or it must be a <b>descendent device</b> of some device that is contained by that context, otherwise this function throws a synchronous <b>exception</b> with the <b>errc::invalid</b> error code.</p>
<pre> void* <b>sycl::aligned_alloc_shared</b>(size_t alignment, size_t numBytes,                                 const <b>queue</b>&amp; syclQueue,                                 const <b>property_list</b>&amp; propList = {}) </pre>	<p>Simplified form where <b>syclQueue</b> provides the <b>device</b> and <b>context</b>.</p>
<pre> template &lt;typename T&gt; T* <b>sycl::aligned_alloc_shared</b>(size_t alignment, size_t count,                                 const <b>queue</b>&amp; syclQueue,                                 const <b>property_list</b>&amp; propList = {}) </pre>	<p>Simplified form where <b>syclQueue</b> provides the <b>device</b> and <b>context</b>.</p>

#### 4.8.3.5. Parameterized allocation functions

The functions in Table 85 take a **kind** parameter that specifies the type of USM to allocate. When **kind** is **usm::alloc::device**, then the allocation device must have **aspect::usm\_device\_allocations**. When **kind** is **usm::alloc::host**, at least one device in the allocation context must have **aspect::usm\_host\_allocations**. When **kind** is **usm::alloc::shared**, the allocation device must have **aspect::usm\_shared\_allocations**. If these requirements are violated, the allocation function throws a synchronous **exception** with the **errc::feature\_not\_supported** error code.

On success, these functions return a pointer to the newly allocated memory, which must eventually be deallocated with **sycl::free** in order to avoid a memory leak. If there are not enough resources to allocate the requested memory, these functions return **nullptr**.

When the allocation size is zero bytes (**numBytes** or **count** is zero), these functions behave in a manner consistent with C++ **std::malloc**. The value returned is unspecified in this case, and the returned pointer may not be used to access storage. If this pointer is not null, it must be passed to **sycl::free** to avoid a memory leak.

Table 85. Parameterized USM Allocation Functions

Function	Description
<pre>void* <b>sycl::malloc</b>(size_t numBytes, const <b>device&amp;</b> syclDevice,                         const <b>context&amp;</b> syclContext, <b>usm::alloc</b> kind,                         const <b>property_list&amp;</b> propList = {})</pre>	<p>Returns a pointer to the newly allocated memory of type <b>kind</b>. This allocation size is specified in bytes. The <b>syclDevice</b> parameter is ignored if <b>kind</b> is <b>usm::alloc::host</b>. If <b>kind</b> is not <b>usm::alloc::host</b>, <b>syclDevice</b> must either be contained by <b>syclContext</b> or it must be a <b>descendent device</b> of some device that is contained by that context, otherwise this function throws a synchronous <b>exception</b> with the <b>errc::invalid</b> error code.</p>
<pre>template &lt;typename T&gt; T* <b>sycl::malloc</b>(size_t count, const <b>device&amp;</b> syclDevice,                 const <b>context&amp;</b> syclContext, <b>usm::alloc</b> kind,                 const <b>property_list&amp;</b> propList = {})</pre>	<p>Returns a pointer to the newly allocated memory of type <b>kind</b>. This allocation size is specified in number of elements of type <b>T</b>. The <b>syclDevice</b> parameter is ignored if <b>kind</b> is <b>usm::alloc::host</b>. If <b>kind</b> is not <b>usm::alloc::host</b>, <b>syclDevice</b> must either be contained by <b>syclContext</b> or it must be a <b>descendent device</b> of some device that is contained by that context, otherwise this function throws a synchronous <b>exception</b> with the <b>errc::invalid</b> error code.</p>
<pre>void* <b>sycl::malloc</b>(size_t numBytes, const <b>queue&amp;</b> syclQueue, <b>usm::alloc</b> kind,                 const <b>property_list&amp;</b> propList = {})</pre>	<p>Simplified form where <b>syclQueue</b> provides the <b>context</b> and any necessary <b>device</b>.</p>
<pre>template &lt;typename T&gt; T* <b>sycl::malloc</b>(size_t count, const <b>queue&amp;</b> syclQueue, <b>usm</b> <b>::alloc</b> kind,                 const <b>property_list&amp;</b> propList = {})</pre>	<p>Simplified form where <b>syclQueue</b> provides the <b>context</b> and any necessary <b>device</b>.</p>
<pre>void* <b>sycl::aligned_alloc</b>(size_t alignment, size_t numBytes,                         const <b>device&amp;</b> syclDevice, const <b>context&amp;</b> syclContext,                         <b>usm::alloc</b> kind, const <b>property_list&amp;</b> propList = {})</pre>	<p>Returns a pointer to the newly allocated memory of type <b>kind</b>. This allocation is specified in bytes and is aligned according to <b>alignment</b>. The <b>syclDevice</b> parameter is ignored if <b>kind</b> is <b>usm::alloc::host</b>. If <b>kind</b> is not <b>usm::alloc::host</b>, <b>syclDevice</b> must either be contained by <b>syclContext</b> or it must be a <b>descendent device</b> of some device that is contained by that context, otherwise this function throws a synchronous <b>exception</b> with the <b>errc::invalid</b> error code.</p>

Function	Description
<pre>template &lt;typename T&gt; T* <b>sycl::aligned_alloc</b>(size_t alignment, size_t count, const <b>device</b>&amp; syclDevice,                         const <b>context</b>&amp; syclContext, usm ::alloc kind,                         const <b>property_list</b>&amp; propList = {})</pre>	Returns a pointer to the newly allocated memory of type <b>kind</b> . This allocation is specified in number of elements of type <b>T</b> and is aligned according to <b>alignment</b> . The <b>syclDevice</b> parameter is ignored if <b>kind</b> is <b>usm::alloc::host</b> . If <b>kind</b> is not <b>usm::alloc::host</b> , <b>syclDevice</b> must either be contained by <b>syclContext</b> or it must be a <b>descendent device</b> of some device that is contained by that context, otherwise this function throws a synchronous <b>exception</b> with the <b>errc::invalid</b> error code.
<pre>void* <b>sycl::aligned_alloc</b>(size_t alignment, size_t numBytes,                         const <b>queue</b>&amp; syclQueue, usm ::alloc kind,                         const <b>property_list</b>&amp; propList = {})</pre>	Simplified form where <b>syclQueue</b> provides the <b>context</b> and any necessary <b>device</b> .
<pre>template &lt;typename T&gt; T* <b>sycl::aligned_alloc</b>(size_t alignment, size_t count, const <b>queue</b>&amp; syclQueue,                         usm::alloc kind, const <b>property_list</b>&amp; propList = {})</pre>	Simplified form where <b>syclQueue</b> provides the <b>context</b> and any necessary <b>device</b> .

#### 4.8.3.6. Memory deallocation functions

##### free

```
void free(void* ptr, const context& ctxt); (1)
void free(void* ptr, const queue& q);      (2)
```

##### Overload (1):

##### Preconditions:

- **ptr** points to memory allocated against **ctxt** using one of the USM allocation routines, or is a null pointer;
- **ptr** has not previously been deallocated; and
- There are no in-progress or enqueued **commands** using the memory pointed to by **ptr**.

**Effects:** Causes the memory pointed to by **ptr** to be deallocated.

[*Note:* Whether **free** is blocking or non-blocking is unspecified. Applications should not rely on **free** for synchronization, nor assume that **free** cannot cause deadlocks.— *end note*]

**Remarks:** If **ptr** is null, this function has no effect.

**Overload (2):**

*Effects:* Equivalent to `return free(ptr, q.get_context());`.

[*Note:* Although this overload accepts a `queue` argument, it does not submit a "free" `command` to the device; the `queue` argument is only used to determine the `context` associated with `ptr`.— *end note*]

**4.8.4. Unified shared memory pointer queries**

Since USM pointers look like raw C++ pointers, users cannot deduce what kind of USM allocation a given pointer may be from examining its type. However, two functions are defined that let users query the type of a USM allocation and, if applicable, the `device` on which it was allocated. These query functions are only supported on the host.

Table 86. USM Pointer Query Functions

Function	Description
<code>usm::alloc get_pointer_type(const void* ptr, const context&amp; syclContext)</code>	Returns the USM allocation type for <code>ptr</code> if <code>ptr</code> falls inside a valid USM allocation for the context <code>syclContext</code> . Returns <code>usm::alloc::unknown</code> if <code>ptr</code> does not point within a valid USM allocation from <code>syclContext</code> .
<code>device get_pointer_device(const void* ptr, const context&amp; syclContext)</code>	Returns the <code>device</code> associated with the USM allocation. If <code>ptr</code> points within a device USM allocation or a shared USM allocation for the context <code>syclContext</code> , returns the same device that was passed when allocating the memory. If <code>ptr</code> points within a host USM allocation for the context <code>syclContext</code> , returns the first device in <code>syclContext</code> . Throws a synchronous <code>exception</code> with the <code>errc::invalid</code> error code if <code>ptr</code> does not point within a valid USM allocation from <code>syclContext</code> .

**4.9. Expressing parallelism through kernels****4.9.1. Ranges and index space identifiers**

The data parallelism of the SYCL kernel execution model requires instantiation of a parallel execution over a range of iteration space coordinates. To achieve this, SYCL exposes types to define the range of execution and to identify a given execution instance's point in the iteration space.

The following types are defined: `range`, `nd_range`, `id`, `item`, `h_item`, `nd_item` and `group`.

When constructing multi-dimensional ids or ranges from integers, the elements are written such that the right-most element varies fastest in a linearization of the multi-dimensional space (see [Section 3.11.1](#)).

Table 87. Summary of types used to identify points in an index space, and ranges over which those points can vary

Type	Description
<code>id</code>	A point within a range
<code>range</code>	Bounds over which an <code>id</code> may vary
<code>item</code>	Pairing of an <code>id</code> (specific point) and the <code>range</code> that it is bounded by
<code>nd_range</code>	Encapsulates both global and local (work-group size) <code>ranges</code> over which work-item <code>ids</code> will vary
<code>nd_item</code>	Encapsulates two <code>items</code> , one for global <code>id</code> and <code>range</code> , and one for local <code>id</code> and <code>range</code>
<code>h_item</code>	Index point queries within hierarchical parallelism ( <code>parallel_for_work_item</code> ). Encapsulates physical global and local <code>ids</code> and <code>ranges</code> , as well as a logical local <code>id</code> and <code>range</code> defined by hierarchical parallelism
<code>group</code>	Work-group queries within hierarchical parallelism ( <code>parallel_for_work_group</code> ), and exposes the <code>parallel_for_work_item</code> construct that identifies code to be executed by each work-item. Encapsulates work-group <code>ids</code> and <code>ranges</code>

#### 4.9.1.1. `range` class

`range<int Dimensions>` is a 1D, 2D or 3D vector that defines the iteration domain of either a single work-group in a parallel dispatch, or the overall Dimensions of the dispatch. It can be constructed from integers.

The SYCL `range` class template provides the common by-value semantics (see [Section 4.5.3](#)).

A synopsis of the SYCL `range` class is provided below. The constructors, member functions and non-member functions of the SYCL `range` class are listed in [Table 88](#), [Table 89](#) and [Table 90](#) respectively. The additional common special member functions and common member functions are listed in [Section 4.5.3](#) in [Table 9](#) and [Table 10](#) respectively.

```

1 namespace sycl {
2 template <int Dimensions = 1> class range {
3 public:
4     static constexpr int dimensions = Dimensions;
5
6     range();
7
8     /* The following constructor is only available in the range class
9      * specialization where: Dimensions==1 */
10    range(size_t dim0);
11    /* The following constructor is only available in the range class

```

```

12  * specialization where: Dimensions==2 */
13  range(size_t dim0, size_t dim1);
14  /* The following constructor is only available in the range class
15  * specialization where: Dimensions==3 */
16  range(size_t dim0, size_t dim1, size_t dim2);
17
18  /* -- common interface members -- */
19
20  size_t get(int dimension) const;
21  size_t& operator[](int dimension);
22  size_t operator[](int dimension) const;
23
24  size_t size() const;
25
26  // OP is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=
27  friend range operatorOP(const range& lhs, const range& rhs) { /* ... */
28  }
29  friend range operatorOP(const range& lhs, const size_t& rhs) { /* ... */
30  }
31
32  // OP is: +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=
33  friend range& operatorOP(range& lhs, const range& rhs) { /* ... */
34  }
35  friend range& operatorOP(range& lhs, const size_t& rhs) { /* ... */
36  }
37
38  // OP is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=
39  friend range operatorOP(const size_t& lhs, const range& rhs) { /* ... */
40  }
41
42  // OP is unary +, -
43  friend range operatorOP(const range& rhs) { /* ... */
44  }
45
46  // OP is prefix ++, --
47  friend range& operatorOP(range& rhs) { /* ... */
48  }
49
50  // OP is postfix ++, --
51  friend range operatorOP(range& lhs, int) { /* ... */
52  }
53 };
54
55 // Deduction guides
56 range(size_t)->range<1>;
57 range(size_t, size_t)->range<2>;
58 range(size_t, size_t, size_t)->range<3>;
59
60 } // namespace sycl

```

Table 88. Constructors of the `range` class template

Constructor	Description
<code>range()</code>	Construct a SYCL <code>range</code> with the value <code>0</code> for each dimension.



Constructor	Description
<code>range(size_t dim0)</code>	Construct a 1D range with value <code>dim0</code> . Only valid when the template parameter <code>Dimensions</code> is equal to 1.
<code>range(size_t dim0, size_t dim1)</code>	Construct a 2D range with values <code>dim0</code> and <code>dim1</code> . Only valid when the template parameter <code>Dimensions</code> is equal to 2.
<code>range(size_t dim0, size_t dim1, size_t dim2)</code>	Construct a 3D range with values <code>dim0</code> , <code>dim1</code> and <code>dim2</code> . Only valid when the template parameter <code>Dimensions</code> is equal to 3.

Table 89. Member functions of the `range` class template

Member function	Description
<code>size_t get(int dimension) const</code>	Return the value of the specified dimension of the <code>range</code> .
<code>size_t&amp; operator[](int dimension)</code>	Return the l-value of the specified dimension of the <code>range</code> .
<code>size_t operator[](int dimension) const</code>	Return the value of the specified dimension of the <code>range</code> .
<code>size_t size() const</code>	Return the size of the range computed as <code>dimension0*...*dimensionN</code> .

Table 90. Hidden friend functions of the SYCL `range` class template

Hidden friend function	Description
<code>range operatorOP(const range&amp; lhs, const range&amp; rhs)</code>	<p>Where <code>OP</code> is: <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%</code>, <code>&lt;&lt;</code>, <code>&gt;&gt;</code>, <code>&amp;</code>, <code> </code>, <code>^</code>, <code>&amp;&amp;</code>, <code>  </code>, <code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code>, <code>&gt;=</code>.</p> <p>Constructs and returns a new instance of the SYCL <code>range</code> class template with the same dimensionality as <code>lhs range</code>, where each element of the new SYCL <code>range</code> instance is the result of an element-wise <code>OP</code> operator between each element of <code>lhs range</code> and each element of the <code>rhs range</code>. If the operator returns a <code>bool</code>, the result is the cast to <code>size_t</code>.</p>

Hidden friend function	Description
<pre data-bbox="161 185 874 219"><code>range&lt;T&gt; operatorOP(const range&lt;T&gt; lhs, const size_t&lt;T&gt; rhs)</code></pre>	<p data-bbox="1002 159 1461 230">Where <code>OP</code> is: <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%</code>, <code>&lt;&lt;</code>, <code>&gt;&gt;</code>, <code>&amp;</code>, <code> </code>, <code>^</code>, <code>&amp;&amp;</code>, <code>  </code>, <code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code>, <code>&gt;=</code>.</p> <p data-bbox="1002 259 1461 651">Constructs and returns a new instance of the SYCL <code>range</code> class template with the same dimensionality as <code>lhs range</code>, where each element of the new SYCL <code>range</code> instance is the result of an element-wise <code>OP</code> operator between each element of this SYCL <code>range</code> and the <code>rhs size_t</code>. If the operator returns a <code>bool</code>, the result is the cast to <code>size_t</code>.</p>
<pre data-bbox="161 701 794 734"><code>range&lt;T&gt; operatorOP(range&lt;T&gt; lhs, const range&lt;T&gt; rhs)</code></pre>	<p data-bbox="1002 674 1461 745">Where <code>OP</code> is: <code>+=</code>, <code>-=</code>, <code>*=</code>, <code>/=</code>, <code>%=</code>, <code>&lt;&lt;=</code>, <code>&gt;&gt;=</code>, <code>&amp;=</code>, <code> =</code>, <code>^=</code>.</p> <p data-bbox="1002 775 1461 1055">Assigns each element of <code>lhs range</code> instance with the result of an element-wise <code>OP</code> operator between each element of <code>lhs range</code> and each element of the <code>rhs range</code> and returns <code>lhs range</code>. If the operator returns a <code>bool</code>, the result is the cast to <code>size_t</code>.</p>
<pre data-bbox="161 1104 805 1137"><code>range&lt;T&gt; operatorOP(range&lt;T&gt; lhs, const size_t&lt;T&gt; rhs)</code></pre>	<p data-bbox="1002 1077 1461 1149">Where <code>OP</code> is: <code>+=</code>, <code>-=</code>, <code>*=</code>, <code>/=</code>, <code>%=</code>, <code>&lt;&lt;=</code>, <code>&gt;&gt;=</code>, <code>&amp;=</code>, <code> =</code>, <code>^=</code>.</p> <p data-bbox="1002 1178 1461 1424">Assigns each element of <code>lhs range</code> instance with the result of an element-wise <code>OP</code> operator between each element of <code>lhs range</code> and the <code>rhs size_t</code> and returns <code>lhs range</code>. If the operator returns a <code>bool</code>, the result is the cast to <code>size_t</code>.</p>
<pre data-bbox="161 1473 874 1507"><code>range operatorOP(const size_t&lt;T&gt; lhs, const range&lt;T&gt; rhs)</code></pre>	<p data-bbox="1002 1447 1461 1518">Where <code>OP</code> is: <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%</code>, <code>&lt;&lt;</code>, <code>&gt;&gt;</code>, <code>&amp;</code>, <code> </code>, <code>^</code>, <code>&amp;&amp;</code>, <code>  </code>, <code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code>, <code>&gt;=</code>.</p> <p data-bbox="1002 1547 1461 1928">Constructs and returns a new instance of the SYCL <code>range</code> class template with the same dimensionality as the <code>rhs SYCL range</code>, where each element of the new SYCL <code>range</code> instance is the result of an element-wise <code>OP</code> operator between the <code>lhs size_t</code> and each element of the <code>rhs SYCL range</code>. If the operator returns a <code>bool</code>, the result is the cast to <code>size_t</code>.</p>

Hidden friend function	Description
<code>range operatorOP(const range&amp; rhs)</code>	Where <b>OP</b> is: unary <b>+</b> , unary <b>-</b> .  Constructs and returns a new instance of the SYCL <code>range</code> class template with the same dimensionality as the <code>rhs</code> SYCL <code>range</code> , where each element of the new SYCL <code>range</code> instance is the result of an element-wise <b>OP</b> operator on the <code>rhs</code> SYCL <code>range</code> .
<code>range&amp; operatorOP(range&amp; rhs)</code>	Where <b>OP</b> is: prefix <b>++</b> , prefix <b>--</b> .  Assigns each element of the <code>rhs</code> <code>range</code> instance with the result of an element-wise <b>OP</b> operator on each element of the <code>rhs</code> <code>range</code> and returns this <code>range</code> .
<code>range operatorOP(range&amp; lhs, int)</code>	Where <b>OP</b> is: postfix <b>++</b> , postfix <b>--</b> .  Make a copy of the <code>lhs</code> <code>range</code> . Assigns each element of the <code>lhs</code> <code>range</code> instance with the result of an element-wise <b>OP</b> operator on each element of the <code>lhs</code> <code>range</code> . Then return the initial copy of the <code>range</code> .

#### 4.9.1.2. `nd_range` class

```

1 namespace sycl {
2 template <int Dimensions = 1> class nd_range {
3 public:
4     static constexpr int dimensions = Dimensions;
5
6     /* -- common interface members -- */
7
8     // The offset is deprecated in SYCL 2020.
9     nd_range(range<Dimensions> globalSize, range<Dimensions> localSize,
10             id<Dimensions> offset = id<Dimensions>());
11
12     range<Dimensions> get_global_range() const;
13     range<Dimensions> get_local_range() const;
14     range<Dimensions> get_group_range() const;
15     id<Dimensions> get_offset() const; // Deprecated in SYCL 2020.
16 };
17 } // namespace sycl

```

`nd_range<int Dimensions>` defines the iteration domain of both the work-groups and the overall dispatch. To define this the `nd_range` comprises two ranges: the whole range over which the kernel is to be executed, and the range of each work group.

The SYCL `nd_range` class template provides the common by-value semantics (see [Section 4.5.3](#)).

A synopsis of the SYCL `nd_range` class is provided below. The constructors and member functions of the

SYCL `nd_range` class are listed in [Table 91](#) and [Table 92](#) respectively. The additional common special member functions and common member functions are listed in [Section 4.5.3](#) in [Table 9](#) and [Table 10](#) respectively.

Table 91. Constructors of the `nd_range` class

Constructor	Description
<pre>nd_range&lt;Dimensions&gt;(   range&lt;Dimensions&gt; globalSize,   range&lt;Dimensions&gt; localSize,   id&lt;Dimensions&gt; offset = id&lt;Dimensions&gt;())</pre>	Construct an <code>nd_range</code> from the local and global constituent ranges. Supplying the option <code>offset</code> is deprecated in SYCL 2020. If the <code>offset</code> is not provided it will default to no offset.

Table 92. Member functions for the `nd_range` class

Member function	Description
<pre>range&lt;Dimensions&gt; get_global_range() const</pre>	Return the constituent global range.
<pre>range&lt;Dimensions&gt; get_local_range() const</pre>	Return the constituent local range.
<pre>range&lt;Dimensions&gt; get_group_range() const</pre>	Return a range representing the number of groups in each dimension. This range would result from <code>globalSize/localSize</code> as provided on construction.
<pre>id&lt;Dimensions&gt; get_offset() const // Deprecated in SYCL 2020.</pre>	Deprecated in SYCL 2020. Return the constituent offset.

#### 4.9.1.3. `id` class

`id<int Dimensions>` is a vector of `Dimensions` that is used to represent an `id` into a global or local `range`. It can be used as an index in an accessor of the same rank. The subscript operator (`operator[](n)`) returns the component `n` as a `size_t`.

The SYCL `id` class template provides the common by-value semantics (see [Section 4.5.3](#)).

A synopsis of the SYCL `id` class is provided below. The constructors, member functions and non-member functions of the SYCL `id` class are listed in [Table 93](#), [Table 94](#) and [Table 95](#) respectively. The additional common special member functions and common member functions are listed in [Section 4.5.3](#) in [Table 9](#) and [Table 10](#) respectively.

```
1 namespace sycl {
2   template <int Dimensions = 1> class id {
3   public:
4     static constexpr int dimensions = Dimensions;
5
6     id();
7
8     /* The following constructor is only available in the id class
9      * specialization where: Dimensions==1 */
10    id(size_t dim0);
11    /* The following constructor is only available in the id class
```

```

12  * specialization where: Dimensions==2 */
13  id(size_t dim0, size_t dim1);
14  /* The following constructor is only available in the id class
15  * specialization where: Dimensions==3 */
16  id(size_t dim0, size_t dim1, size_t dim2);
17
18  /* -- common interface members -- */
19
20  id(const range<Dimensions>& range);
21  id(const item<Dimensions>& item);
22
23  size_t get(int dimension) const;
24  size_t& operator[](int dimension);
25  size_t operator[](int dimension) const;
26
27  // only available if Dimensions == 1
28  operator size_t() const;
29
30  // OP is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=
31  friend id operatorOP(const id& lhs, const id& rhs) { /* ... */
32  }
33  friend id operatorOP(const id& lhs, const size_t& rhs) { /* ... */
34  }
35
36  // OP is: +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=
37  friend id& operatorOP(id& lhs, const id& rhs) { /* ... */
38  }
39  friend id& operatorOP(id& lhs, const size_t& rhs) { /* ... */
40  }
41
42  // OP is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=
43  friend id operatorOP(const size_t& lhs, const id& rhs) { /* ... */
44  }
45
46  // OP is unary +, -
47  friend id operatorOP(const id& rhs) { /* ... */
48  }
49
50  // OP is prefix ++, --
51  friend id& operatorOP(id& rhs) { /* ... */
52  }
53
54  // OP is postfix ++, --
55  friend id operatorOP(id& lhs, int) { /* ... */
56  }
57 };
58
59 // Deduction guides
60 id(size_t)->id<1>;
61 id(size_t, size_t)->id<2>;
62 id(size_t, size_t, size_t)->id<3>;
63
64 } // namespace sycl

```

Table 93. Constructors of the `id` class template

Constructor	Description
<code>id()</code>	Construct a SYCL <code>id</code> with the value <code>0</code> for each dimension.
<code>id(size_t dim0)</code>	Construct a 1D <code>id</code> with value <code>dim0</code> . Only valid when the template parameter <code>Dimensions</code> is equal to 1.
<code>id(size_t dim0, size_t dim1)</code>	Construct a 2D <code>id</code> with values <code>dim0</code> , <code>dim1</code> . Only valid when the template parameter <code>Dimensions</code> is equal to 2.
<code>id(size_t dim0, size_t dim1, size_t dim2)</code>	Construct a 3D <code>id</code> with values <code>dim0</code> , <code>dim1</code> , <code>dim2</code> . Only valid when the template parameter <code>Dimensions</code> is equal to 3.
<code>id(const range&lt;Dimensions&gt;&amp; range)</code>	Construct an <code>id</code> from the dimensions of <code>range</code> .
<code>id(const item&lt;Dimensions&gt;&amp; item)</code>	Construct an <code>id</code> from <code>item.get_id()</code> .

Table 94. Member functions of the `id` class template

Member function	Description
<code>size_t get(int dimension) const</code>	Return the value of the <code>id</code> for dimension <code>Dimension</code> .
<code>size_t&amp; operator[](int dimension)</code>	Return a reference to the requested dimension of the <code>id</code> object.
<code>size_t operator[](int dimension) const</code>	Return the value of the requested dimension of the <code>id</code> object.
<code>operator size_t() const</code>	Available only when: <code>Dimensions == 1</code> Returns the same value as <code>get(0)</code> .

Table 95. Hidden friend functions of the `id` class template

Hidden friend function	Description
<code>id operatorOP(const id&amp; lhs, const id&amp; rhs)</code>	<p>Where <code>OP</code> is: <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%</code>, <code>&lt;&lt;</code>, <code>&gt;&gt;</code>, <code>&amp;</code>, <code> </code>, <code>^</code>, <code>&amp;&amp;</code>, <code>  </code>, <code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code>, <code>&gt;=</code>.</p> <p>Constructs and returns a new instance of the SYCL <code>id</code> class template with the same dimensionality as <code>lhs id</code>, where each element of the new SYCL <code>id</code> instance is the result of an element-wise <code>OP</code> operator between each element of <code>lhs id</code> and each element of the <code>rhs id</code>. If the operator returns a <code>bool</code> the result is the cast to <code>size_t</code>.</p>

Hidden friend function	Description
<pre data-bbox="164 185 791 219"><code>id operatorOP(const id&amp; lhs, const size_t&amp; rhs)</code></pre>	<p data-bbox="1002 159 1460 230">Where <code>OP</code> is: <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%</code>, <code>&lt;&lt;</code>, <code>&gt;&gt;</code>, <code>&amp;</code>, <code> </code>, <code>^</code>, <code>&amp;&amp;</code>, <code>  </code>, <code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code>, <code>&gt;=</code>.</p> <p data-bbox="1002 259 1460 618">Constructs and returns a new instance of the SYCL <code>id</code> class template with the same dimensionality as <code>lhs id</code>, where each element of the new SYCL <code>id</code> instance is the result of an element-wise <code>OP</code> operator between each element of <code>lhs id</code> and the <code>rhs size_t</code>. If the operator returns a <code>bool</code> the result is the cast to <code>size_t</code>.</p>
<pre data-bbox="164 667 671 701"><code>id&amp; operatorOP(id&amp; lhs, const id&amp; rhs)</code></pre>	<p data-bbox="1002 640 1460 712">Where <code>OP</code> is: <code>+=</code>, <code>-=</code>, <code>*=</code>, <code>/=</code>, <code>%=</code>, <code>&lt;&lt;=</code>, <code>&gt;&gt;=</code>, <code>&amp;=</code>, <code> =</code>, <code>^=</code>.</p> <p data-bbox="1002 741 1460 987">Assigns each element of <code>lhs id</code> instance with the result of an element-wise <code>OP</code> operator between each element of <code>lhs id</code> and each element of the <code>rhs id</code> and returns <code>lhs id</code>. If the operator returns a <code>bool</code> the result is the cast to <code>size_t</code>.</p>
<pre data-bbox="164 1037 724 1070"><code>id&amp; operatorOP(id&amp; lhs, const size_t&amp; rhs)</code></pre>	<p data-bbox="1002 1010 1460 1081">Where <code>OP</code> is: <code>+=</code>, <code>-=</code>, <code>*=</code>, <code>/=</code>, <code>%=</code>, <code>&lt;&lt;=</code>, <code>&gt;&gt;=</code>, <code>&amp;=</code>, <code> =</code>, <code>^=</code>.</p> <p data-bbox="1002 1111 1460 1357">Assigns each element of <code>lhs id</code> instance with the result of an element-wise <code>OP</code> operator between each element of <code>lhs id</code> and the <code>rhs size_t</code> and returns <code>lhs id</code>. If the operator returns a <code>bool</code> the result is the cast to <code>size_t</code>.</p>
<pre data-bbox="164 1406 791 1440"><code>id operatorOP(const size_t&amp; lhs, const id&amp; rhs)</code></pre>	<p data-bbox="1002 1379 1460 1451">Where <code>OP</code> is: <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%</code>, <code>&lt;&lt;</code>, <code>&gt;&gt;</code>, <code>&amp;</code>, <code> </code>, <code>^</code>, <code>&amp;&amp;</code>, <code>  </code>, <code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code>, <code>&gt;=</code>.</p> <p data-bbox="1002 1480 1460 1827">Constructs and returns a new instance of the SYCL <code>id</code> class template with the same dimensionality as the <code>rhs SYCL id</code>, where each element of the new SYCL <code>id</code> instance is the result of an element-wise <code>OP</code> operator between the <code>lhs size_t</code> and each element of the <code>rhs SYCL id</code>. If the operator returns a <code>bool</code> the result is the cast to <code>size_t</code>.</p>

Hidden friend function	Description
<code>id operatorOP(const id&amp; rhs)</code>	Where <code>OP</code> is: unary <code>+</code> , unary <code>-</code> .  Constructs and returns a new instance of the SYCL <code>id</code> class template with the same dimensionality as the <code>rhs</code> SYCL <code>id</code> , where each element of the new SYCL <code>id</code> instance is the result of an element-wise <code>OP</code> operator on the <code>rhs</code> SYCL <code>id</code> .
<code>id&amp; operatorOP(id&amp; rhs)</code>	Where <code>OP</code> is: prefix <code>++</code> , prefix <code>--</code> .  Assigns each element of the <code>rhs id</code> instance with the result of an element-wise <code>OP</code> operator on each element of the <code>rhs id</code> and returns this <code>id</code> .
<code>id operatorOP(id&amp; lhs, int)</code>	Where <code>OP</code> is: postfix <code>++</code> , postfix <code>--</code> .  Make a copy of the <code>lhs id</code> . Assigns each element of the <code>lhs id</code> instance with the result of an element-wise <code>OP</code> operator on each element of the <code>lhs id</code> . Then return the initial copy of the <code>id</code> .

#### 4.9.1.4. `item` class

`item` identifies an instance of the function object executing at each point in a `range`. It is passed to a `parallel_for` call or returned by member functions of `h_item`. It encapsulates enough information to identify the work-item's range of possible values and its ID in that range. It can optionally carry the offset of the range if provided to the `parallel_for`; note this is deprecated in SYCL 2020. Instances of the `item` class are not user-constructible and are passed by the runtime to each instance of the function object.

The SYCL `item` class template provides the common by-value semantics (see [Section 4.5.3](#)).

A synopsis of the SYCL `item` class is provided below. The member functions of the SYCL `item` class are listed in [Table 94](#). The additional common special member functions and common member functions are listed in [Section 4.5.3](#) in [Table 9](#) and [Table 10](#) respectively.

```

1 namespace sycl {
2 template <int Dimensions = 1, bool WithOffset = true> class item {
3 public:
4     static constexpr int dimensions = Dimensions;
5
6     item() = delete;
7
8     /* -- common interface members -- */
9
10    id<Dimensions> get_id() const;
11
12    size_t get_id(int dimension) const;
13
14    size_t operator[](int dimension) const;
15

```



```

16  range<Dimensions> get_range() const;
17
18  size_t get_range(int dimension) const;
19
20  // Deprecated in SYCL 2020.
21  // only available if WithOffset is true
22  id<Dimensions> get_offset() const;
23
24  // Deprecated in SYCL 2020.
25  // only available if WithOffset is false
26  operator item<Dimensions, true>() const;
27
28  // only available if Dimensions == 1
29  operator size_t() const;
30
31  size_t get_linear_id() const;
32 };
33 } // namespace sycl

```

Table 96. Member functions for the `item` class

Member function	Description
<code>id&lt;Dimensions&gt; get_id() const</code>	Return the constituent <code>id</code> representing the work-item's position in the iteration space.
<code>size_t get_id(int dimension) const</code>	Return the same value as <code>get_id()[dimension]</code> .
<code>size_t operator[] (int dimension) const</code>	Return the same value as <code>get_id(dimension)</code> .
<code>range&lt;Dimensions&gt; get_range() const</code>	Returns a <code>range</code> representing the dimensions of the range of possible values of the <code>item</code> .
<code>size_t get_range(int dimension) const</code>	Return the same value as <code>get_range().get(dimension)</code> .
<code>id&lt;Dimensions&gt; get_offset() const</code> <code>// Deprecated in SYCL 2020.</code>	<p>Deprecated in SYCL 2020. Returns an <code>id</code> representing the <math>n</math>-dimensional offset provided to the <code>parallel_for</code> and that is added by the runtime to the global-ID of each work-item, if this item represents a global range. For an item converted from an item with no offset this will always return an <code>id</code> of all 0 values.</p> <p>This member function is only available if <code>WithOffset</code> is <code>true</code>.</p>

Member function	Description
<pre>operator item&lt;Dimensions, true&gt;() const // Deprecated in SYCL 2020.</pre>	<p>Deprecated in SYCL 2020.</p> <p>Available only when: <code>WithOffset == false</code></p> <p>Returns an <code>item</code> representing the same information as the object holds but also includes the offset set to 0. This conversion allow users to seamlessly write code that assumes an offset and still provides an offset-less <code>item</code>.</p>
<pre>operator size_t() const</pre>	<p>Available only when: <code>Dimensions == 1</code></p> <p>Returns the same value as <code>get_id(0)</code>.</p>
<pre>size_t get_linear_id() const</pre>	<p>Return the id as a linear index value. Calculating a linear address from the multi-dimensional index follows <a href="#">Section 3.11.1</a>.</p>

#### 4.9.1.5. `nd_item` class

`nd_item<int Dimensions>` identifies an instance of the function object executing at each point in an `nd_range<int Dimensions>` passed to a `parallel_for` call. It encapsulates enough information to identify the `work-item`'s local and global `ids`, the `work-group id` and also provides access to the `group` and `sub-group` classes. Instances of the `nd_item<int Dimensions>` class are not user-constructible and are passed by the runtime to each instance of the function object.

The SYCL `nd_item` class template provides the common by-value semantics (see [Section 4.5.3](#)).

A synopsis of the SYCL `nd_item` class is provided below. The member functions of the SYCL `nd_item` class are listed in [Table 97](#). The additional common special member functions and common member functions are listed in [Section 4.5.3](#) in [Table 9](#) and [Table 10](#) respectively.

```

1 namespace sycl {
2   template <int Dimensions = 1> class nd_item {
3   public:
4     static constexpr int dimensions = Dimensions;
5
6     nd_item() = delete;
7
8     /* -- common interface members -- */
9
10    id<Dimensions> get_global_id() const;
11
12    size_t get_global_id(int dimension) const;
13
14    size_t get_global_linear_id() const;
15
16    id<Dimensions> get_local_id() const;
17
18    size_t get_local_id(int dimension) const;
19
```

```

20  size_t get_local_linear_id() const;
21
22  group<Dimensions> get_group() const;
23
24  sub_group get_sub_group() const;
25
26  size_t get_group(int dimension) const;
27
28  size_t get_group_linear_id() const;
29
30  range<Dimensions> get_group_range() const;
31
32  size_t get_group_range(int dimension) const;
33
34  range<Dimensions> get_global_range() const;
35
36  size_t get_global_range(int dimension) const;
37
38  range<Dimensions> get_local_range() const;
39
40  size_t get_local_range(int dimension) const;
41
42  // Deprecated in SYCL 2020.
43  id<Dimensions> get_offset() const;
44
45  nd_range<Dimensions> get_nd_range() const;
46
47  // Deprecated in SYCL 2020.
48  template <typename DataT>
49  device_event async_work_group_copy(local_ptr<DataT> dest,
50                                     global_ptr<DataT> src,
51                                     size_t numElements) const;
52
53  // Deprecated in SYCL 2020.
54  template <typename DataT>
55  device_event async_work_group_copy(global_ptr<DataT> dest,
56                                     local_ptr<DataT> src,
57                                     size_t numElements) const;
58
59  // Deprecated in SYCL 2020.
60  template <typename DataT>
61  device_event async_work_group_copy(local_ptr<DataT> dest,
62                                     global_ptr<DataT> src,
63                                     size_t numElements,
64                                     size_t srcStride) const;
65
66  // Deprecated in SYCL 2020.
67  template <typename DataT>
68  device_event async_work_group_copy(global_ptr<DataT> dest,
69                                     local_ptr<DataT> src,
70                                     size_t numElements,
71                                     size_t destStride) const;
72
73  /* Available only when: (std::is_same_v<DestDataT,
74     std::remove_const_t<SrcDataT>> == true) */
75  template <typename DestDataT, typename SrcDataT>

```

```

76  device_event async_work_group_copy(decorated_local_ptr<DestDataT> dest,
77                                   decorated_global_ptr<SrcDataT> src,
78                                   size_t numElements) const;
79
80  /* Available only when: (std::is_same_v<DestDataT,
81     std::remove_const_t<SrcDataT>> == true) */
82  template <typename DestDataT, typename SrcDataT>
83  device_event async_work_group_copy(decorated_global_ptr<DestDataT> dest,
84                                   decorated_local_ptr<SrcDataT> src,
85                                   size_t numElements) const;
86
87  /* Available only when: (std::is_same_v<DestDataT,
88     std::remove_const_t<SrcDataT>> == true) */
89  template <typename DestDataT, typename SrcDataT>
90  device_event async_work_group_copy(decorated_local_ptr<DestDataT> dest,
91                                   decorated_global_ptr<SrcDataT> src,
92                                   size_t numElements,
93                                   size_t srcStride) const;
94
95  /* Available only when: (std::is_same_v<DestDataT,
96     std::remove_const_t<SrcDataT>> == true) */
97  template <typename DestDataT, typename SrcDataT>
98  device_event async_work_group_copy(decorated_global_ptr<DestDataT> dest,
99                                   decorated_local_ptr<SrcDataT> src,
100                                   size_t numElements,
101                                   size_t destStride) const;
102
103  template <typename... EventTN> void wait_for(EventTN... events) const;
104 };
105 } // namespace sycl

```

Table 97. Member functions for the `nd_item` class

Member function	Description
<code>id&lt;Dimensions&gt; get_global_id() const</code>	Return the constituent <a href="#">global id</a> representing the work-item's position in the global iteration space.
<code>size_t get_global_id(int dimension) const</code>	Return the constituent element of the <a href="#">global id</a> representing the work-item's position in the <a href="#">nd-range</a> in the given <a href="#">Dimension</a> .
<code>size_t get_global_linear_id() const</code>	Return the constituent <a href="#">global id</a> as a linear index value, representing the work-item's position in the global iteration space. The linear address is calculated from the multi-dimensional index by first subtracting the offset and then following <a href="#">Section 3.11.1</a> .
<code>id&lt;Dimensions&gt; get_local_id() const</code>	Return the constituent <a href="#">local id</a> representing the work-item's position within the current <a href="#">work-group</a> .

Member function	Description
<code>size_t get_local_id(int dimension) const</code>	Return the constituent element of the <code>local id</code> representing the work-item's position within the current <code>work-group</code> in the given <code>Dimension</code> .
<code>size_t get_local_linear_id() const</code>	Return the constituent <code>local id</code> as a linear index value, representing the work-item's position within the current <code>work-group</code> . The linear address is calculated from the multi-dimensional index following <a href="#">Section 3.11.1</a> .
<code>group&lt;Dimensions&gt; get_group() const</code>	Return the constituent <code>work-group</code> , <code>group</code> representing the <code>work-group</code> 's position within the overall <code>nd-range</code> .
<code>sub_group get_sub_group() const</code>	Return a <code>sub_group</code> representing the <code>sub-group</code> to which the work-item belongs.
<code>size_t get_group_id(int dimension) const</code>	Return the constituent element of the <code>group id</code> representing the <code>work-group</code> 's position within the overall <code>nd_range</code> in the given <code>Dimension</code> .
<code>size_t get_group_linear_id() const</code>	Return the <code>group id</code> as a linear index value. Calculating a linear address from a multi-dimensional index follows <a href="#">Section 3.11.1</a> .
<code>range&lt;Dimensions&gt; get_group_range() const</code>	Returns the number of <code>work-groups</code> in the iteration space.
<code>size_t get_group_range(int dimension) const</code>	Return the number of <code>work-groups</code> for <code>Dimension</code> in the iteration space.
<code>range&lt;Dimensions&gt; get_global_range() const</code>	Returns a <code>range</code> representing the dimensions of the global iteration space.
<code>size_t get_global_range(int dimension) const</code>	Return the same value as <code>get_global_range().get(dimension)</code> .
<code>range&lt;Dimensions&gt; get_local_range() const</code>	Returns a <code>range</code> representing the dimensions of the current <code>work-group</code> .
<code>size_t get_local_range(int dimension) const</code>	Return the same value as <code>get_local_range().get(dimension)</code> .

Member function	Description
<pre>id&lt;Dimensions&gt; get_offset() const // Deprecated in SYCL 2020.</pre>	Deprecated in SYCL 2020. Returns an <code>id</code> representing the n-dimensional offset provided to the constructor of the <code>nd_range</code> and that is added by the runtime to the <code>global id</code> of each <code>work-item</code> .
<pre>nd_range&lt;Dimensions&gt; get_nd_range() const</pre>	Returns the <code>nd_range</code> of the current execution.
<pre>template &lt;typename DataT&gt; device_event async_work_group_copy(local_ptr&lt;DataT&gt; dest,                                    global_ptr&lt;DataT&gt; src,                                    size_t numElements) const</pre>	Deprecated in SYCL 2020. Has the same effect as the overload taking <code>decorated_local_ptr</code> and <code>decorated_global_ptr</code> except that the dest and src parameters are <code>multi_ptr</code> with <code>access::decorated::legacy</code> .
<pre>template &lt;typename DataT&gt; device_event async_work_group_copy(global_ptr&lt;DataT&gt; dest,                                    local_ptr&lt;DataT&gt; src,                                    size_t numElements) const</pre>	Deprecated in SYCL 2020. Has the same effect as the overload taking <code>decorated_local_ptr</code> and <code>decorated_global_ptr</code> except that the dest and src parameters are <code>multi_ptr</code> with <code>access::decorated::legacy</code> .
<pre>template &lt;typename DataT&gt; device_event async_work_group_copy(local_ptr&lt;DataT&gt; dest,                                    global_ptr&lt;DataT&gt; src,                                    size_t numElements,                                    size_t srcStride) const</pre>	Deprecated in SYCL 2020. Has the same effect as the overload taking <code>decorated_local_ptr</code> and <code>decorated_global_ptr</code> except that the dest and src parameters are <code>multi_ptr</code> with <code>access::decorated::legacy</code> .
<pre>template &lt;typename DataT&gt; device_event async_work_group_copy(global_ptr&lt;DataT&gt; dest,                                    local_ptr&lt;DataT&gt; src,                                    size_t numElements,                                    size_t destStride) const</pre>	Deprecated in SYCL 2020. Has the same effect as the overload taking <code>decorated_local_ptr</code> and <code>decorated_global_ptr</code> except that the dest and src parameters are <code>multi_ptr</code> with <code>access::decorated::legacy</code> .
<pre>template &lt;typename DestDataT, typename SrcDataT&gt; device_event async_work_group_copy(decorated_local_ptr &lt;DestDataT&gt; dest,                                    decorated_global_ptr &lt;SrcDataT&gt; src,                                    size_t numElements) const</pre>	<p>Available only when: <code>(std::is_same_v&lt;DestDataT, std::remove_const_t&lt;SrcDataT&gt;&gt; == true)</code></p> <p>Permitted types for <code>DataT</code> are all scalar and vector types. Asynchronously copies a number of elements specified by <code>numElements</code> from the source pointer <code>src</code> to destination pointer <code>dest</code> and returns a SYCL <code>device_event</code> which can be used to wait on the completion of the copy.</p>

Member function	Description
<pre>template &lt;typename DestDataT, typename SrcDataT&gt; device_event async_work_group_copy(decorated_global_ptr &lt;DestDataT&gt; dest,                                    decorated_local_ptr &lt;SrcDataT&gt; src,                                    size_t numElements) const</pre>	<p>Available only when: (std::is_same_v&lt;DestDataT, std::remove_const_t&lt;SrcDataT&gt;&gt; == true)</p> <p>Permitted types for <code>DataT</code> are all scalar and vector types. Asynchronously copies a number of elements specified by <code>numElements</code> from the source pointer <code>src</code> to destination pointer <code>dest</code> and returns a SYCL <code>device_event</code> which can be used to wait on the completion of the copy.</p>
<pre>template &lt;typename DestDataT, typename SrcDataT&gt; device_event async_work_group_copy(decorated_local_ptr &lt;DestDataT&gt; dest,                                    decorated_global_ptr &lt;SrcDataT&gt; src,                                    size_t numElements, size_t srcStride) const</pre>	<p>Available only when: (std::is_same_v&lt;DestDataT, std::remove_const_t&lt;SrcDataT&gt;&gt; == true)</p> <p>Permitted types for <code>DataT</code> are all scalar and vector types. Asynchronously copies a number of elements specified by <code>numElements</code> from the source pointer <code>src</code> to destination pointer <code>dest</code> with a source stride specified by <code>srcStride</code> and returns a SYCL <code>device_event</code> which can be used to wait on the completion of the copy.</p>
<pre>template &lt;typename DestDataT, SrcDataT&gt; device_event async_work_group_copy(decorated_global_ptr &lt;DestDataT&gt; dest,                                    decorated_local_ptr &lt;SrcDataT&gt; src,                                    size_t numElements, size_t destStride) const</pre>	<p>Available only when: (std::is_same_v&lt;DestDataT, std::remove_const_t&lt;SrcDataT&gt;&gt; == true)</p> <p>Permitted types for <code>DataT</code> are all scalar and vector types. Asynchronously copies a number of elements specified by <code>numElements</code> from the source pointer <code>src</code> to destination pointer <code>dest</code> with a destination stride specified by <code>destStride</code> and returns a SYCL <code>device_event</code> which can be used to wait on the completion of the copy.</p>
<pre>template &lt;typename... EventTN&gt; void wait_for(EventTN... events) const</pre>	<p>Permitted type for <code>EventTN</code> is <code>device_event</code>. Waits for the asynchronous operations associated with each <code>device_event</code> to complete.</p>

#### 4.9.1.6. `h_item` class (deprecated)

The `h_item` class is deprecated in SYCL 2020.

`h_item<int Dimensions>` identifies an instance of a `group::parallel_for_work_item` function object executing at each point in a local `range<int Dimensions>` passed to a `parallel_for_work_item` call or to the corresponding `parallel_for_work_group` call if no `range` is passed to the `parallel_for_work_item` call. It encapsulates enough information to identify the `work-item`'s local and global `items` according to the information given to `parallel_for_work_group` (physical ids) as well as the `work-item`'s logical local `items` in the logical local range. All returned `items` objects are offset-less. Instances of the `h_item<int Dimensions>` class are not user-constructible and are passed by the runtime to each instance of the function object.

The SYCL `h_item` class template provides the common by-value semantics (see [Section 4.5.3](#)).

A synopsis of the SYCL `h_item` class is provided below. The member functions of the SYCL `h_item` class are listed in [Table 98](#). The additional common special member functions and common member functions are listed in [Section 4.5.3](#) in [Table 9](#) and [Table 10](#) respectively.

```

1 namespace sycl {
2  /* Deprecated in SYCL 2020 */
3  template <int Dimensions> class h_item {
4  public:
5      static constexpr int dimensions = Dimensions;
6
7      h_item() = delete;
8
9      /* -- common interface members -- */
10
11      item<Dimensions, false> get_global() const;
12
13      item<Dimensions, false> get_local() const;
14
15      item<Dimensions, false> get_logical_local() const;
16
17      item<Dimensions, false> get_physical_local() const;
18
19      range<Dimensions> get_global_range() const;
20
21      size_t get_global_range(int dimension) const;
22
23      id<Dimensions> get_global_id() const;
24
25      size_t get_global_id(int dimension) const;
26
27      range<Dimensions> get_local_range() const;
28
29      size_t get_local_range(int dimension) const;
30
31      id<Dimensions> get_local_id() const;
32
33      size_t get_local_id(int dimension) const;
34
35      range<Dimensions> get_logical_local_range() const;
36
37      size_t get_logical_local_range(int dimension) const;
38
39      id<Dimensions> get_logical_local_id() const;
40
41      size_t get_logical_local_id(int dimension) const;
42

```



```

43  range<Dimensions> get_physical_local_range() const;
44
45  size_t get_physical_local_range(int dimension) const;
46
47  id<Dimensions> get_physical_local_id() const;
48
49  size_t get_physical_local_id(int dimension) const;
50 };
51 } // namespace sycl

```

Table 98. Member functions for the `h_item` class

Member function	Description
<code>item&lt;Dimensions, false&gt; get_global() const</code>	Return the constituent global <code>item</code> representing the work-item's position in the global iteration space as provided upon kernel invocation.
<code>item&lt;Dimensions, false&gt; get_local() const</code>	Return the same value as <code>get_logical_local()</code> .
<code>item&lt;Dimensions, false&gt; get_logical_local() const</code>	<p>Return the constituent element of the logical local <code>item</code> work-item's position in the local iteration space as provided upon the invocation of the <code>group::parallel_for_work_item</code>.</p> <p>If the <code>group::parallel_for_work_item</code> was called without any logical local range then the member function returns the physical local <code>item</code>.</p> <p>A physical id can be computed from a logical id by getting the remainder of the integer division of the logical id and the physical range: <code>get_logical_local().get() % get_physical_local.get_range() == get_physical_local().get()</code>.</p>
<code>item&lt;Dimensions, false&gt; get_physical_local() const</code>	Return the constituent element of the physical local <code>item</code> work-item's position in the local iteration space as provided (by the user or the runtime) upon the kernel invocation.
<code>range&lt;Dimensions&gt; get_global_range() const</code>	Return the same value as <code>get_global().get_range()</code>
<code>size_t get_global_range(int dimension) const</code>	Return the same value as <code>get_global().get_range(dimension)</code>
<code>id&lt;Dimensions&gt; get_global_id() const</code>	Return the same value as <code>get_global().get_id()</code>

Member function	Description
<code>size_t get_global_id(int dimension) const</code>	Return the same value as <code>get_global().get_id(dimension)</code>
<code>range&lt;Dimensions&gt; get_local_range() const</code>	Return the same value as <code>get_local().get_range()</code>
<code>size_t get_local_range(int dimension) const</code>	Return the same value as <code>get_local().get_range(dimension)</code>
<code>id&lt;Dimensions&gt; get_local_id() const</code>	Return the same value as <code>get_local().get_id()</code>
<code>size_t get_local_id(int dimension) const</code>	Return the same value as <code>get_local().get_id(dimension)</code>
<code>range&lt;Dimensions&gt; get_logical_local_range() const</code>	Return the same value as <code>get_logical_local().get_range()</code>
<code>size_t get_logical_local_range(int dimension) const</code>	Return the same value as <code>get_logical_local().get_range(dimension)</code>
<code>id&lt;Dimensions&gt; get_logical_local_id() const</code>	Return the same value as <code>get_logical_local().get_id()</code>
<code>size_t get_logical_local_id(int dimension) const</code>	Return the same value as <code>get_logical_local().get_id(dimension)</code>
<code>range&lt;Dimensions&gt; get_physical_local_range() const</code>	Return the same value as <code>get_physical_local().get_range()</code>
<code>size_t get_physical_local_range(int dimension) const</code>	Return the same value as <code>get_physical_local().get_range(dimension)</code>
<code>id&lt;Dimensions&gt; get_physical_local_id() const</code>	Return the same value as <code>get_physical_local().get_id()</code>
<code>size_t get_physical_local_id(int dimension) const</code>	Return the same value as <code>get_physical_local().get_id(dimension)</code>

#### 4.9.1.7. `group` class

The `group<int Dimensions>` encapsulates all functionality required to represent a particular `work-group` within a parallel execution. It is not user-constructible.

The local range stored in the `group` class is provided either by the programmer, when it is passed as an optional parameter to `parallel_for_work_group`, or by the runtime system when it selects the optimal work-group size. This allows the developer to always know how many work-items are in each executing work-group, even through the abstracted iteration range of the `parallel_for_work_item` loops.

The SYCL `group` class template provides the common by-value semantics (see [Section 4.5.3](#)).

A synopsis of the SYCL `group` class is provided below. The member functions of the SYCL `group` class are listed in [Table 99](#). The additional common special member functions and common member functions are listed in [Section 4.5.3](#) in [Table 9](#) and [Table 10](#) respectively.

```

1 namespace sycl {
2 template <int Dimensions = 1> class group {
3 public:
4     using id_type = id<Dimensions>;
5     using range_type = range<Dimensions>;
6     using linear_id_type = size_t;
7     static constexpr int dimensions = Dimensions;
8     static constexpr memory_scope fence_scope = memory_scope::work_group;
9
10    /* -- common interface members -- */
11
12    id<Dimensions> get_group_id() const;
13
14    size_t get_group_id(int dimension) const;
15
16    id<Dimensions> get_local_id() const;
17
18    size_t get_local_id(int dimension) const;
19
20    range<Dimensions> get_local_range() const;
21
22    size_t get_local_range(int dimension) const;
23
24    range<Dimensions> get_group_range() const;
25
26    size_t get_group_range(int dimension) const;
27
28    range<Dimensions> get_max_local_range() const;
29
30    size_t operator[](int dimension) const;
31
32    size_t get_group_linear_id() const;
33
34    size_t get_local_linear_id() const;
35
36    size_t get_group_linear_range() const;
37
38    size_t get_local_linear_range() const;
39
40    bool leader() const;
41
42    // Deprecated in SYCL 2020.
43    template <typename WorkItemFunctionT>
44    void parallel_for_work_item(const WorkItemFunctionT& func) const;
45
46    // Deprecated in SYCL 2020.
47    template <typename WorkItemFunctionT>
48    void parallel_for_work_item(range<Dimensions> logicalRange,
49                               const WorkItemFunctionT& func) const;
50
51    // Deprecated in SYCL 2020.
52    template <typename DataT>
53    device_event async_work_group_copy(local_ptr<DataT> dest,
54                                       global_ptr<DataT> src,
55                                       size_t numElements) const;

```

```

56
57 // Deprecated in SYCL 2020.
58 template <typename DataT>
59 device_event async_work_group_copy(global_ptr<DataT> dest,
60                                   local_ptr<DataT> src,
61                                   size_t numElements) const;
62
63 // Deprecated in SYCL 2020.
64 template <typename DataT>
65 device_event async_work_group_copy(local_ptr<DataT> dest,
66                                   global_ptr<DataT> src,
67                                   size_t numElements,
68                                   size_t srcStride) const;
69
70 // Deprecated in SYCL 2020.
71 template <typename DataT>
72 device_event async_work_group_copy(global_ptr<DataT> dest,
73                                   local_ptr<DataT> src,
74                                   size_t numElements,
75                                   size_t destStride) const;
76
77 /* Available only when: (std::is_same_v<DestDataT,
78   std::remove_const_t<SrcDataT>> == true) */
79 template <typename DestDataT, typename SrcDataT>
80 device_event async_work_group_copy(decorated_local_ptr<DestDataT> dest,
81                                   decorated_global_ptr<SrcDataT> src,
82                                   size_t numElements) const;
83
84 /* Available only when: (std::is_same_v<DestDataT,
85   std::remove_const_t<SrcDataT>> == true) */
86 template <typename DestDataT, typename SrcDataT>
87 device_event async_work_group_copy(decorated_global_ptr<DestDataT> dest,
88                                   decorated_local_ptr<SrcDataT> src,
89                                   size_t numElements) const;
90
91 /* Available only when: (std::is_same_v<DestDataT,
92   std::remove_const_t<SrcDataT>> == true) */
93 template <typename DestDataT, typename SrcDataT>
94 device_event async_work_group_copy(decorated_local_ptr<DestDataT> dest,
95                                   decorated_global_ptr<SrcDataT> src,
96                                   size_t numElements,
97                                   size_t srcStride) const;
98
99 /* Available only when: (std::is_same_v<DestDataT,
100   std::remove_const_t<SrcDataT>> == true) */
101 template <typename DestDataT, typename SrcDataT>
102 device_event async_work_group_copy(decorated_global_ptr<DestDataT> dest,
103                                   decorated_local_ptr<SrcDataT> src,
104                                   size_t numElements,
105                                   size_t destStride) const;
106
107 template <typename... EventTN> void wait_for(EventTN... events) const;
108 };
109 } // namespace sycl

```

Table 99. Member functions for the `group` class

Member function	Description
<code>id&lt;Dimensions&gt; get_group_id() const</code>	Return an <code>id</code> representing the index of the work-group within the global <code>nd-range</code> for every dimension. Since the work-items in a work-group have a defined position within the global <code>nd-range</code> , the returned group id can be used along with the local id to uniquely identify the work-item in the global <code>nd-range</code> .
<code>size_t get_group_id(int dimension) const</code>	Return the same value as <code>get_group_id()[dimension]</code> .
<code>id&lt;Dimensions&gt; get_local_id() const</code>	Return a SYCL <code>id</code> representing the calling work-item's position within the <code>work-group</code> .  It is undefined behavior for this member function to be invoked from within a <code>parallel_for_work_item</code> context.
<code>size_t get_local_id(int dimension) const</code>	Return the same value as <code>get_local_id()[dimension]</code> .  It is undefined behavior for this member function to be invoked from within a <code>parallel_for_work_item</code> context.
<code>range&lt;Dimensions&gt; get_local_range() const</code>	Return a SYCL <code>range</code> representing all dimensions of the local range. This local range may have been provided by the programmer, or chosen by the SYCL runtime.
<code>size_t get_local_range(int dimension) const</code>	Return the same value as <code>get_local_range()[dimension]</code> .
<code>range&lt;Dimensions&gt; get_group_range() const</code>	Return a <code>range</code> representing the number of <code>work-groups</code> in the <code>nd_range</code> .
<code>size_t get_group_range(int dimension) const</code>	Return the same value as <code>get_group_range()[dimension]</code> .
<code>size_t operator[](int dimension) const</code>	Return the same value as <code>get_group_id(dimension)</code> .
<code>range&lt;Dimensions&gt; get_max_local_range() const</code>	Return a <code>range</code> representing the maximum number of work-items in any <code>work-group</code> in the <code>nd_range</code> .
<code>size_t get_group_linear_id() const</code>	Get a linearized version of the <code>work-group id</code> . Calculating a linear <code>work-group id</code> from a multi-dimensional index follows <a href="#">Section 3.11.1</a> .

Member function	Description
<code>size_t get_group_linear_range() const</code>	Return the total number of <code>work-groups</code> in the <code>nd_range</code> .
<code>size_t get_local_linear_id() const</code>	<p>Get a linearized version of the calling work-item's <code>local id</code>. Calculating a linear <code>local id</code> from a multi-dimensional index follows <a href="#">Section 3.11.1</a>.</p> <p>It is undefined behavior for this member function to be invoked from within a <code>parallel_for_work_item</code> context.</p>
<code>size_t get_local_linear_range() const</code>	Return the total number of work-items in the <code>work-group</code> .
<code>bool leader() const</code>	<p>Return true for exactly one work-item in the <code>work-group</code>, if the calling work-item is the leader of the work-group, and false for all other work-items in the work-group.</p> <p>The leader of the work-group is determined during construction of the work-group, and is invariant for the lifetime of the work-group. The leader of the work-group is guaranteed to be the work-item with a local id of 0.</p>
<pre>template &lt;typename WorkItemFunctionT&gt; void parallel_for_work_item(const WorkItemFunctionT&amp; func) const</pre>	<p>Deprecated in SYCL 2020. Launch the work-items for this work-group.</p> <p><code>func</code> is a function object type with a public member function <code>void F::operator()(h_item&lt;Dimensions&gt;)</code> representing the work-item computation.</p> <p>This member function can only be invoked within a <code>parallel_for_work_group</code> context. It is undefined behavior for this member function to be invoked from within the <code>parallel_for_work_group</code> form that does not define work-group size, because then the number of work-items that should execute the code is not defined. It is expected that this form of <code>parallel_for_work_item</code> is invoked within the <code>parallel_for_work_group</code> form that specifies the size of a work-group.</p>

Member function	Description
<pre>template &lt;typename WorkItemFunctionT&gt; void parallel_for_work_item(range&lt;Dimensions&gt; logicalRange,                            const WorkItemFunctionT&amp; func) const</pre>	<p>Deprecated in SYCL 2020. Launch the work-items for this work-group using a logical local range. The function object <code>func</code> is executed as if the kernel were invoked with <code>logicalRange</code> as the local range. This new local range is emulated and may not map one-to-one with the physical range.</p> <p><code>logicalRange</code> is the new local range to be used. This range can be smaller or larger than the one used to invoke the kernel. <code>func</code> is a function object type with a public member function <code>void F::operator()(h_item&lt;Dimensions&gt;)</code> representing the work-item computation.</p> <p>Note that the logical range does not need to be uniform across all work-groups in a kernel. For example the logical range may depend on a work-group varying query (e.g. <code>group::get_linear_id</code>), such that different work-groups in the same kernel invocation execute different logical range sizes.</p> <p>This member function can only be invoked within a <code>parallel_for_work_group</code> context.</p>
<pre>template &lt;typename DataT&gt; device_event async_work_group_copy(local_ptr&lt;DataT&gt; dest,                                    global_ptr&lt;DataT&gt; src,                                    size_t numElements) const</pre>	<p>Deprecated in SYCL 2020. Has the same effect as the overload taking <code>decorated_local_ptr</code> and <code>decorated_global_ptr</code> except that the dest and src parameters are <code>multi_ptr</code> with <code>access::decorated::legacy</code>.</p>
<pre>template &lt;typename DataT&gt; device_event async_work_group_copy(global_ptr&lt;DataT&gt; dest,                                    local_ptr&lt;DataT&gt; src,                                    size_t numElements) const</pre>	<p>Deprecated in SYCL 2020. Has the same effect as the overload taking <code>decorated_local_ptr</code> and <code>decorated_global_ptr</code> except that the dest and src parameters are <code>multi_ptr</code> with <code>access::decorated::legacy</code>.</p>
<pre>template &lt;typename DataT&gt; device_event async_work_group_copy(local_ptr&lt;DataT&gt; dest,                                    global_ptr&lt;DataT&gt; src,                                    size_t numElements,                                    size_t srcStride) const</pre>	<p>Deprecated in SYCL 2020. Has the same effect as the overload taking <code>decorated_local_ptr</code> and <code>decorated_global_ptr</code> except that the dest and src parameters are <code>multi_ptr</code> with <code>access::decorated::legacy</code>.</p>

Member function	Description
<pre> template &lt;typename DataT&gt; device_event async_work_group_copy(global_ptr&lt;DataT&gt; dest,                                    local_ptr&lt;DataT&gt; src,                                    size_t numElements,                                    size_t destStride) const </pre>	<p>Deprecated in SYCL 2020. Has the same effect as the overload taking <code>decorated_local_ptr</code> and <code>decorated_global_ptr</code> except that the <code>dest</code> and <code>src</code> parameters are <code>multi_ptr</code> with <code>access::decorated::legacy</code>.</p>
<pre> template &lt;typename DestDataT, typename SrcDataT&gt; device_event async_work_group_copy(decorated_global_ptr &lt;DestDataT&gt; dest,                                    decorated_local_ptr &lt;SrcDataT&gt; src,                                    size_t numElements) const </pre>	<p>Available only when: <code>(std::is_same_v&lt;DestDataT, std::remove_const_t&lt;SrcDataT&gt;&gt; == true)</code></p> <p>Permitted types for <code>DataT</code> are all scalar and vector types. Asynchronously copies a number of elements specified by <code>numElements</code> from the source pointer <code>src</code> to destination pointer <code>dest</code> and returns a SYCL <code>device_event</code> which can be used to wait on the completion of the copy.</p>
<pre> template &lt;typename DestDataT, typename SrcDataT&gt; device_event async_work_group_copy(decorated_local_ptr &lt;DestDataT&gt; dest,                                    decorated_global_ptr &lt;SrcDataT&gt; src,                                    size_t numElements,                                    size_t srcStride) const </pre>	<p>Available only when: <code>(std::is_same_v&lt;DestDataT, std::remove_const_t&lt;SrcDataT&gt;&gt; == true)</code></p> <p>Permitted types for <code>DataT</code> are all scalar and vector types. Asynchronously copies a number of elements specified by <code>numElements</code> from the source pointer <code>src</code> to destination pointer <code>dest</code> with a source stride specified by <code>srcStride</code> and returns a SYCL <code>device_event</code> which can be used to wait on the completion of the copy.</p>
<pre> template &lt;typename DestDataT, SrcDataT&gt; device_event async_work_group_copy(decorated_global_ptr &lt;DestDataT&gt; dest,                                    decorated_local_ptr &lt;SrcDataT&gt; src,                                    size_t numElements,                                    size_t destStride) const </pre>	<p>Available only when: <code>(std::is_same_v&lt;DestDataT, std::remove_const_t&lt;SrcDataT&gt;&gt; == true)</code></p> <p>Permitted types for <code>DataT</code> are all scalar and vector types. Asynchronously copies a number of elements specified by <code>numElements</code> from the source pointer <code>src</code> to destination pointer <code>dest</code> with a destination stride specified by <code>destStride</code> and returns a SYCL <code>device_event</code> which can be used to wait on the completion of the copy.</p>



Member function	Description
<pre>template &lt;typename... EventTN&gt; void wait_for(EventTN... events) const</pre>	Permitted type for <code>EventTN</code> is <code>device_event</code> . Waits for the asynchronous operations associated with each <code>device_event</code> to complete.

#### 4.9.1.8. `sub_group` class

The `sub_group` class encapsulates all functionality required to represent a particular `sub-group` within a parallel execution. It is not user-constructible.

The SYCL `sub_group` class provides the common by-value semantics (see [Section 4.5.3](#)).

A synopsis of the SYCL `sub_group` class is provided below. The member functions of the SYCL `sub_group` class are listed in [Table 100](#). The additional common special member functions and common member functions are listed in [Section 4.5.3](#) in [Table 9](#) and [Table 10](#) respectively.

```
1 namespace sycl {
2   class sub_group {
3   public:
4     using id_type = id<1>;
5     using range_type = range<1>;
6     using linear_id_type = uint32_t;
7     static constexpr int dimensions = 1;
8     static constexpr memory_scope fence_scope = memory_scope::sub_group;
9
10    /* -- common interface members -- */
11
12    id<1> get_group_id() const;
13
14    id<1> get_local_id() const;
15
16    range<1> get_local_range() const;
17
18    range<1> get_group_range() const;
19
20    range<1> get_max_local_range() const;
21
22    uint32_t get_group_linear_id() const;
23
24    uint32_t get_local_linear_id() const;
25
26    uint32_t get_group_linear_range() const;
27
28    uint32_t get_local_linear_range() const;
29
30    bool leader() const;
31  };
32 } // namespace sycl
```

Table 100. Member functions for the `sub_group` class

Member function	Description
<code>id&lt;1&gt; get_group_id() const</code>	Return an <code>id</code> representing the index of the sub-group within the <code>work-group</code> . Since the work-items that compose a sub-group are chosen in an implementation defined way, the returned sub-group id cannot be used to identify a particular work-item in the global nd-range. Rather, the returned sub-group id is merely an abstract identifier of the sub-group containing this work-item.
<code>id&lt;1&gt; get_local_id() const</code>	Return a SYCL <code>id</code> representing the calling work-item's position within the <code>sub-group</code> .
<code>range&lt;1&gt; get_local_range() const</code>	Return a <code>range</code> representing the size of the <code>sub-group</code> . This size may be less than the value returned by <code>get_max_local_range()</code> , depending on the position of the sub-group within its parent <code>work-group</code> and the manner in which sub-groups are constructed by the implementation.
<code>range&lt;1&gt; get_group_range() const</code>	Return a <code>range</code> representing the number of <code>sub-groups</code> in the <code>work-group</code> .
<code>range&lt;1&gt; get_max_local_range() const</code>	Return a <code>range</code> representing the maximum number of work-items permitted in a <code>sub-group</code> for the executing kernel. This value may have been chosen by the programmer via an attribute, or chosen by the <code>device compiler</code> .
<code>uint32_t get_group_linear_id() const</code>	Return the same value as <code>get_group_id()[0]</code> .
<code>uint32_t get_group_linear_range() const</code>	Return the same value as <code>get_group_range()[0]</code> .
<code>uint32_t get_local_linear_id() const</code>	Return the same value as <code>get_local_id()[0]</code> .
<code>uint32_t get_local_linear_range() const</code>	Return the same value as <code>get_local_range()[0]</code> .

Member function	Description
<pre>bool leader() const</pre>	<p>Return true for exactly one work-item in the <a href="#">sub-group</a>, if the calling work-item is the leader of the sub-group, and false for all other work-items in the sub-group.</p> <p>The leader of the sub-group is determined during construction of the sub-group, and is invariant for the lifetime of the sub-group. The leader of the sub-group is guaranteed to be the work-item with a local id of 0.</p>

## 4.9.2. Reduction variables

All functionality related to [reductions](#) is captured by the [reducer](#) class and the [reduction](#) function.

The example below demonstrates how to write a [reduction](#) kernel that performs two reductions simultaneously on the same input values, computing both the sum of all values in a buffer and the maximum value in the buffer. For each reduction variable passed to [parallel\\_for](#), a reference to a [reducer](#) object is passed as a parameter to the kernel function in the same order.

```

1  buffer<int> valuesBuf{1024};
2  {
3      // Initialize buffer on the host with 0, 1, 2, 3, ..., 1023
4      host\_accessor a{valuesBuf};
5      std::iota(a.begin(), a.end(), 0);
6  }
7
8  // Buffers with just 1 element to get the reduction results
9  int sumResult = 0;
10 buffer<int> sumBuf{&sumResult, 1};
11 int maxResult = 0;
12 buffer<int> maxBuf{&maxResult, 1};
13
14 myQueue.submit([&](handler& cgh) {
15     // Input values to reductions are standard accessors
16     auto inputValues = valuesBuf.get\_access<access\_mode::read>(cgh);
17
18     // Create temporary objects describing variables with reduction semantics
19     auto sumReduction = reduction(sumBuf, cgh, plus<>());
20     auto maxReduction = reduction(maxBuf, cgh, maximum<>());
21
22     // parallel_for performs two reduction operations
23     // For each reduction variable, the implementation:
24     // - Creates a corresponding reducer
25     // - Passes a reference to the reducer to the lambda as a parameter
26     cgh.parallel\_for(range<1>{1024}, sumReduction, maxReduction,
27                     [=](id<1> idx, auto& sum, auto& max) {
28                         // plus<>() corresponds to += operator, so sum can be
29                         // updated via += or combine()
30                         sum += inputValues[idx];
31

```

```

32          // maximum<>() has no shorthand operator, so max can only
33          // be updated via combine()
34          max.combine(inputValues[idx]);
35      });
36  });
37
38  // sumBuf and maxBuf contain the reduction results once the kernel completes
39  assert(maxBuf.get_host_access()[0] == 1023 &&
40         sumBuf.get_host_access()[0] == 523776);

```

Reductions are supported for all trivially copyable types (as defined by the C++ core language). If the reduction operator is non-associative or non-commutative, the behavior of a reduction may be non-deterministic. If multiple reductions reference the same reduction variable, or a reduction variable is accessed directly during the lifetime of a reduction (e.g. via an [accessor](#) or USM pointer), the behavior is undefined.

Some of the overloads for the [reduction](#) function take an identity value and some do not. An implementation is required to compute a correct reduction even when the application does not specify an identity value. However, the implementation may be more efficient when the identity value is either provided by the application or is known by the implementation. For reductions using standard binary operators and fundamental types (e.g. [plus](#) and arithmetic types), an implementation can determine the correct identity value automatically in order to avoid performance penalties.

If an implementation can identify an identity value for a given combination of accumulator type and function object type, the value is defined as a member of the [known\\_identity](#) trait class and can be retrieved using the [known\\_identity\\_v](#) variable template. Whether this member value exists can be tested using the [has\\_known\\_identity](#) trait class or the [has\\_known\\_identity\\_v](#) variable template.

```

1  template <typename BinaryOperation, typename AccumulatorT>
2  struct known_identity {
3      static constexpr AccumulatorT value;
4  };
5
6  template <typename BinaryOperation, typename AccumulatorT>
7  inline constexpr AccumulatorT known_identity_v =
8      known_identity<BinaryOperation, AccumulatorT>::value;
9
10 template <typename BinaryOperation, typename AccumulatorT>
11 struct has_known_identity {
12     static constexpr bool value;
13 };
14
15 template <typename BinaryOperation, typename AccumulatorT>
16 inline constexpr bool has_known_identity_v =
17     has_known_identity<BinaryOperation, AccumulatorT>::value;

```

For each of the partial specializations listed in [Table 101](#), [known\\_identity](#) exists and has the value shown.

*Table 101. Known identities.*

Operator	Available Only When	Identity
<code>sycl::plus</code>	<code>std::is_arithmetic_v&lt;AccumulatorT&gt;    std::is_same_v&lt;std::remove_cv_t&lt;AccumulatorT&gt;, sycl::half&gt;</code>	<code>AccumulatorT{}</code>
<code>sycl::multiplies</code>	<code>std::is_arithmetic_v&lt;AccumulatorT&gt;    std::is_same_v&lt;std::remove_cv_t&lt;AccumulatorT&gt;, sycl::half&gt;</code>	<code>AccumulatorT{1}</code>
<code>sycl::bit_and</code>	<code>std::is_integral_v&lt;AccumulatorT&gt;</code>	<code>~AccumulatorT{}</code>
<code>sycl::bit_or</code>	<code>std::is_integral_v&lt;AccumulatorT&gt;</code>	<code>AccumulatorT{}</code>
<code>sycl::bit_xor</code>	<code>std::is_integral_v&lt;AccumulatorT&gt;</code>	<code>AccumulatorT{}</code>
<code>sycl::logical_and</code>	<code>std::is_same_v&lt;std::remove_cv_t&lt;AccumulatorT&gt;, bool&gt;</code>	<code>true</code>
<code>sycl::logical_or</code>	<code>std::is_same_v&lt;std::remove_cv_t&lt;AccumulatorT&gt;, bool&gt;</code>	<code>false</code>
<code>sycl::minimum</code>	<code>std::is_integral_v&lt;AccumulatorT&gt;</code>	<code>std::numeric_limits&lt;AccumulatorT&gt;::max()</code>
<code>sycl::minimum</code>	<code>std::is_floating_point_v&lt;AccumulatorT&gt;    std::is_same_v&lt;std::remove_cv_t&lt;AccumulatorT&gt;, sycl::half&gt;</code>	<code>std::numeric_limits&lt;AccumulatorT&gt;::infinity()</code>
<code>sycl::maximum</code>	<code>std::is_integral_v&lt;AccumulatorT&gt;</code>	<code>std::numeric_limits&lt;AccumulatorT&gt;::lowest()</code>
<code>sycl::maximum</code>	<code>std::is_floating_point_v&lt;AccumulatorT&gt;    std::is_same_v&lt;std::remove_cv_t&lt;AccumulatorT&gt;, sycl::half&gt;</code>	<code>-std::numeric_limits&lt;AccumulatorT&gt;::infinity()</code>

The reduction interface is limited to reduction variables whose size can be determined at compile-time.

As such, `buffer` and USM pointer arguments are interpreted by the reduction interface as describing a single variable. A reduction operation associated with a `span` represents an array reduction. An array reduction of size  $N$  is functionally equivalent to specifying  $N$  independent scalar reductions. The combination operations performed by an array reduction are limited to the extent of a USM allocation described by a `span`, and access to elements outside of these regions results in undefined behavior.



Since a `span` is one-dimensional, there is currently no way to describe an array reduction with more than one dimension. This is expected to change in a future version of the SYCL specification, but depends on the introduction of a multi-dimensional `span`.

#### 4.9.2.1. `reduction` interface

The `reduction` interface is used to attach `reduction` semantics to a variable, by specifying: the reduction variable, the reduction operator and an optional identity value associated with the operator. The overloads of the interface are described in Table 102. The return value of the `reduction` interface is an implementation-defined object of unspecified type, which is interpreted by `parallel_for` to construct an appropriate `reducer` type as detailed in Section 4.9.2.3.

An implementation may use an unspecified number of temporary variables inside of any `reducer` objects it creates. If an identity value is supplied to a reduction, an implementation will use that value to initialize any such temporary variables.



Since the number of temporary variables is unspecified, supplying an identity value different to the identity value associated with the reduction operator may lead to unexpected results.

The initial value of the reduction variable is included in the reduction operation, unless the `property::reduction::initialize_to_identity` property was specified when the `reduction` interface was invoked.

The reduction variable is updated so as to contain the result of the reduction when the kernel finishes execution.

```

1 template <typename BufferT, typename BinaryOperation>
2 __unspecified__ reduction(BufferT vars, handler& cgh, BinaryOperation combiner,
3                             const property_list& propList = {});
4
5 template <typename T, typename BinaryOperation>
6 __unspecified__ reduction(T* var, BinaryOperation combiner,
7                             const property_list& propList = {});
8
9 template <typename T, typename Extent, typename BinaryOperation>
10 __unspecified__ reduction(span<T, Extent> vars, BinaryOperation combiner,
11                             const property_list& propList = {});
12
13 template <typename BufferT, typename BinaryOperation>
14 __unspecified__
15 reduction(BufferT vars, handler& cgh, const BufferT::value_type& identity,
16           BinaryOperation combiner, const property_list& propList = {});
17
18 template <typename T, typename BinaryOperation>
19 __unspecified__ reduction(T* var, const T& identity, BinaryOperation combiner,
20                             const property_list& propList = {});
21
22 template <typename T, typename Extent, typename BinaryOperation>
23 __unspecified__ reduction(span<T, Extent> vars, const T& identity,
```

```

24         BinaryOperation combiner,
25         const property_list& propList = {});

```

Table 102. Overloads of the `reduction` interface

Function	Description
<pre> reduction&lt;BufferT, BinaryOperation&gt;(BufferT vars, handler&amp; cgh,                                 BinaryOperation combiner,                                 const property_list&amp; propList = {}) </pre>	Construct an unspecified object representing a reduction of the variable(s) described by <code>vars</code> using the combination operation specified by <code>combiner</code> . Zero or more properties can be provided via an instance of <code>property_list</code> . Throws an <code>exception</code> with the <code>errc::invalid</code> error code if the range of the <code>vars</code> buffer is not 1.
<pre> reduction&lt;T, BinaryOperation&gt;(T* var, BinaryOperation combiner,                                 const property_list&amp; propList = {}) </pre>	Construct an unspecified object representing a reduction of the variable described by <code>var</code> using the combination operation specified by <code>combiner</code> . Zero or more properties can be provided via an instance of <code>property_list</code> .
<pre> reduction&lt;T, BinaryOperation&gt;(span&lt;T, Extent&gt; vars, BinaryOperation combiner,                                 const property_list&amp; propList = {}) </pre>	Available only when <code>Extent != sycl::dynamic_extent</code> . Construct an unspecified object representing a reduction of the variable(s) described by <code>vars</code> using the combination operation specified by <code>combiner</code> . Zero or more properties can be provided via an instance of <code>property_list</code> .
<pre> reduction&lt;BufferT, BinaryOperation&gt;(BufferT vars, handler&amp; cgh,                                 const BufferT ::value_type&amp; identity,                                 BinaryOperation combiner,                                 const property_list&amp; propList = {}) </pre>	Construct an unspecified object representing a reduction of the variable(s) described by <code>vars</code> using the combination operation specified by <code>combiner</code> . The value of <code>identity</code> may be used by the implementation to initialize an unspecified number of temporary accumulation variables. Zero or more properties can be provided via an instance of <code>property_list</code> . Throws an <code>exception</code> with the <code>errc::invalid</code> error code if the range of the <code>vars</code> buffer is not 1.

Function	Description
<pre>reduction&lt;T, BinaryOperation&gt;(T* var, const T&amp; identity,                                BinaryOperation combiner,                                const property_list&amp; proplist                                = {})</pre>	Construct an unspecified object representing a reduction of the variable described by <code>var</code> using the combination operation specified by <code>combiner</code> . The value of <code>identity</code> may be used by the implementation to initialize an unspecified number of temporary accumulation variables. Zero or more properties can be provided via an instance of <code>property_list</code> .
<pre>reduction&lt;T, BinaryOperation&gt;(span&lt;T, Extent&gt; vars, const                                T&amp; identity,                                BinaryOperation combiner,                                const property_list&amp; proplist                                = {})</pre>	Available only when <code>Extent != sycl::dynamic_extent</code> . Construct an unspecified object representing a reduction of the variable(s) described by <code>vars</code> using the combination operation specified by <code>combiner</code> . The value of <code>identity</code> may be used by the implementation to initialize an unspecified number of temporary accumulation variables. Zero or more properties can be provided via an instance of <code>property_list</code> .

#### 4.9.2.2. Reduction properties

The properties that can be provided when using the `reduction` interface are described in [Table 103](#).

Table 103. Properties supported by the `reduction` interface

Property	Description
<pre>property::reduction::initialize_to_identity</pre>	The <code>initialize_to_identity</code> property adds the requirement that the SYCL runtime must initialize the <code>reduction</code> variable to the identity value passed to the reduction interface, or to the identity value determined by the <code>known_identity</code> trait if no identity value was specified. If no identity value was specified and an identity value cannot be determined by the <code>known_identity</code> trait, the implementation must throw an <code>exception</code> with the <code>errc::invalid</code> error code. When this property is set, the original value of the reduction variable is not included in the reduction.

The constructors of the reduction property classes are listed in [Table 104](#).

Table 104. Constructors of the `reduction property` classes



Constructor	Description
<code>property::reduction::initialize_to_identity::initialize_to_identity()</code>	Constructs an <code>initialize_to_identity</code> property instance.

#### 4.9.2.3. `reducer` class

The `reducer` class defines the interface between a work-item and a reduction variable during the execution of a SYCL kernel, restricting access to the underlying reduction variable. The intermediate values of a reduction variable cannot be inspected during kernel execution, and the variable cannot be updated using anything other than the reduction's specified combination operation. The combination order of different reducers is unspecified, as are when and how the value of each reducer is combined with the original reduction variable.

To enable compile-time specialization of reduction algorithms, the implementation of the `reducer` class is unspecified, except for the functions and operators defined in [Table 106](#) and [Table 107](#). As such, developers should not specify the template arguments of a `reducer` directly, and should instead employ generic programming techniques that allow kernel functions to accept a reference to a variable of any `reducer` type. Kernels written as lambdas should employ `auto&` or `auto&...`, and kernels written as function objects should employ template parameters or template parameter packs.

An implementation must guarantee that it is safe for multiple work-items in a kernel to call the combine function of a `reducer` concurrently. An implementation is free to re-use reducer variables (e.g. across work-groups scheduled to the same compute unit) if it can guarantee that it is safe to do so.

The type aliases and constant static members of the `reducer` class are listed in [Table 105](#) and its member functions are listed in [Table 106](#). Additional shorthand operators may be made available for certain combinations of reduction variable type and combination operation, as described in [Table 107](#).

```

1 // Exposition only
2 template <typename T, typename BinaryOperation, int Dimensions,
3         /* unspecified */>
4 class reducer {
5 public:
6     using value_type = T;
7     using binary_operation = BinaryOperation;
8     static constexpr int dimensions = Dimensions;
9
10    reducer(const reducer&) = delete;
11    reducer(reducer&&) = delete;
12    reducer& operator=(const reducer&) = delete;
13    reducer& operator=(reducer&&) = delete;
14
15    ~reducer();
16
17    /* Only available if Dimensions == 0 */
18    reducer& combine(const T& partial);
19
20    /* Only available if Dimensions > 0 */
21    __unspecified__ operator[](size_t index)
22
23    /* Only available if identity value is known */
24    T identity() const;
25
26    /* Only available if Dimensions == 0 and either

```

```

27  * BinaryOperation == plus<> or BinaryOperation == plus<T> */
28  friend reducer& operator+=(reducer&, const T&) { /* ... */
29  }
30
31  /* Only available if Dimensions == 0 and either
32  * BinaryOperation == multiplies<> or BinaryOperation == multiplies<T> */
33  friend reducer& operator*=(reducer&, const T&) { /* ... */
34  }
35
36  /* Only available if Dimensions == 0, T is an integral type and either
37  * BinaryOperation == bit_and<> or BinaryOperation == bit_and<T> */
38  friend reducer& operator&=(reducer&, const T&) { /* ... */
39  }
40
41  /* Only available if Dimensions == 0, T is an integral type and either
42  * BinaryOperation == bit_or<> or BinaryOperation == bit_or<T> */
43  friend reducer& operator|=(reducer&, const T&) { /* ... */
44  }
45
46  /* Only available if Dimensions == 0, T is an integral type and either
47  * BinaryOperation == bit_xor<> or BinaryOperation == bit_xor<T> */
48  friend reducer& operator^=(reducer&, const T&) { /* ... */
49  }
50
51  /* Only available if Dimensions == 0, T is an integral type, T is not bool and
52  * either BinaryOperation == plus<> or BinaryOperation == plus<T> */
53  friend reducer& operator++(reducer&) { /* ... */
54  }
55 };

```

Table 105. Member types and constants of the `reducer` class

Member	Description
<code>value_type</code>	The data type of the reduction variable. If this reducer object was created from a buffer type <code>BufferT</code> , this type is <code>BufferT::value_type</code> . If this reducer object was created from a USM pointer <code>T*</code> or a span <code>span&lt;T, Extent&gt;</code> , this type is <code>T</code> .
<code>binary_operation</code>	The type of the combiner operator <code>BinaryOperation</code> that was passed to the reduction function that created this reducer object.
<code>static constexpr int dimensions</code>	The number of dimensions of the reduction variable. If this reducer object was created from a buffer or a USM pointer, the number of dimensions is <code>0</code> . If this reducer object was created from a span, the number of dimensions is <code>1</code> .

Table 106. Member functions of the `reducer` class

Member function	Description
<code>reducer&amp; combine(const T&amp; partial)</code>	Available only when: <code>Dimensions == 0</code> . Combine the value of <code>partial</code> with the reduction variable associated with this <code>reducer</code> . Returns <code>*this</code> .
<code>__unspecified__ operator[](size_t index)</code>	Available only when: <code>Dimensions &gt; 0</code> . Returns an instance of an undefined intermediate type representing a <code>reducer</code> of the same type as this <code>reducer</code> , with the dimensionality <code>Dimensions-1</code> and containing an implicit SYCL <code>id</code> with index <code>Dimensions</code> set to <code>index</code> . The intermediate type returned must provide all member functions and operators defined by the <code>reducer</code> class that are appropriate for the type it represents (including this subscript operator).
<code>T identity() const</code>	Return the identity value of the combination operation associated with this <code>reducer</code> . Only available if the identity value is known to the implementation.

Table 107. Hidden friend operators of the `reducer` class

Operator	Description
<code>reducer&amp; operator+=(reducer&amp; accum, const T&amp; partial)</code>	Equivalent to calling <code>accum.combine(partial)</code> . Available only when: <code>Dimensions == 0 &amp;&amp; (std::is_same_v&lt;BinaryOperation, plus&lt;&gt;&gt;    std::is_same_v&lt;BinaryOperation, plus&lt;T&gt;&gt;)</code> .
<code>reducer&amp; operator*=(reducer&amp; accum, const T&amp; partial)</code>	Equivalent to calling <code>accum.combine(partial)</code> . Available only when: <code>Dimensions == 0 &amp;&amp; (std::is_same_v&lt;BinaryOperation, multiplies&lt;&gt;&gt;    std::is_same_v&lt;BinaryOperation, multiplies&lt;T&gt;&gt;)</code> .
<code>reducer&amp; operator&amp;=(reducer&amp; accum, const T&amp; partial)</code>	Equivalent to calling <code>accum.combine(partial)</code> . Available only when: <code>Dimensions == 0 &amp;&amp; is_integral_v&lt;T&gt; &amp;&amp; (std::is_same_v&lt;BinaryOperation, bit_and&lt;&gt;&gt;    std::is_same_v&lt;BinaryOperation, bit_and&lt;T&gt;&gt;)</code> .

Operator	Description
<code>reducer&amp; operator =(reducer&amp; accum, const T&amp; partial)</code>	Equivalent to calling <code>accum.combine(partial)</code> . Available only when: <code>Dimensions == 0 &amp;&amp; is_integral_v&lt;T&gt; &amp;&amp; (std::is_same_v&lt;BinaryOperation, bit_or&lt;&gt;&gt;    std::is_same_v&lt;BinaryOperation, bit_or&lt;T&gt;&gt;)</code> .
<code>reducer&amp; operator^=(reducer&amp; accum, const T&amp; partial)</code>	Equivalent to calling <code>accum.combine(partial)</code> . Available only when: <code>Dimensions == 0 &amp;&amp; is_integral_v&lt;T&gt; &amp;&amp; (std::is_same_v&lt;BinaryOperation, bit_xor&lt;&gt;&gt;    std::is_same_v&lt;BinaryOperation, bit_xor&lt;T&gt;&gt;)</code> .
<code>reducer&amp; operator++(reducer&amp; accum)</code>	Equivalent to calling <code>accum.combine(1)</code> . Available only when: <code>Dimensions == 0 &amp;&amp; std::is_integral_v&lt;T&gt; &amp;&amp; !std::is_same_v&lt;T, bool&gt; &amp;&amp; (std::is_same_v&lt;BinaryOperation, plus&lt;&gt;&gt;    std::is_same_v&lt;BinaryOperation, plus&lt;T&gt;&gt;)</code> .

### 4.9.3. Command group scope

A [command group scope](#), as defined in [Section 3.7.1](#), may execute a single [command](#) such as invoking a kernel, copying memory, or executing a host task. It is legal for a [command group scope](#) to statically contain more than one call to a [command](#) function, but any single execution of the [command group function object](#) may execute no more than one [command](#). If an application fails to do this, the function that submits the [command group function object](#) (i.e., `queue::submit`) must throw a synchronous [exception](#) with the `errc::invalid` error code. The statements that call [commands](#) together with the statements that define the requirements for a kernel form the [command group function object](#). The command group function object takes as a parameter an instance of the [command group handler](#) class which encapsulates all the member functions executed in the command group scope. The member functions and objects defined in this scope will define the requirements for the kernel execution or explicit memory operation, and will be used by the [SYCL runtime](#) to evaluate if the operation is ready for execution. Host code within a [command group function object](#) (typically setting up requirements) is executed once, before the command group submit call returns. This abstraction of the kernel execution unifies the data with its processing, and consequently allows more abstraction and flexibility in the parallel programming models that can be implemented on top of SYCL.

The [command group function object](#) and the [handler](#) class serve as an interface for the encapsulation of [command group scope](#). A [SYCL kernel function](#) is defined as a function object. All the device data accesses are defined inside this group and any transfers are managed by the [SYCL runtime](#). The rules for the data transfers regarding device and host data accesses are better described in [Section 4.7](#), where buffers ([Section 4.7.2](#)) and accessor ([Section 4.7.6](#)) classes are described. The overall memory model of the SYCL application is described in [Section 3.8.1](#).

It is possible for a [command group function object](#) to fail to enqueue to a queue, or for it to fail to execute correctly. A user can therefore supply a secondary queue when submitting a command group to the primary queue. If the [SYCL runtime](#) fails to enqueue or execute a command group on a primary queue, it can attempt to run the command group on the secondary queue. The circumstances in which it is, or is not, possible for a [SYCL runtime](#) to fall-back from primary to secondary queue are unspecified in the specification. Even if a command group is run on the secondary queue, the requirement that host code

within the command group is executed exactly once remains, regardless of whether the fallback queue is used for execution.

The command group **handler** class provides the interface for all of the member functions that are able to be executed inside the command group scope, and it is also provided as a scoped object to all of the data access requests. The **command group handler** class provides the interface in which every command in the command group scope will be submitted to a queue.

#### 4.9.4. Command group **handler** class

A **command group handler** object can only be constructed by the SYCL runtime. All of the accessors defined in **command group scope** take as a parameter an instance of the **command group handler**, and all the kernel invocation functions are member functions of this class.

The constructors of the SYCL **handler** class are described in [Table 108](#).

It is disallowed for an instance of the SYCL **handler** class to be moved or copied.

```

1 namespace sycl {
2
3 class handler {
4 private:
5     // implementation defined constructor
6     handler(____unspecified____);
7
8 public:
9     template <typename DataT, int Dimensions, access_mode AccessMode,
10             target AccessTarget, access::placeholder IsPlaceholder>
11     void require(
12         accessor<DataT, Dimensions, AccessMode, AccessTarget, IsPlaceholder> acc);
13
14     void depends_on(event depEvent);
15
16     void depends_on(const std::vector<event>& depEvents);
17
18     //----- Backend interoperability interface
19     //
20     template <typename T> void set_arg(int argIndex, T&& arg);
21
22     template <typename... Ts> void set_args(Ts&&... args);
23
24     //----- Kernel dispatch API
25     //
26     // Note: In all kernel dispatch functions, the template parameter
27     // "typename KernelName" is optional.
28     //
29     template <typename KernelName, typename KernelType>
30     void single_task(const KernelType& kernelFunc);
31
32     // Parameter pack acts as-if: Reductions&&... reductions, const KernelType
33     // &kernelFunc
34     template <typename KernelName, int Dimensions, typename... Rest>
35     void parallel_for(range<Dimensions> numWorkItems, Rest&&... rest);
36
37     // Deprecated in SYCL 2020.
38     template <typename KernelName, typename KernelType, int Dimensions>

```

```

39 void parallel_for(range<Dimensions> numWorkItems,
40                  id<Dimensions> workItemOffset,
41                  const KernelType& kernelFunc);
42
43 // Parameter pack acts as-if: Reductions&&... reductions, const KernelType
44 // &kernelFunc
45 template <typename KernelName, int Dimensions, typename... Rest>
46 void parallel_for(nd_range<Dimensions> executionRange, Rest&&... rest);
47
48 // Deprecated in SYCL 2020.
49 template <typename KernelName, typename WorkgroupFunctionType, int Dimensions>
50 void parallel_for_work_group(range<Dimensions> numWorkGroups,
51                             const WorkgroupFunctionType& kernelFunc);
52
53 // Deprecated in SYCL 2020.
54 template <typename KernelName, typename WorkgroupFunctionType, int Dimensions>
55 void parallel_for_work_group(range<Dimensions> numWorkGroups,
56                             range<Dimensions> workGroupSize,
57                             const WorkgroupFunctionType& kernelFunc);
58
59 void single_task(const kernel& kernelObject);
60
61 template <int Dimensions>
62 void parallel_for(range<Dimensions> numWorkItems, const kernel& kernelObject);
63
64 template <int Dimensions>
65 void parallel_for(nd_range<Dimensions> ndRange, const kernel& kernelObject);
66
67 //----- USM functions
68 //
69
70 void memcpy(void* dest, const void* src, size_t numBytes);
71
72 template <typename T> void copy(const T* src, T* dest, size_t count);
73
74 void memset(void* ptr, int value, size_t numBytes);
75
76 template <typename T> void fill(void* ptr, const T& pattern, size_t count);
77
78 void prefetch(void* ptr, size_t numBytes);
79
80 void mem_advise(void* ptr, size_t numBytes, int advice);
81
82 //----- Explicit memory operation APIs
83 //
84 template <typename SrcT, int SrcDim, access_mode SrcMode, target SrcTgt,
85          access::placeholder IsPlaceholder, typename DestT>
86 void copy(accessor<SrcT, SrcDim, SrcMode, SrcTgt, IsPlaceholder> src,
87          std::shared_ptr<DestT> dest);
88
89 template <typename SrcT, typename DestT, int DestDim, access_mode DestMode,
90          target DestTgt, access::placeholder IsPlaceholder>
91 void copy(std::shared_ptr<SrcT> src,
92          accessor<DestT, DestDim, DestMode, DestTgt, IsPlaceholder> dest);
93
94 template <typename SrcT, int SrcDim, access_mode SrcMode, target SrcTgt,

```

```

95         access::placeholder IsPlaceholder, typename DestT>
96 void copy(accessor<SrcT, SrcDim, SrcMode, SrcTgt, IsPlaceholder> src,
97           DestT* dest);
98
99 template <typename SrcT, typename DestT, int DestDim, access_mode DestMode,
100           target DestTgt, access::placeholder IsPlaceholder>
101 void copy(const SrcT* src,
102           accessor<DestT, DestDim, DestMode, DestTgt, IsPlaceholder> dest);
103
104 template <typename SrcT, int SrcDim, access_mode SrcMode, target SrcTgt,
105           access::placeholder SrcIsPlaceholder, typename DestT, int DestDim,
106           access_mode DestMode, target DestTgt,
107           access::placeholder DestIsPlaceholder>
108 void
109 copy(accessor<SrcT, SrcDim, SrcMode, SrcTgt, SrcIsPlaceholder> src,
110       accessor<DestT, DestDim, DestMode, DestTgt, DestIsPlaceholder> dest);
111
112 template <typename T, int Dim, access_mode Mode, target Tgt,
113           access::placeholder IsPlaceholder>
114 void update_host(accessor<T, Dim, Mode, Tgt, IsPlaceholder> acc);
115
116 template <typename T, int Dim, access_mode Mode, target Tgt,
117           access::placeholder IsPlaceholder>
118 void fill(accessor<T, Dim, Mode, Tgt, IsPlaceholder> dest, const T& src);
119
120 void
121 use_kernel_bundle(const kernel_bundle<bundle_state::executable>& execBundle);
122
123 template <auto& SpecName>
124 void set_specialization_constant(
125     typename std::remove_reference_t<decltype(SpecName)>::value_type value);
126
127 template <auto& SpecName>
128 typename std::remove_reference_t<decltype(SpecName)>::value_type
129 get_specialization_constant();
130 };
131 } // namespace sycl

```

Table 108. Constructors of the `handler` class

Constructor	Description
<code>handler(____unspecified____)</code>	Unspecified implementation-defined constructor.

#### 4.9.4.1. SYCL functions for adding requirements

When an accessor is created from a [command group handler](#), a **requirement** is implicitly added to the [command group](#) for the accessor's data. However, this does not happen when creating a placeholder accessor. In order to create a **requirement** for a placeholder accessor, code must call the `handler::require()` member function.

Note that the default constructed `accessor` is not a placeholder, so it may be passed to a [SYCL kernel function](#) without calling `handler::require()`. However, this accessor also has no underlying memory object, so such an accessor does not create any **requirement** for the command group, and attempting to access data elements from it produces undefined behavior.



SYCL events may also be used to create requirements for a [command group](#). Such requirements state that the actions represented by the events must complete before the [command group](#) may execute. Such requirements are added when code calls the `handler::depends_on()` member function.

Table 109. Member functions of the `handler` class

Member function	Description
<pre>template &lt;typename DataT, int Dimensions, access_mode AccessMode,         target AccessTarget, access::placeholder IsPlaceholder&gt; void require(     accessor&lt;DataT, Dimensions, AccessMode, AccessTarget, IsPlaceholder&gt; acc)</pre>	Calling this function has no effect unless <code>acc</code> is a placeholder accessor. When <code>acc</code> is a placeholder accessor, this function adds a <b>requirement</b> to the handler's <a href="#">command group</a> for the memory object represented by <code>acc</code> . If the accessor has already been registered with the <a href="#">command group</a> , calling this function has no effect.
<pre>void depends_on(event depEvent)</pre>	The <a href="#">command group</a> now has a <b>requirement</b> that the action represented by <code>depEvent</code> must complete before executing this <a href="#">command-group's</a> action.
<pre>void depends_on(const std::vector&lt;event&gt;&amp; depEvents)</pre>	The <a href="#">command group</a> now has a <b>requirement</b> that the actions represented by each event in <code>depEvents</code> must complete before executing this <a href="#">command-group's</a> action.

#### 4.9.4.2. SYCL functions for invoking kernels

[Kernels](#) can be invoked as single tasks, basic data-parallel [kernels](#), [nd-range](#) in [work-groups](#), or hierarchical parallelism.

Each function takes an optional kernel name template parameter. The user may optionally provide a [kernel name](#), otherwise an implementation-defined name will be generated for the kernel.

All the functions for invoking kernels are member functions of the command group `handler` class ([Section 4.9.4](#)), which is used to encapsulate all the member functions provided in a command group scope. [Table 110](#) lists all the members of the `handler` class related to the kernel invocation.

Table 110. Member functions of the `handler` class

Member function	Description
<pre>template &lt;typename T&gt; void set_arg(int argIndex, T&amp;&amp; arg)</pre>	This function must only be used to set arguments for a kernel that was constructed using a backend specific interoperability function or for a device built-in kernel. Attempting to use this function to set arguments for other kernels results in undefined behavior. The precise semantics of this function are defined by each SYCL backend specification.



Member function	Description
<pre>template &lt;typename... Ts&gt; void set_args(Ts&amp;&amp;... args)</pre>	Set all arguments for a given kernel, as if each argument in <code>args</code> was passed to <code>set_arg</code> in the same order and with an increasing index starting at 0.
<pre>template &lt;typename KernelName, typename KernelType&gt; void single_task(const KernelType&amp; kernelFunc)</pre>	Defines and invokes a <a href="#">SYCL kernel function</a> as a lambda expression or a named function object type. Specification of a <a href="#">kernel name</a> ( <code>typename KernelName</code> ), as described in <a href="#">Section 4.9.4.2</a> , is optional. The callable <code>KernelType</code> can optionally take a <code>kernel_handler</code> in which case the <a href="#">SYCL runtime</a> will construct an instance of <code>kernel_handler</code> and pass it to <code>KernelType</code> .
<pre>template &lt;typename KernelName, int Dimensions, typename... Rest&gt; void parallel_for(range&lt;Dimensions&gt; numWorkItems, Rest&amp;&amp;... rest)</pre>	Defines and invokes a <a href="#">SYCL kernel function</a> as a lambda expression or a named function object type, for the specified range and given an item or integral type (e.g <code>int</code> , <code>size_t</code> ), if range is 1-dimensional, for indexing in the indexing space defined by range. Generic kernel functions are permitted, in that case the argument type is an <code>item</code> . Specification of a <a href="#">kernel name</a> ( <code>typename KernelName</code> ), as described in <a href="#">Section 4.9.4.2</a> , is optional. The <code>rest</code> parameter pack consists of 0 or more objects created by the <code>reduction</code> function, followed by a callable. For each object in <code>rest</code> , the kernel function must take an additional reference parameter corresponding to that object's <code>reducer</code> type, in the same order. The callable can optionally take a <code>kernel_handler</code> as its last parameter, in which case the <a href="#">SYCL runtime</a> will construct an instance of <code>kernel_handler</code> and pass it to the callable.

Member function	Description
<pre> template &lt;typename KernelName, int Dimensions, typename... Rest&gt; void parallel_for(range&lt;Dimensions&gt; numWorkItems, id &lt;Dimensions&gt; workItemOffset,                 const KernelType&amp; kernelFunc)     // Deprecated in SYCL 2020. </pre>	<p>Deprecated in SYCL 2020. Defines and invokes a <a href="#">SYCL kernel function</a> as a lambda expression or a named function object type, for the specified range and offset and given an item or integral type (e.g <code>int</code>, <code>size_t</code>), if range is 1-dimensional, for indexing in the indexing space defined by range. Generic kernel functions are permitted, in that case the argument type is an <code>item</code>. Specification of a <a href="#">kernel name</a> (<code>typename KernelName</code>), as described in <a href="#">Section 4.9.4.2</a>, is optional. The <code>rest</code> parameter pack consists of 0 or more objects created by the <code>reduction</code> function, followed by a callable. For each object in <code>rest</code>, the kernel function must take an additional reference parameter corresponding to that object's <code>reducer</code> type, in the same order. The callable can optionally take a <code>kernel_handler</code> as its last parameter, in which case the <a href="#">SYCL runtime</a> will construct an instance of <code>kernel_handler</code> and pass it to the callable.</p>

Member function	Description
<pre> template &lt;typename KernelName, int Dimensions, typename... Rest&gt; void parallel_for(nd_range&lt;Dimensions&gt; executionRange, Rest&amp;&amp;... rest) </pre>	<p>Defines and invokes a <a href="#">SYCL kernel function</a> as a lambda expression or a named function object type, for the specified <a href="#">nd-range</a> and given an <a href="#">nd-item</a> for indexing in the indexing space defined by the <a href="#">nd-range</a>. Generic kernel functions are permitted, in that case the argument type is an <a href="#">nd-item</a>. Specification of a <a href="#">kernel name</a> (<a href="#">typename KernelName</a>), as described in <a href="#">Section 4.9.4.2</a>, is optional. The <a href="#">rest</a> parameter pack consists of 0 or more objects created by the <a href="#">reduction</a> function, followed by a callable. For each object in <a href="#">rest</a>, the kernel function must take an additional reference parameter corresponding to that object's <a href="#">reducer</a> type, in the same order. The callable can optionally take a <a href="#">kernel_handler</a> as its last parameter, in which case the <a href="#">SYCL runtime</a> will construct an instance of <a href="#">kernel_handler</a> and pass it to the callable.</p> <p>Throws an <a href="#">exception</a> with the <a href="#">errc::nd_range</a> error code if the global size defined in the associated <a href="#">executionRange</a> defines a non-zero index space which is not evenly divisible by the local size in each dimension.</p>

Member function	Description
<pre> template &lt;typename KernelName, typename WorkgroupFunctionType, int Dimensions&gt; void parallel_for_work_group(range&lt;Dimensions&gt; numWorkGroups,                                 const WorkgroupFunctionType&amp; kernelFunc) </pre>	<p>Deprecated in SYCL 2020. Defines and invokes a hierarchical kernel as a lambda expression or a named function object type, encoding the body of each work-group to launch. Generic kernel functions are permitted, in that case the argument type is a <code>group</code>. May contain multiple calls to <code>parallel_for_work_item(..)</code> member functions representing the execution on each work-item. Launches <code>num_work_groups</code> work-groups of runtime-defined size. Described in detail in <a href="#">Section 4.9.4.2</a>. The callable <code>WorkgroupFunctionType</code> can optionally take a <code>kernel_handler</code> as its last parameter, in which case the <a href="#">SYCL runtime</a> will construct an instance of <code>kernel_handler</code> and pass it to <code>WorkgroupFunctionType</code>.</p>
<pre> template &lt;typename KernelName, typename WorkgroupFunctionType, int Dimensions&gt; void parallel_for_work_group(range&lt;Dimensions&gt; numWorkGroups,                                 range&lt;Dimensions&gt; workGroupSize,                                 const WorkgroupFunctionType&amp; kernelFunc) </pre>	<p>Deprecated in SYCL 2020. Defines and invokes a hierarchical kernel as a lambda expression or a named function object type, encoding the body of each work-group to launch. Generic kernel functions are permitted, in that case the argument type is a <code>group</code>. May contain multiple calls to <code>parallel_for_work_item</code> member functions representing the execution on each work-item. Launches <code>num_work_groups</code> work-groups of <code>work_group_size</code> work-items each. Described in detail in <a href="#">Section 4.9.4.2</a>. The callable <code>WorkgroupFunctionType</code> can optionally take a <code>kernel_handler</code> as its last parameter, in which case the <a href="#">SYCL runtime</a> will construct an instance of <code>kernel_handler</code> and pass it to <code>WorkgroupFunctionType</code>.</p>

Member function	Description
<pre data-bbox="161 185 751 219">void single_task(const kernel&amp; kernelObject)</pre>	<p data-bbox="1003 159 1458 658">This function must only be used to invoke a kernel that was constructed using a backend specific interoperability function or to invoke a device built-in kernel. Attempting to use this function to invoke other kernels throws a synchronous <code>exception</code> with the <code>errc::invalid</code> error code. The precise semantics of this function are defined by each SYCL backend specification, but the intent is that the kernel should execute exactly once.</p> <p data-bbox="1003 692 1458 1196">This invocation function ignores any <code>kernel_bundle</code> that was bound to this command group handler via <code>handler::use_kernel_bundle()</code> and instead implicitly uses the kernel bundle that contains the <code>kernelObject</code>. Throws an <code>exception</code> with the <code>errc::kernel_not_supported</code> error code if the <code>kernelObject</code> is not compatible with either the device associated with the primary queue of the <code>command group</code> or with the device associated with the secondary queue (if specified).</p>

Member function	Description
<pre> template &lt;int Dimensions&gt; void parallel_for(range&lt;Dimensions&gt; numWorkItems, const kernel&amp; kernelObject) </pre>	<p>This function must only be used to invoke a kernel that was constructed using a backend specific interoperability function or to invoke a device built-in kernel. Attempting to use this function to invoke other kernels throws a synchronous <code>exception</code> with the <code>errc::invalid</code> error code. The precise semantics of this function are defined by each SYCL backend specification, but the intent is that the kernel should be invoked for the specified range of index values.</p> <p>This invocation function ignores any <code>kernel_bundle</code> that was bound to this command group handler via <code>handler::use_kernel_bundle()</code> and instead implicitly uses the kernel bundle that contains the <code>kernelObject</code>. Throws an <code>exception</code> with the <code>errc::kernel_not_supported</code> error code if the <code>kernelObject</code> is not compatible with either the device associated with the primary queue of the <code>command group</code> or with the device associated with the secondary queue (if specified).</p>

Member function	Description
<pre>template &lt;int Dimensions&gt; void parallel_for(nd_range&lt;Dimensions&gt; executionRange,                   const kernel&amp; kernelObject)</pre>	<p>This function must only be used to invoke a kernel that was constructed using a backend specific interoperability function or to invoke a device built-in kernel. Attempting to use this function to invoke other kernels throws a synchronous <code>exception</code> with the <code>errc::invalid</code> error code. The precise semantics of this function are defined by each SYCL backend specification, but the intent is that the kernel should be invoked for the specified <code>executionRange</code>.</p> <p>Throws an <code>exception</code> with the <code>errc::nd_range</code> error code if the global size defined in the associated <code>executionRange</code> defines a non-zero index space which is not evenly divisible by the local size in each dimension.</p> <p>This invocation function ignores any <code>kernel_bundle</code> that was bound to this command group handler via <code>handler::use_kernel_bundle()</code> and instead implicitly uses the kernel bundle that contains the <code>kernelObject</code>. Throws an <code>exception</code> with the <code>errc::kernel_not_supported</code> error code if the <code>kernelObject</code> is not compatible with either the device associated with the primary queue of the <code>command group</code> or with the device associated with the secondary queue (if specified).</p>

#### 4.9.4.2.1. `single_task` invoke

SYCL provides a simple interface to enqueue a kernel that will be sequentially executed on a device. Only one instance of the kernel will be executed. This interface is useful as a primitive for more complicated parallel algorithms, as it can easily create a chain of sequential tasks on a SYCL device with each of them managing its own data transfers.

This function can only be called inside a command group using the `handler` object created by the runtime. Any accessors that are used in a kernel should be defined inside the same command group.

Local accessors are disallowed for single task invocations.

```
1 myQueue.submit([&](handler& cgh) {
2   cgh.single_task(
3     [=] () {
4       // [kernel code]
```

```
5     }));
6  });
```

For single tasks, the kernel member function takes no parameters, as there is no need for [index space classes](#) in a unary index space.

A `kernel_handler` can optionally be passed as a parameter to the [SYCL kernel function](#) that is invoked by `single_task` for the purpose explained in [Section 4.9.5.3](#).

```
1 myQueue.submit([&](handler& cgh) {
2   cgh.single_task(
3     [=] (kernel_handler kh) {
4       // [kernel code]
5     });
6 });
```

#### 4.9.4.2.2. `parallel_for` invoke

The `parallel_for` member function of the SYCL `handler` class provides an interface to define and invoke a SYCL kernel function in a command group, to execute in parallel execution over a 3 dimensional index space. There are three overloads of the `parallel_for` member function which provide variations of this interface, each with a different level of complexity and providing a different set of features.

For the simplest case, users need only provide the global range (the total number of work-items in the index space) via a SYCL `range` parameter. In this case the function object that represents the SYCL kernel function must take one of: 1) a single SYCL `item` parameter, 2) a single generic parameter (`template` parameter or `auto`) that will be treated as an `item` parameter, 3) any other type implicitly converted from SYCL `item`, representing the currently executing work-item within the range specified by the `range` parameter.



Case 3) above allows the kernel function to take an argument of type `id` because `item` is implicitly convertible to `id`. It also allows a 1-D kernel function to take an integral argument (e.g. `int` or `size_t`) because a 1-D `item` is implicitly convertible to these types. Finally, it allows the kernel function to take a user-defined argument type that can be constructed from `item`, enabling users to layer their own abstractions on top of SYCL.

The execution of the kernel function is the same whether the parameter to the SYCL kernel function is a SYCL `id` or a SYCL `item`. What differs is the functionality that is available to the SYCL kernel function via the respective interfaces.

Below is an example of invoking a SYCL kernel function with `parallel_for` using a lambda expression, and passing a SYCL `id` parameter. In this case, only the global id is available. This variant of `parallel_for` is designed for when it is not necessary to query the global range of the index space being executed across.

```
1 myQueue.submit([&](handler& cgh) {
2   accessor acc{myBuffer, cgh, write_only};
3
4   cgh.parallel_for(range<1>(numWorkItems),
5                     [=](id<1> index) { acc[index] = 42.0f; });
6 });
```

Below is an example of invoking a SYCL kernel function with `parallel_for` using a lambda expression and passing a SYCL `item` parameter. In this case, both the global id and global range are queryable. This



variant of `parallel_for` is designed for when it is necessary to query the global range of the index space being executed across.

```
1 myQueue.submit([&](handler& cgh) {
2   accessor acc{myBuffer, cgh, write_only};
3
4   cgh.parallel_for(range<1>(numWorkItems), [=](item<1> item) {
5     // kernel argument type is item
6     size_t index = item.get_linear_id();
7     acc[index] = index;
8   });
9 });
```

Below is an example of invoking a SYCL kernel function with `parallel_for` using a lambda expression and passing `auto` parameter, treated as `item`. In this case, both the global id and global range are queryable. The same effect can be achieved using class with templated `operator()`. This variant of `parallel_for` is designed for when it is necessary to query the global range within which the global id will vary.

```
1 myQueue.submit([&](handler& cgh) {
2   auto acc = myBuffer.get_access<access_mode::write>(cgh);
3
4   cgh.parallel_for(range<1>(numWorkItems), [=](auto item) {
5     // kernel argument type is auto treated as an item
6     size_t index = item.get_linear_id();
7     acc[index] = index;
8   });
9 });
```

Below is an example of invoking a SYCL kernel function with `parallel_for` using a lambda expression and passing an integral type parameter. This example is only valid when calling `parallel_for` with `range<1>`. In this case only the global id is available. This variant of `parallel_for` is designed for when it is not necessary to query the global range of the index space being executed across.

```
1 myQueue.submit([&](handler& cgh) {
2   auto acc = myBuffer.get_access<access_mode::write>(cgh);
3
4   cgh.parallel_for(range<1>(numWorkItems), [=](size_t index) {
5     // kernel argument type is size_t
6     acc[index] = index;
7   });
8 });
```

The `parallel_for` overload without an offset can be called with either a number or a `braced-init-list` with 1-3 elements. In that case the following calls are equivalent:

- `parallel_for(N, some_kernel)` has same effect as `parallel_for(range<1>(N), some_kernel)`
- `parallel_for({N}, some_kernel)` has same effect as `parallel_for(range<1>(N), some_kernel)`
- `parallel_for({N1, N2}, some_kernel)` has same effect as `parallel_for(range<2>(N1, N2), some_kernel)`
- `parallel_for({N1, N2, N3}, some_kernel)` has same effect as `parallel_for(range<3>(N1, N2, N3), some_kernel)`

Below is an example of invoking `parallel_for` with a number instead of an explicit `range` object.

```
1 myQueue.submit([&](handler& cgh) {
2   auto acc = myBuffer.get_access<access_mode::write>(cgh);
3
4   // parallel_for may be called with number (with numWorkItems)
5   cgh.parallel_for(numWorkItems, [=](auto item) {
6     size_t index = item.get_linear_id();
7     acc[index] = index;
8   });
9 });
```

For SYCL kernel functions invoked via the above described overload of the `parallel_for` member function, it is disallowed to use local accessors or to use a `work-group barrier`.

The following two examples show how a kernel function object can be launched over a 3D grid, with 3 elements in each dimension. In the first case work-item ids range from 0 to 2 inclusive, and in the second case work-item ids run from 1 to 3.

```
1 myQueue.submit([&](handler& cgh) {
2   cgh.parallel_for(range<3>(3, 3, 3), // global range
3                   [=](item<3> it) {
4                       //[kernel code]
5                   });
6 });
7
8 // This form of parallel_for with the "offset" parameter is deprecated in SYCL
9 // 2020
10 myQueue.submit([&](handler& cgh) {
11   cgh.parallel_for(range<3>(3, 3, 3), // global range
12                   id<3>(1, 1, 1),    // offset
13                   [=](item<3> it) {
14                       //[kernel code]
15                   });
16 });
```

The last case of a `parallel_for` invocation enables low-level functionality of work-items and work-groups. This becomes valuable when an execution requires groups of work-items to coordinate with one another. These are exposed in SYCL through `parallel_for(nd_range,...)` and the `nd_item` class. In this case, the developer needs to define the `nd_range` that the kernel will execute on in order to have fine grained control of the enqueueing of the kernel. This variation of `parallel_for` expects an `nd_range`, specifying both local and global ranges, defining the global number of work-items and the number in each cooperating work-group. The function object that represents the SYCL kernel function must take one of: 1) a single SYCL `nd_item` parameter, 2) a single generic parameter (`template` parameter or `auto`) that will be treated as an `nd_item` parameter, 3) any other type converted from SYCL `nd_item`, representing the currently executing work-item within the range specified by the `nd_range` parameter. The `nd_item` parameter makes all information about the work-item and its position in the range available, and provides access to functions enabling the use of a `work-group barrier`.



Case 3) above includes user-defined types that can be constructed from `nd_item`, enabling users to layer their own abstractions on top of SYCL.

The following example shows how sixty-four work-items may be launched in a three-dimensional grid with four in each dimension, and divided into eight work-groups. Each group of work-items uses a `work-`

`group_barrier` for coordination.

```

1 myQueue.submit([&](handler& cgh) {
2   cgh.parallel_for(nd_range<3>(range<3>(4, 4, 4), range<3>(2, 2, 2)),
3                     [=](nd_item<3> item) {
4                       //[kernel code]
5                       group_barrier(item.get_group());
6                       //[kernel code]
7                     });
8 });

```

In all of these cases the underlying `nd_range` will be created and the kernel defined as a function object will be created and enqueued as part of the command group scope.

Some forms of `parallel_for` accept an offset parameter of type `id<Dimensions>`, where the number of dimensions of the `id` is the same as the number of dimensions of the `range` that determines the iteration space. These forms of `parallel_for` execute the same number of iterations as the form with no offset. The difference is that the `id` or `item` parameter passed to the kernel function has the value of `offset` implicitly added. This offset parameter is deprecated in SYCL 2020.

An offset can also be passed to the forms of `parallel_for` that accept an `nd_range` via the third parameter to the `nd_range` constructor. These forms of `parallel_for` also execute the same number of iterations as if no offset was specified. The difference is that the `nd_item` parameter passed to the kernel function has the value of the offset implicitly added to the constituent `global id`. This offset parameter is deprecated in SYCL 2020.

A `kernel_handler` can optionally be passed as a parameter to the SYCL kernel function that is invoked by both variants of `parallel_for`.

```

1 myQueue.submit([&](handler& cgh) {
2   cgh.parallel_for(range<3>(3, 3, 3), // global range
3                     [=](item<3> it, kernel_handler kh) {
4                       //[kernel code]
5                     });
6 });
7
8 // This form of parallel_for with the "offset" parameter is deprecated in SYCL
9 // 2020
10 myQueue.submit([&](handler& cgh) {
11   cgh.parallel_for(range<3>(3, 3, 3), // global range
12                    id<3>(1, 1, 1),    // offset
13                    [=](item<3> it, kernel_handler kh) {
14                      //[kernel code]
15                    });
16 });

```

#### 4.9.4.2.3. Parallel for hierarchical invoke (deprecated)

The behavior in this section and the `private_memory` class are deprecated in SYCL 2020.

The hierarchical parallel kernel execution interface provides the same functionality as is available from the `nd_range` interface, but exposed differently. To execute the same sixty-four work-items in eight work-groups that we saw in a previous example, we execute an outer `parallel_for_work_group` call to create the groups. The member function `handler::parallel_for_work_group` is parameterized by the number of work-groups, such that the size of each group is chosen by the runtime, or by the number of work-

groups and number of work-items for users who need more control.

The body of the outer `parallel_for_work_group` call consists of a lambda expression or function object. The body of this function object contains code that is executed only once for the entire work-group. If the code has no side-effects and the compiler heuristic suggests that it is more efficient to do so, this code will be executed for each work-item.

Within this region any variable declared will have the semantics of `local memory`, shared between all `work-items` in the `work-group`. If the device compiler can prove that an array of such variables is accessed only by a single work-item throughout the lifetime of the work-group, for example if access is derived from the id of the work-item with no transformation, then it can allocate the data in private memory or registers instead.

To guarantee use of private per-work-item memory, the `private_memory` class can be used to wrap the data. This class simply constructs private data for a given group across the entire group. The id of the current work-item is passed to any access to grab the correct data.

The `private_memory` class has the following interface:

```
1 namespace sycl {
2  /* Deprecated in SYCL 2020 */
3  template <typename T, int Dimensions = 1> class private_memory {
4  public:
5    // Construct based directly off the number of work-items
6    private_memory(const group<Dimensions>&);
7
8    // Access the instance for the current work-item
9    T& operator()(const h_item<Dimensions>& id);
10 };
11 } // namespace sycl
```

Table 111. Constructor of the `private_memory` class

Constructor	Description
<code>private_memory(const group&lt;Dimensions&gt;&amp;)</code>	Place an object of type <code>T</code> in the underlying private memory of each <code>work-items</code> . The type <code>T</code> must be default constructible. The underlying constructor will be called for each <code>work-item</code> .

Table 112. Member functions of the `private_memory` class

Member functions	Description
<code>T&amp; operator()(const h_item&lt;Dimensions&gt;&amp; id)</code>	Retrieve a reference to the object for the <code>work-items</code> .

`Private memory` is allocated per underlying `work-item`, not per iteration of the `parallel_for_work_item` loop. The number of instances of a private memory object is only under direct control if a work-group size is passed to the `parallel_for_work_group` call. If the underlying work-group size is chosen by the run-time, the number of private memory instances is opaque to the program. Explicit private memory declarations should therefore be used with care and with a full understanding of which instances of a `parallel_for_work_item` loop will share the same underlying variable.

Also within the lambda body can be a sequence of calls to `parallel_for_work_item`. No work-item can begin executing a `parallel_for_work_item` until all work-items in the group have completed executing the

previous `parallel_for_work_item`. As a result the pair of `parallel_for_work_item` calls in the code below is equivalent to the parallel execution with a `work-group barrier` in the earlier example.

```

1 myQueue.submit([&](handler& cgh) {
2   // Issue 8 work-groups of 8 work-items each
3   cgh.parallel_for_work_group(
4     range<3>(2, 2, 2), range<3>(2, 2, 2), [=](group<3> myGroup) {
5     // [workgroup code]
6     int myLocal; // this variable is shared between workitems
7     // this variable will be instantiated for each work-item separately
8     private_memory<int> myPrivate(myGroup);
9
10    // Issue parallel work-items. The number issued per work-group is
11    // determined by the work-group size range of parallel_for_work_group.
12    // In this case, 8 work-items will execute the parallel_for_work_item
13    // body for each of the 8 work-groups, resulting in 64 executions
14    // globally/total.
15    myGroup.parallel_for_work_item([&](h_item<3> myItem) {
16      // [work-item code]
17      myPrivate(myItem) = 0;
18    });
19
20    // Implicit work-group barrier
21
22    // Carry private value across loops
23    myGroup.parallel_for_work_item([&](h_item<3> myItem) {
24      // [work-item code]
25      output[myItem.get_global_id()] = myPrivate(myItem);
26    });
27    // [workgroup code]
28  });
29 });

```

It is valid to use more flexible dimensions of the work-item loops. In the following example we issue 8 work-groups but let the runtime choose their size, by not passing a work-group size to the `parallel_for_work_group` call. The `parallel_for_work_item` loops may also vary in size, with their execution ranges unrelated to the dimensions of the work-group, and the compiler generating an appropriate iteration space to fill the gap. In this case, the `h_item` provides access to local ids and ranges that reflect both kernel and `parallel_for_work_item` invocation ranges.

```

1 myQueue.submit([&](handler& cgh) {
2   // Issue 8 work-groups. The work-group size is chosen by the runtime because
3   // unspecified
4   cgh.parallel_for_work_group(range<3>(2, 2, 2), [=](group<3> myGroup) {
5     // Launch a set of work-items for each work-group. The number of work-items
6     // is chosen by the runtime because the work-group size was not specified to
7     // parallel_for_work_group and a logical range is not specified to
8     // parallel_for_work_item.
9     myGroup.parallel_for_work_item([&](h_item<3> myItem) {
10      // [work-item code]
11    });
12
13    // Implicit work-group barrier
14
15    // Launch 512 logical work-items that will be executed by the underlying

```

```

16 // work-group size chosen by the runtime. myItem allows the logical and
17 // physical work-item IDs to be queried. 512 logical work-items will
18 // execute for each work-group, and the parallel_for body will therefore be
19 // executed 8*512 = 4096 times globally/total.
20 myGroup.parallel_for_work_item(range<3>(8, 8, 8), [=](h_item<3> myItem) {
21     //[work-item code]
22 });
23 //[workgroup code]
24 });
25 });

```

This interface offers a more intuitive way for tiling parallel programming paradigms. In summary, the hierarchical model allows a developer to distinguish the execution at work-group level and at work-item level using the `parallel_for_work_group` and the nested `parallel_for_work_item` functions. It also provides this visibility to the compiler without the need for difficult loop fission such that host execution may be more efficient.

A `kernel_handler` can optionally be passed as a parameter to the SYCL kernel function that is invoked by any variant of `parallel_for_work_group`.

```

1 myQueue.submit([&](handler& cgh) {
2     // Issue 8 work-groups of 8 work-items each
3     cgh.parallel_for_work_group(
4         range<3>(2, 2, 2), range<3>(2, 2, 2),
5         [=](group<3> myGroup, kernel_handler kh) {
6             //[workgroup code]
7             int myLocal; // this variable is shared between workitems
8             // this variable will be instantiated for each work-item separately
9             private_memory<int> myPrivate(myGroup);
10
11             // Issue parallel work-items. The number issued per work-group is
12             // determined by the work-group size range of parallel_for_work_group.
13             // In this case, 8 work-items will execute the parallel_for_work_item
14             // body for each of the 8 work-groups, resulting in 64 executions
15             // globally/total.
16             myGroup.parallel_for_work_item([&](h_item<3> myItem) {
17                 //[work-item code]
18                 myPrivate(myItem) = 0;
19             });
20
21             // Implicit work-group barrier
22
23             // Carry private value across loops
24             myGroup.parallel_for_work_item([&](h_item<3> myItem) {
25                 //[work-item code]
26                 output[myItem.get_global_id()] = myPrivate(myItem);
27             });
28             //[workgroup code]
29         });
30 });

```

#### 4.9.4.3. SYCL functions for explicit memory operations

In addition to `kernels`, `command_group` objects can also be used to perform manual operations on host and device memory by using the copy API of the `command_group_handler`. Manual copy operations can

be seen as specialized kernels executing on the device, except that typically this operations will be implemented using a host API that exists as part of a backend (e.g, OpenCL enqueue copy operations).

These explicit copy operations have a source and a destination. When an accessor is the *source* of the operation, the destination can be a host pointer or another accessor. The *source* accessor must have either `access_mode::read` or `access_mode::read_write` access mode. When an accessor is the *destination* of the explicit copy operation, the source can be a host pointer or another accessor. The *destination* accessor must have either `access_mode::write`, `access_mode::read_write`, `access_mode::discard_write` or `access_mode::discard_read_write` access mode.

When an accessor is used as a parameter to one of these explicit copy operations, the target must be either `target::device` or `target::constant_buffer`.

When accessors are both the source and the destination, the operation is executed on objects controlled by the SYCL runtime. The SYCL runtime is allowed to not perform an explicit in-copy operation if a different path to update the data is available according to the SYCL application memory model.

The most recent copy of the memory object may reside on any context controlled by the SYCL runtime, or on the host in a pointer controlled by the SYCL runtime. The SYCL runtime will ensure that data is copied to the destination once the [command group](#) has completed execution.

Whenever a host pointer is used as either the source or the destination of these explicit memory operations, it is the responsibility of the user for that pointer to have at least as much memory allocated as the accessor is giving access to, e.g: if an accessor accesses a range of 10 elements of `int` type, the host pointer must at least have `10 * sizeof(int)` bytes of memory allocated.

A special case is the `update_host` member function. This member function only requires an accessor, and instructs the runtime to update the internal copy of the data in the host, if any. This is particularly useful when used in conjunction with the `buffer` constructor overloads which accept mutex objects.

[Table 113](#) describes the interface for the explicit copy operations.

Table 113. Member functions of the `handler` class

Member function	Description
<pre>template &lt;typename SrcT, int SrcDims, access_mode SrcMode,           target SrcTgt,           typename DestT, access::placeholder           IsPlaceholder&gt; void copy(accessor&lt;SrcT, SrcDims, SrcMode, SrcTgt,                 IsPlaceholder&gt; src,           std::shared_ptr&lt;DestT&gt; dest)</pre>	<p>Copies the contents of the memory object accessed by <code>src</code> into the memory pointed to by <code>dest</code>. <code>dest</code> must be a host pointer and must have at least as many bytes as the range accessed by <code>src</code>. The type <code>DestT</code> must be <a href="#">device copyable</a>.</p>
<pre>template &lt;typename SrcT, typename DestT, int DestDims,           access_mode DestMode,           target DestTgt, access::placeholder           IsPlaceholder&gt; void copy(std::shared_ptr&lt;SrcT&gt; src,           accessor&lt;DestT, DestDims, DestMode, DestTgt,                 IsPlaceholder&gt; dest)</pre>	<p>Copies the contents of the memory pointed to by <code>src</code> into the memory object accessed by <code>dest</code>. <code>src</code> must be a host pointer and must have at least as many bytes as the range accessed by <code>dest</code>. The type <code>SrcT</code> must be <a href="#">device copyable</a>.</p>



Member function	Description
<pre> template &lt;typename SrcT, int SrcDims, access_mode SrcMode,           target SrcTgt,           typename DestT, access::placeholder           IsPlaceholder&gt; void copy(accessor&lt;SrcT, SrcDims, SrcMode, SrcTgt,                 IsPlaceholder&gt; src,           DestT* dest) </pre>	<p>Copies the contents of the memory object accessed by <code>src</code> into the memory pointed to by <code>dest</code>. <code>dest</code> must be a host pointer and must have at least as many bytes as the range accessed by <code>src</code>. The type <code>DestT</code> must be <a href="#">device copyable</a>.</p>
<pre> template &lt;typename SrcT, typename DestT, int DestDims,           access_mode DestMode,           target DestTgt, access::placeholder           IsPlaceholder&gt; void copy(const SrcT* src,           accessor&lt;DestT, DestDims, DestMode, DestTgt,                 IsPlaceholder&gt; dest) </pre>	<p>Copies the contents of the memory pointed to by <code>src</code> into the memory object accessed by <code>dest</code>. <code>src</code> must be a host pointer and must have at least as many bytes as the range accessed by <code>dest</code>. The type <code>SrcT</code> must be <a href="#">device copyable</a>.</p>
<pre> template &lt;typename SrcT, int SrcDims, access_mode SrcMode,           target SrcTgt,           access::placeholder IsSrcPlaceholder, typename           DestT, int DestDims,           access_mode DestMode, target DestTgt,           access::placeholder IsDestPlaceholder&gt; void copy(accessor&lt;SrcT, SrcDims, SrcMode, SrcTgt,                 IsSrcPlaceholder&gt; src,           accessor&lt;DestT, DestDims, DestMode, DestTgt,                 IsDestPlaceholder&gt; dest) </pre>	<p>Copies the contents of the memory object accessed by <code>src</code> into the memory object accessed by <code>dest</code>. The size of the <code>src</code> accessor determines the number of bytes that are copied, and <code>dest</code> must have at least this many bytes. If the size of <code>dest</code> is too small, the implementation throws a synchronous <a href="#">exception</a> with the <code>errc::invalid</code> error code.</p>
<pre> template &lt;typename T, int Dims, access_mode Mode, target           Tgt,           access::placeholder IsPlaceholder&gt; void update_host(accessor&lt;T, Dims, Mode, Tgt,                 IsPlaceholder&gt; acc) </pre>	<p>The contents of the memory object accessed via <code>acc</code> on the host are guaranteed to be up-to-date after this <a href="#">command group</a> object execution is complete.</p>
<pre> template &lt;typename T, int Dims, access_mode Mode, target           Tgt,           access::placeholder IsPlaceholder&gt; void fill(accessor&lt;T, Dims, Mode, Tgt, IsPlaceholder&gt; dest,           const T&amp; src) </pre>	<p>Replicates the value of <code>src</code> into the memory object accessed by <code>dest</code>.</p>



Member function	Description
<pre>void memcpy(void* dest, const void* src, size_t numBytes)</pre>	<p>Copies <code>numBytes</code> of data from the pointer <code>src</code> to the pointer <code>dest</code>. The <code>dest</code> and <code>src</code> parameters must each either be a host pointer or a pointer within a USM allocation that is accessible on the handler's device. If a pointer is to a USM allocation, that allocation must have been created from the same context as the handler's queue. For more detail on USM, please see <a href="#">Section 4.8</a>.</p>
<pre>template &lt;typename T&gt; void copy(const T* src, T* dest, size_t count)</pre>	<p>Copies <code>count</code> elements of type <code>T</code> from the pointer <code>src</code> to the pointer <code>dest</code>. The <code>dest</code> and <code>src</code> parameters must each either be a host pointer or a pointer within a USM allocation that is accessible on the handler's device. If a pointer is to a USM allocation, that allocation must have been created from the same context as the handler's queue. For more detail on USM, please see <a href="#">Section 4.8</a>.</p> <p>The type <code>T</code> must be <a href="#">device copyable</a>.</p>
<pre>void memset(void* ptr, int value, size_t numBytes)</pre>	<p>Fills <code>numBytes</code> bytes of memory beginning at address <code>ptr</code> with <code>value</code>. The <code>ptr</code> must point within a USM allocation from the same context as the handler's queue, and the pointer must be accessible from the queue's device. Note that <code>value</code> is interpreted as an <code>unsigned char</code>. For more detail on USM, please see <a href="#">Section 4.8</a>.</p>
<pre>template &lt;typename T&gt; void fill(void* ptr, const T&amp; pattern, size_t count)</pre>	<p>Replicates the provided <code>pattern</code> into the memory at address <code>ptr</code>. The <code>ptr</code> must point within a USM allocation from the same context as the handler's queue, and the pointer must be accessible from the queue's device. The <code>pattern</code> is filled <code>count</code> times. For more detail on USM, please see <a href="#">Section 4.8</a>.</p> <p>The type <code>T</code> must be <a href="#">device copyable</a>.</p>

Member function	Description
<code>void prefetch(void* ptr, size_t numBytes)</code>	Enqueues a prefetch of <code>num_bytes</code> of data starting at address <code>ptr</code> . The <code>ptr</code> must point within a USM allocation from the same context as the handler's queue, and the pointer must be accessible from the queue's device. For more detail on USM, please see <a href="#">Section 4.8</a> .
<code>void mem_advise(void* ptr, size_t numBytes, int advice)</code>	Enqueues a command that provides information to the implementation about a region of USM starting at <code>ptr</code> and extending for <code>numBytes</code> bytes. The <code>ptr</code> must point within a USM allocation from the same context as the handler's queue, and the pointer must be accessible from the queue's device. The values for <code>advice</code> are vendor- or backend-specific, with the exception of the value <code>0</code> which reverts the advice for <code>ptr</code> to the default behavior. For more detail on USM, please see <a href="#">Section 4.8</a> .

The listing below illustrates how to use explicit copy operations in SYCL. The example copies half of the contents of a `std::vector` into the device, leaving the rest of the contents of the buffer on the device unchanged.

```

1  const size_t nElems = 10u;
2
3  // Create a vector and fill it with values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
4  std::vector<int> v{nElems};
5  std::iota(std::begin(v), std::end(v), 0);
6
7  // Create a buffer with no associated user storage
8  sycl::buffer<int, 1> b{range<1>(nElems)};
9
10 // Create a queue
11 queue myQueue;
12
13 myQueue.submit([&](handler& cgh) {
14     // Retrieve a ranged write accessor to a global buffer with access to the
15     // first half of the buffer
16     accessor acc{b, cgh, range<1>(nElems / 2), id<1>(0), write_only};
17     // Copy the first five elements of the vector into the buffer associated with
18     // the accessor
19     cgh.copy(v.data(), acc);
20 });

```

#### 4.9.4.4. Functions for using a kernel bundle

```

1  void use_kernel_bundle(

```

```
2    const kernel_bundle<bundle_state::executable>& execBundle);
```

*Effects:* The `command group` associated with the `handler` will use `device images` of the `kernel_bundle execBundle` in any of its `kernel invocation commands`. If the `kernel_bundle` contains multiple `device images` that are compatible with the `device` to which the kernel is submitted, then the `device image` chosen is implementation-defined.

If the `command group` attempts to invoke a kernel that is not contained by a compatible device image in `execBundle`, the `kernel invocation command` throws a synchronous `exception` with the `errc::kernel_not_supported` error code. If the `command group` has a secondary queue, then the `execBundle` must contain a kernel that is compatible with both the primary queue's device and the secondary queue's device, otherwise the `kernel invocation command` throws this exception.

Since the handler method for setting specialization constants is incompatible with the kernel bundle method, applications should not call this function if `handler::set_specialization_constant()` has been previously called for this same `command group`.

*Throws:*

- An `exception` with the `errc::invalid` error code if the `context` associated with the `command group handler` via its associated primary `queue` or the `context` associated with the secondary `queue` (if provided) is different from the `context` associated with the `kernel bundle` specified by `execBundle`.
- An `exception` with the `errc::invalid` error code if `handler::set_specialization_constant()` has been called for this `command group`.

## 4.9.5. Specialization constants

Device code can make use of `specialization constants` which represent constants whose values can be set dynamically during execution of the `SYCL application`. The values of these constants are fixed when a `SYCL kernel function` is invoked, and they do not change during the execution of the kernel. However, the application is able to set a new value for a specialization constant each time a kernel is invoked, so the values can be tuned differently for each invocation.

There are two methods for an application to use specialization constants, one method requires creating a `kernel_bundle` object and the other does not. The syntax for both methods is mostly the same. Both methods declare specialization constants in the same way, and kernels read their values in the same way. The main difference is whether their values are set via `handler::set_specialization_constant()` or via `kernel_bundle::set_specialization_constant()`. These two methods are incompatible with one another, so they may not both be used by the same `command group`.



Implementations that support online compilation of kernel bundles will likely implement both methods of specialization constants using kernel bundles. Therefore, applications should expect that there is some overhead associated with invoking a kernel with new values for its specialization constants. A typical implementation records the values of specialization constants set via `handler::set_specialization_constant()` and remembers these values until a kernel is invoked (e.g. via `parallel_for()`). At this point, the implementation determines the bundle that contains the invoked kernel. If that bundle has already been compiled for the handler's device and compiled with the correct values for the specialization constants, the kernel is scheduled for invocation. Otherwise, the implementation compiles the bundle before scheduling the kernel for invocation. Therefore, applications that frequently change the values of specialization constants may see an overhead associated with recompilation of the kernel's bundle.

### 4.9.5.1. Declaring a specialization constant

Specialization constants must be declared using the `specialization_id` class with the following restric-

tions:

- the template parameter `T` must be a [device copyable](#) type;
- the `specialization_id` variable must be declared as `constexpr`;
- the `specialization_id` variable must be declared in either namespace scope or in class scope;
- if the `specialization_id` variable is declared in class scope, it must have public accessibility when referenced from namespace scope;
- the `specialization_id` variable may not be shadowed by another identifier `X` which has the same name and is declared in an `inline` namespace, such that the `specialization_id` variable is no longer accessible after the declaration of `X`;
- if the `specialization_id` variable is declared in a namespace, none of the enclosing namespace names `N` may be shadowed by another identifier `X` which has the same name as `N` and is declared in an `inline` namespace, such that `N` is no longer accessible after the declaration of `X`.



The expectation is that some implementations may conceptually insert code at the end of a translation unit which references each `specialization_id` variable that is declared in that translation unit. The restrictions listed above make this possible by ensuring that these variables are accessible at the end of the translation unit.

The following example illustrates some of these restrictions:

```

1 #include <sycl/sycl.hpp>
2 using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names
3
4 struct Compound {
5     int i;
6     float f;
7 };
8
9 constexpr specialization_id<int> a{1};           // OK
10 constexpr specialization_id<Compound> b{2, 3.14}; // OK
11 inline constexpr specialization_id<int> c{3};     // OK
12 static constexpr specialization_id<int> d{4};     // OK
13 specialization_id<int> e{5};                      // ILLEGAL: not constexpr
14
15 struct Bar {
16     static constexpr specialization_id<int> f{6}; // OK
17 };
18 struct Baz {
19     struct Inner {
20         static constexpr specialization_id<int> g{7}; // OK
21     };
22 };
23 class Boo {
24     static constexpr specialization_id<int> h{8}; // ILLEGAL: not public member
25 };
26
27 void Func() {
28     static constexpr specialization_id<int> i{9}; // ILLEGAL: not at namespace
29                                                     // or class scope
30     /* ... */
31 }
32
33 constexpr specialization_id<int> same_name{10}; // OK

```

```

34 namespace foo {
35 constexpr specialization_id<int> same_name{11}; // OK
36 }
37 namespace {
38 constexpr specialization_id<int> same_name{12}; // OK
39 }
40 inline namespace other {
41 int same_name; // ILLEGAL: shadows "specialization_id" variable with same name
42                // in enclosing namespace scope
43 }
44 inline namespace other2 {
45 namespace foo { // ILLEGAL: namespace name shadows "::foo" namespace which
46                // contains "specialization_id" variable.
47 } // namespace foo
48 } // namespace other2

```

A synopsis of this class is shown below.

```

1 namespace sycl {
2
3 template <typename T> class specialization_id {
4 public:
5     using value_type = T;
6
7     template <class... Args> explicit constexpr specialization_id(Args&&... args);
8
9     specialization_id(const specialization_id& rhs) = delete;
10    specialization_id(specialization_id&& rhs) = delete;
11    specialization_id& operator=(const specialization_id& rhs) = delete;
12    specialization_id& operator=(specialization_id&& rhs) = delete;
13 };
14
15 } // namespace sycl

```

#### 4.9.5.1.1. Constructors

```
template <class... Args> explicit constexpr specialization_id(Args&&... args);
```

*Constraints:* Available only when `std::is_constructible_v<T, Args...>` evaluates to `true`.

*Effects:* Constructs a `specialization_id` containing an instance of `T` initialized with `args...`, which represents the specialization constant's default value.

#### 4.9.5.1.2. Special member functions

```

specialization_id(const specialization_id& rhs) = delete;           // (1)
specialization_id(specialization_id&& rhs) = delete;             // (2)
specialization_id& operator=(const specialization_id& rhs) = delete; // (3)
specialization_id& operator=(specialization_id&& rhs) = delete;   // (4)

```

1. Deleted copy constructor.
2. Deleted move constructor.

3. Deleted copy assignment operator.
4. Deleted move assignment operator.

#### 4.9.5.2. Setting and getting the value of a specialization constant

If the application uses specialization constants without creating a `kernel_bundle` object, it can set and get their values from `command group scope` by calling member functions of the `handler` class. These member functions have a template parameter `SpecName` whose value must be a reference to a variable of type `specialization_id`, which defines the type and default value of the specialization constant.

When not using a kernel bundle, the value of a specialization constant that is used in a kernel invoked from a `command group` is affected by calls to set its value from that same `command group`, but it is not affected by calls from other `command groups` even if those calls are from another invocation of the same `command group function object`.

```
template <auto& SpecName>
void set_specialization_constant(
    typename std::remove_reference_t<decltype(SpecName)>::value_type value);
```

*Effects:* Sets the value of the specialization constant whose address is `SpecName` for this handler's `command group`. If the specialization constant's value was previously set in this same `command group`, the value is overwritten.

This function may be called even if the specialization constant `SpecName` isn't used by the kernel that is invoked by this handler's `command group`. Doing so has no effect on the invoked kernel.

*Throws:*

- An `exception` with the `errc::invalid` error code if a kernel bundle has been bound to the `handler` via `use_kernel_bundle()`.

```
template <auto& SpecName>
typename std::remove_reference_t<decltype(SpecName)>::value_type
get_specialization_constant();
```

*Returns:* The value of the specialization constant whose address is `SpecName` for this handler's `command group`. If the value was previously set in this handler's `command group`, that value is returned. Otherwise, the specialization constant's default value is returned.

*Throws:*

- An `exception` with the `errc::invalid` error code if a kernel bundle has been bound to the `handler` via `use_kernel_bundle()`.

#### 4.9.5.3. Reading the value of a specialization constant from device code

In order to read the value of a specialization constant from device code, the `SYCL kernel function` must be declared to take an object of type `kernel_handler` as its last parameter. The `SYCL runtime` constructs this object, which has a member function for reading the specialization constant's value. A synopsis of this class is shown below.

```
1 namespace sycl {
2
3   class kernel_handler {
4   public:
```

```

5  template <auto& SpecName>
6  typename std::remove_reference_t<decltype(SpecName)>::value_type
7  get_specialization_constant();
8  };
9
10 } // namespace sycl

```

#### 4.9.5.3.1. Member functions

```

1  template<auto& SpecName>
2  typename std::remove_reference_t<decltype(SpecName)>::value_type
3  get_specialization_constant();

```

*Returns:* The value of the [specialization constant](#) whose address is [SpecName](#). For a kernel invoked from a [command group](#) that was not bound to a kernel bundle, the value is the same as what would have been returned if `handler::get_specialization_constant()` was called immediately before invoking the kernel. For a kernel invoked from a [command group](#) that was bound to a kernel bundle, the value is the same as what would be returned if `kernel_bundle::get_specialization_constant()` was called on the bound bundle.

#### 4.9.5.4. Example usage

The following example performs a convolution and uses [specialization constants](#) to set the values of the coefficients.

```

1  #include <sycl/sycl.hpp>
2  using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names
3
4  using coeff_t = std::array<std::array<float, 3>, 3>;
5
6  // Read coefficients from somewhere.
7  coeff_t get_coefficients();
8
9  // Identify the specialization constant.
10 constexpr specialization_id<coeff_t> coeff_id;
11
12 void do_conv(buffer<float, 2> in, buffer<float, 2> out) {
13     queue myQueue;
14
15     myQueue.submit([&](handler& cgh) {
16         accessor in_acc{in, cgh, read_only};
17         accessor out_acc{out, cgh, write_only};
18
19         // Set the coefficient of the convolution as constant.
20         // This will build a specific kernel the coefficient available as literals.
21         cgh.set_specialization_constant<coeff_id>(get_coefficients());
22
23         cgh.parallel_for<class Convolution>(in.get_range(), [=](item<2> item_id,
24                                                             kernel_handler h) {
25             float acc = 0;
26             coeff_t coeff = h.get_specialization_constant<coeff_id>();
27             for (int i = -1; i <= 1; i++) {
28                 if (item_id[0] + i < 0 || item_id[0] + i >= in_acc.get_range()[0])
29                     continue;

```



```

30     for (int j = -1; j <= 1; j++) {
31         if (item_id[1] + j < 0 || item_id[1] + j >= in_acc.get_range()[1])
32             continue;
33         // The underlying JIT can see all the values of the array returned
34         // by coeff.get().
35         acc += coeff[i + 1][j + 1] * in_acc[item_id[0] + i][item_id[1] + j];
36     }
37 }
38 out_acc[item_id] = acc;
39 });
40 });
41
42 myQueue.wait();
43 }

```

## 4.10. Host tasks

### 4.10.1. Overview

A [host task](#) is a native C++ callable which is scheduled by the [SYCL runtime](#). A [host task](#) is submitted to a [queue](#) via a [command group](#) by a [host task command](#).

When a [host task command](#) is submitted to a [queue](#) it is scheduled based on its data dependencies with other [commands](#) including [kernel invocation commands](#) and asynchronous copies, resolving any requisites created by [accessors](#) attached to the [command group](#) as defined in [Section 3.8.1](#).

Since a [host task](#) is invoked directly by the [SYCL runtime](#) rather than being compiled as a [SYCL kernel function](#), it does not have the same restrictions as a [SYCL kernel function](#), and can therefore contain any arbitrary C++ code.

Capturing [accessors](#) in a [host task](#) is allowed, however, capturing or using any other SYCL class that has reference semantics (see [Section 4.5.2](#)) is undefined behavior.

A [host task](#) can be enqueued on any [queue](#) and the callable will be invoked directly by the SYCL runtime, regardless of which [device](#) the [queue](#) is associated with.

A [host task](#) is enqueued on a [queue](#) via the `host_task` member function of the `handler` class. The `event` returned by the submission of the associated [command group](#) enters the completed state (corresponding to a status of `info::event_command_status::complete`) once the invocation of the provided C++ callable has returned. Any uncaught exception thrown during the execution of a [host task](#) will be turned into an [asynchronous error](#) that can be handled as described in [Section 4.13.1.1](#).

A [host task](#) can optionally be used to interoperate with the [native backend objects](#) associated with the [queue](#) executing the [host task](#), the [context](#) that the [queue](#) is associated with, the [device](#) that the [queue](#) is associated with and the [accessors](#) that have been captured in the callable, via an optional `interop_handle` parameter.

This allows [host tasks](#) to be used for two purposes: either as a task which can perform arbitrary C++ code within the scheduling of the [SYCL runtime](#) or as a task which can perform interoperability at a point within the scheduling of the [SYCL runtime](#).

For the former use case, construct a buffer accessor with `target::host_task` or an image accessor with `image_target::host_task`. This makes the buffer or image available on the host during execution of the [host task](#).

For the latter case, construct a buffer accessor with `target::device` or `target::constant_buffer`, or con-



struct an image accessor with `image_target::device`. This makes the buffer or image available on the device that is associated with the queue used to submit the `host task`, so that it can be accessed via interoperability member functions provided by the `interop_handle` class.

Local `accessors` cannot be used within a `host task`.



If a C++ lambda is passed to a `host task`, the lambda may capture by reference or by value. Since the `host task` callable executes asynchronously, care must be taken to ensure that lifetimes of objects captured by reference by a `host task` lambda last at least until the `host task` completes.

```

1 namespace sycl {
2
3 class interop_handle {
4 private:
5     interop_handle(__unspecified__);
6
7 public:
8     interop_handle() = delete;
9
10    backend get_backend() const noexcept;
11
12    template <backend Backend, typename DataT, int Dims, access_mode AccessMode,
13              target AccessTarget, access::placeholder isPlaceholder>
14    backend_return_t<Backend, buffer<DataT, Dims>>
15    get_native_mem(const accessor<DataT, Dims, AccessMode, AccessTarget,
16                          isPlaceholder>& bufferAccessor) const;
17
18    template <backend Backend, typename DataT, int Dims, access_mode AccMode>
19    backend_return_t<Backend, unsampled_image<Dims>> get_native_mem(
20        const unsampled_image_accessor<DataT, Dims, AccMode,
21                                      image_target::device>& imageAcc) const;
22
23    template <backend Backend, typename DataT, int Dims>
24    backend_return_t<Backend, sampled_image<Dims>> get_native_mem(
25        const sampled_image_accessor<DataT, Dims, image_target::device>& imageAcc)
26        const;
27
28    template <backend Backend>
29    backend_return_t<Backend, queue> get_native_queue() const;
30
31    template <backend Backend>
32    backend_return_t<Backend, device> get_native_device() const;
33
34    template <backend Backend>
35    backend_return_t<Backend, context> get_native_context() const;
36 };
37
38 class handler {
39 public:
40     // ...
41
42     template <typename T>
43     void host_task(T&& hostTaskCallable);
44
45     // ...

```

```

46 };
47
48 } // namespace sycl

```

## 4.10.2. Class `interop_handle`

The `interop_handle` class is an abstraction over the `queue` which is being used to invoke the `host task` and its associated `device` and `context`. It also represents the state of the `SYCL runtime` dependency model at the point the `host task` is invoked.

The `interop_handle` class provides access to the `native backend object` associated with the `queue`, `device`, `context` and any `buffers` or `images` that are captured in the callable being invoked in order to allow a `host task` to be used for interoperability purposes.

An `interop_handle` cannot be constructed by user-code, only by the `SYCL runtime`.

```

1 class interop_handle;

```

### 4.10.2.1. Constructors

```

1 private:
2   interop_handle(__unspecified__); // (1)
3
4 public:
5   interop_handle() = delete; // (2)

```

1. Private implementation-defined constructor with unspecified arguments so that the `SYCL runtime` can construct a `interop_handle`.
2. Explicitly deleted default constructor.

### 4.10.2.2. Member functions

```

1 backend get_backend() const noexcept;

```

1. *Returns:* Returns a `backend` identifying the `SYCL backend` associated with the `queue` associated with this `interop_handle`.

### 4.10.2.3. Template member functions `get_native_*`

```

1   // SPDX-License-Identifier: MIT
2
3   template <backend Backend, typename DataT, int Dims, access_mode AccMode,
4             target AccTarget, access::placeholder IsPlaceholder>
5   backend_return_t<Backend, buffer<DataT, Dims>>
6   get_native_mem(const accessor<DataT, Dims, AccMode, AccTarget, // (1)
7                  IsPlaceholder>& bufferAcc) const;
8
9   template <backend Backend, typename DataT, int Dims, access_mode AccMode>
10  backend_return_t<Backend, unsampled_image<Dims>> get_native_mem( // (2)
11    const unsampled_image_accessor<DataT, Dims, AccMode, image_target::device>&
12    imageAcc) const;
13

```

```

14 template <backend Backend, typename DataT, int Dims>
15 backend_return_t<Backend, sampled_image<Dims>> get_native_mem( // (3)
16     const sampled_image_accessor<DataT, Dims, image_target::device>& imageAcc)
17     const;
18
19 template <backend Backend>
20 backend_return_t<Backend, queue> get_native_queue() const; // (4)
21
22 template <backend Backend>
23 backend_return_t<Backend, device> get_native_device() const; // (5)
24
25 template <backend Backend>
26 backend_return_t<Backend, context> get_native_context() const; // (6)

```

1. *Constraints:* Available only if the optional interoperability function `get_native` taking a `buffer` is available and if `accTarget` is `target::device`.

*Returns:* The `native backend object` associated with the underlying `buffer` of `accessor bufferAcc`. The `native backend object` returned must be in a state where it represents the memory in its current state within the `SYCL runtime` dependency model and is capable of being used in a way appropriate for the associated `SYCL backend`. It is undefined behavior to use the `native backend object` outside of the scope of the `host task`.

*Throws:* An `exception` with the `errc::invalid` error code if the `accessor bufferAcc` was not registered with the `command group` which contained the `host task`. Must throw an `exception` with the `errc::backend_mismatch` error code if `Backend != get_backend()`.

2. *Constraints:* Available only if the optional interoperability function `get_native` taking an `unsampled_image` is available.

*Returns:* The `native backend object` associated with with the underlying `unsampled_image` of `accessor imageAcc`. The `native backend object` returned must be in a state where it represents the memory in its current state within the `SYCL runtime` dependency model and is capable of being used in a way appropriate for the associated `SYCL backend`. It is undefined behavior to use the `native backend object` outside of the scope of the `host task`.

*Throws:* An `exception` with the `errc::invalid` error code if the `accessor imageAcc` was not registered with the `command group` which contained the `host task`.

3. *Constraints:* Available only if the optional interoperability function `get_native` taking an `sampled_image` is available.

*Returns:* The `native backend object` associated with with the underlying `sampled_image` of `accessor imageAcc`. The `native backend object` returned must be in a state where it represents the memory in its current state within the `SYCL runtime` dependency model and is capable of being used in a way appropriate for the associated `SYCL backend`. It is undefined behavior to use the `native backend object` outside of the scope of the `host task`.

*Throws:* An `exception` with the `errc::invalid` error code if the `accessor imageAcc` was not registered with the `command group` which contained the `host task`. Must throw an `exception` with the `errc::backend_mismatch` error code if `Backend != get_backend()`.

4. *Constraints:* Available only if the optional interoperability function `get_native` taking a `queue` is available.

*Returns:* The `native backend object` associated with the `queue` that the `host task` was submitted to. If the `command group` was submitted with a secondary `queue` and the fall-back was triggered, the `queue` that is associated with the `interop_handle` must be the fall-back `queue`. The `native backend`

`object` returned must be in a state where it is capable of being used in a way appropriate for the associated `SYCL backend`. It is undefined behavior to use the `native backend object` outside of the scope of the `host task`.

*Throws:* Must throw an `exception` with the `errc::backend_mismatch` error code if `Backend != get_backend()`.

5. *Constraints:* Available only if the optional interoperability function `get_native` taking a `device` is available.

*Returns:* The `native backend object` associated with the `device` that is associated with the `queue` that the `host task` was submitted to. The `native backend object` returned must be in a state where it is capable of being used in a way appropriate for the associated `SYCL backend`. It is undefined behavior to use the `native backend object` outside of the scope of the `host task`.

*Throws:* Must throw an `exception` with the `errc::backend_mismatch` error code if `Backend != get_backend()`.

6. *Constraints:* Available only if the optional interoperability function `get_native` taking a `context` is available.

*Returns:* The `native backend object` associated with the `context` that is associated with the `queue` that the `host task` was submitted to. The `native backend object` returned must be in a state where it is capable of being used in a way appropriate for the associated `SYCL backend`. It is undefined behavior to use the `native backend object` outside of the scope of the `host task`.

*Throws:* Must throw an `exception` with the `errc::backend_mismatch` error code if `Backend != get_backend()`.

### 4.10.3. Additions to the `handler` class

This section describes member functions in the `command group handler` class that are used with host tasks.

```

1 class handler {
2 public:
3     // ...
4
5     template <typename T>
6     void host_task(T&& hostTaskCallable); // (1)
7
8     // ...
9 };

```

1. *Effects:* Enqueues an implementation-defined command to the `SYCL runtime` to invoke `hostTaskCallable` exactly once. The scheduling of the invocation of `hostTaskCallable` in relation to other `commands` enqueued to the `SYCL runtime` must be in accordance with the dependency model described in [Section 3.8.1](#). Initializes an `interop_handle` object and passes it to `hostTaskCallable` when it is invoked if `std::is_invocable_v<T, interop_handle>` evaluates to `true`, otherwise invokes `hostTaskCallable` as a nullary function.

## 4.11. Kernel bundles

Kernel bundles provide several features to a `SYCL application`. For implementations that support an online compiler, they provide fine grained control over the online compilation of device code. For example, an application can use a kernel bundle to compile its `kernels` at a specific time during the applica-

tion's execution (such as during its initialization), rather than relying on the implementation's default behavior (which may not compile kernels until they are submitted).

Kernel bundles also provide a way for the application to set the values of specialization constants in many kernels before any of them are submitted to a device, which could potentially be more efficient in some cases.

Kernel bundles provide a way for the application to introspect its kernels. For example, an application can use a bundle to query a kernel's work-group size when it is run on a specific device.

Finally, kernel bundles provide an extension point to interoperate with backend and device specific features. Some examples of this include invocation of device specific built-in kernels, online compilation of kernel code with vendor specific options, or interoperation with kernels created with backend APIs.

### 4.11.1. Overview

A kernel bundle is a high-level abstraction which represents a set of [kernels](#) that are associated with a [context](#) and can be executed on a number of [devices](#), where each device is associated with that same context. Depending on how a bundle is obtained, it could represent all of the [SYCL kernel functions](#) in the [SYCL application](#), or a certain subset of them.

A kernel bundle is composed of one or more [device images](#), where each device image is an indivisible unit of compilation and/or linking. When the [SYCL runtime](#) compiles or links one of the kernels represented by the device image, it must also compile or link any other kernels the device image represents. Once a device image is compiled and linked, any of the other kernels which that device image represents may be invoked without further compilation or linking.

Each [SYCL kernel function](#) a bundle represents must reside in at least one of the bundle's device images. However, it is not necessary for each device image to contain all of the kernel functions that the bundle represents. The granularity in which kernel functions are grouped into device images is an implementation detail.



To illustrate the intent of device images, a hypothetical implementation could represent an application's kernel functions in both the SPIR-V format and also in a native device code format. The implementation's ahead-of-time compiler in this example produces device images with native code for certain devices and also produces SPIR-V device images for use with other devices. Note that in such an implementation, a particular kernel function could be represented in more than one device image.

An implementation could choose to have all kernel functions from all translation units grouped together in a single device image, to have each kernel function represented in its own device image, or to group kernel functions in some other way.

Each device associated with a kernel bundle must have at least one compatible device image, meaning that the implementation can either invoke the image's kernel functions directly on the device or that the implementation can translate the device image into a format that allows it to invoke the kernel functions.

An outcome of this definition is that each kernel function in a bundle must be invocable on at least one of the devices associated with the bundle. However, it is not necessary for every kernel function in the bundle to be invocable on every associated device.



One common reason why a kernel function might not be invocable on every device associated with a bundle is if the kernel uses optional device features. It's possible that these features are available to only some devices in the bundle.

The use of optional device features could affect how the implementation groups kernels

into device images, depending on how these features are represented. For example, consider an implementation where the optional feature is represented in SPIR-V but translation of that SPIR-V into native code will fail if the target device does not support the feature. In such an implementation, kernels that use optional features should not be grouped into the same device image as kernels that do not use these features. Since a device image is an indivisible unit of compilation, doing so would cause a compilation failure if a kernel K1 is invoked on a device D1 if K1 happened to reside in the same device image as another kernel K2 that used a feature which is not supported on device D1.

See [Section 5.7](#) for more about optional device features.

A [SYCL application](#) can obtain a kernel bundle by calling one of the overloads of the `get_kernel_bundle()` free function. Certain backends may provide additional mechanisms for obtaining bundles with other representations. If this is supported, the backend specification document will describe the details.

Once a kernel bundle has been obtained there are a number of free functions for performing compilation, linking and joining. Once a bundle is compiled and linked, the application can invoke kernels from the bundle by calling `handler::use_kernel_bundle()` as described in [Section 4.9.4.4](#).

## 4.11.2. Synopsis

```

1 namespace sycl {
2
3 enum class bundle_state : /* unspecified */ { input, object, executable };
4
5 class kernel_id { /* ... */
6 };
7
8 template <bundle_state State> class kernel_bundle { /* ... */
9 };
10
11 template <typename KernelName> kernel_id get_kernel_id();
12
13 std::vector<kernel_id> get_kernel_ids();
14
15 template <bundle_state State>
16 kernel_bundle<State> get_kernel_bundle(const context& ctxt);
17
18 template <bundle_state State>
19 kernel_bundle<State> get_kernel_bundle(const context& ctxt,
20                                     const std::vector<kernel_id>& kernelIds);
21
22 template <typename KernelName, bundle_state State>
23 kernel_bundle<State> get_kernel_bundle(const context& ctxt);
24
25 template <bundle_state State>
26 kernel_bundle<State> get_kernel_bundle(const context& ctxt,
27                                     const std::vector<device>& devs);
28
29 template <bundle_state State>
30 kernel_bundle<State> get_kernel_bundle(const context& ctxt,
31                                     const std::vector<device>& devs,
32                                     const std::vector<kernel_id>& kernelIds);
33
34 template <typename KernelName, bundle_state State>

```



```

35 kernel_bundle<State> get_kernel_bundle(const context& ctxt,
36                                         const std::vector<device>& devs);
37
38 template <bundle_state State, typename Selector>
39 kernel_bundle<State> get_kernel_bundle(const context& ctxt, Selector selector);
40
41 template <bundle_state State, typename Selector>
42 kernel_bundle<State> get_kernel_bundle(const context& ctxt,
43                                         const std::vector<device>& devs,
44                                         Selector selector);
45
46 template <bundle_state State> bool has_kernel_bundle(const context& ctxt);
47
48 template <bundle_state State>
49 bool has_kernel_bundle(const context& ctxt,
50                         const std::vector<kernel_id>& kernelIds);
51
52 template <typename KernelName, bundle_state State>
53 bool has_kernel_bundle(const context& ctxt);
54
55 template <bundle_state State>
56 bool has_kernel_bundle(const context& ctxt, const std::vector<device>& devs);
57
58 template <bundle_state State>
59 bool has_kernel_bundle(const context& ctxt, const std::vector<device>& devs,
60                         const std::vector<kernel_id>& kernelIds);
61
62 template <typename KernelName, bundle_state State>
63 bool has_kernel_bundle(const context& ctxt, const std::vector<device>& devs);
64
65 bool is_compatible(const std::vector<kernel_id>& kernelIds, const device& dev);
66
67 template <typename KernelName> bool is_compatible(const device& dev);
68
69 template <bundle_state State>
70 kernel_bundle<State> join(const std::vector<kernel_bundle<State>>& bundles);
71
72 kernel_bundle<bundle_state::object>
73 compile(const kernel_bundle<bundle_state::input>& inputBundle,
74         const property_list& propList = {});
75
76 kernel_bundle<bundle_state::object>
77 compile(const kernel_bundle<bundle_state::input>& inputBundle,
78         const std::vector<device>& devs, const property_list& propList = {});
79
80 kernel_bundle<bundle_state::executable>
81 link(const kernel_bundle<bundle_state::object>& objectBundle,
82       const property_list& propList = {});
83
84 kernel_bundle<bundle_state::executable>
85 link(const std::vector<kernel_bundle<bundle_state::object>>& objectBundles,
86       const property_list& propList = {});
87
88 kernel_bundle<bundle_state::executable>
89 link(const kernel_bundle<bundle_state::object>& objectBundle,
90       const std::vector<device>& devs, const property_list& propList = {});

```

```

91
92 kernel_bundle<bundle_state::executable>
93 link(const std::vector<kernel_bundle<bundle_state::object>>& objectBundles,
94       const std::vector<device>& devs, const property_list& propList = {});
95
96 kernel_bundle<bundle_state::executable>
97 build(const kernel_bundle<bundle_state::input>& inputBundle,
98        const property_list& propList = {});
99
100 kernel_bundle<bundle_state::executable>
101 build(const kernel_bundle<bundle_state::input>& inputBundle,
102        const std::vector<device>& devs, const property_list& propList = {});
103
104 } // namespace sycl

```

### 4.11.3. Fixed-function built-in kernels

SYCL allows a [SYCL backend](#) to expose fixed functionality as non-programmable built-in kernels. The availability and behavior of these built-in kernels are backend specific and are not required to follow the SYCL execution and memory models. However, the basic interface is common to all backends.

### 4.11.4. Bundle states

A [kernel bundle](#) can be in one of three different [bundle states](#) which are represented by an enum class called [bundle\\_state](#). [Table 114](#) describes the semantics of these three states.

The states form a progression. A bundle in [bundle\\_state::input](#) can be translated into [bundle\\_state::object](#) by online compilation of the bundle. A bundle in [bundle\\_state::object](#) can be translated into [bundle\\_state::executable](#) by online linking.



Each implementation is free to define the "online compilation" and "online linking" operations as it sees fit, so long as this progression of bundle states is preserved and so long as the bundles in each state behave as specified.

There is no requirement that an implementation must expose kernels in [bundle\\_state::input](#) or [bundle\\_state::object](#). In fact, an implementation could expose some kernels in these states but not others. For example, this behavior could be controlled by implementation specific options to the ahead-of-time compiler. Kernels that are not exposed in these states cannot be online compiled or online linked by the application.

All kernels defined in the [SYCL application](#), however, must be exposed in [bundle\\_state::executable](#) because this is the only state that allows a kernel to be invoked on a device. Device built-in kernels are also exposed in [bundle\\_state::executable](#).

If an application exposes a bundle in [bundle\\_state::input](#) for a device D, then the implementation must also provide an online compiler for device D. Therefore, an application need not explicitly test for [aspect::online\\_compiler](#) if it successfully obtains a bundle in [bundle\\_state::input](#) for that device. Likewise, an implementation must provide an online linker for device D if it exposes a bundle in [bundle\\_state::object](#) for device D.

*Table 114. Enumeration of possible bundle states*



Bundle State	Description
<code>bundle_state::input</code>	The <a href="#">device images</a> in the kernel bundle have a format that must be compiled and linked before their kernels can be invoked. For example, an implementation could use this state for device images that are stored in an intermediate language format or for device images that are stored as source code strings.
<code>bundle_state::object</code>	The <a href="#">device images</a> in the kernel bundle have a format that must be linked before their kernels can be invoked.
<code>bundle_state::executable</code>	The <a href="#">device images</a> in the kernel bundle are in a format that allows them to be invoked on a device. For example, an implementation could use this state for device images that have been compiled into the device's native code.

### 4.11.5. Kernel identifiers

Some of the functions related to kernel bundles take an input parameter of type `kernel_id` which identifies a kernel. A synopsis of the `kernel_id` class is shown below along with a description of its member functions. Additionally, this class provides the common special member functions and common member functions that are listed in [Section 4.5.2](#) in [Table 7](#) and [Table 8](#), respectively.

As with all SYCL objects that have the common reference semantics, kernel identifiers are equality comparable. Two `kernel_id` objects compare equal if and only if they refer to the same application kernel or to the same device built-in kernel.

There is no public default constructor for this class.

```

1 namespace sycl {
2
3 class kernel_id {
4 public:
5     kernel_id() = delete;
6
7     const char* get_name() const noexcept;
8 };
9
10 } // namespace sycl

```

```
const char* get_name() const noexcept;
```

*Returns:* An implementation-defined null-terminated string containing the name of the kernel. There is no guarantee that this name is unique amongst all the kernels, nor is there a guarantee that the name is stable from one run of the application to another. The lifetime of the memory containing the name is unspecified.



In practice, the lifetime of the memory containing the name will typically extend until the application terminates, unless the kernel associated with the name comes from a dynamic library. In this case, the lifetime of the memory may end if the dynamic library is unloaded.

## 4.11.6. Obtaining a kernel identifier

An application can obtain an identifier for a kernel that is defined in the application by calling one of the following free functions, or it may obtain an identifier for a device's built-in kernels by querying the device with `info::device::built_in_kernel_ids`.

```
template <typename KernelName> kernel_id get_kernel_id();
```

*Preconditions:* The template parameter `KernelName` must be the `type kernel name` of a kernel that is defined in the `SYCL application`. Since lambda expressions have no standard type name, kernels defined as lambda expressions must specify a `KernelName` in their `kernel invocation command` in order to obtain their identifier via this function. Applications which call `get_kernel_id()` for a `KernelName` that is not defined are ill formed, and the implementation must issue a diagnostic in this case.

*Returns:* The identifier of the kernel associated with `KernelName`.

```
std::vector<kernel_id> get_kernel_ids();
```

*Returns:* A vector with the identifiers for all kernels defined in the `SYCL application`. This does not include identifiers for any device built-in kernels.

## 4.11.7. Obtaining a kernel bundle

A `SYCL application` can obtain a kernel bundle by calling one of the overloads of the free function `get_kernel_bundle()`. The implementation may return a bundle that consists of device images that were created by the ahead-of-time compiler, or it may call the online compiler or linker to create the bundle's device images in the requested state. A bundle may also contain device images that represent a device's built-in kernels.

When `get_kernel_bundle()` is used to obtain a kernel bundle in `bundle_state::object` or `bundle_state::executable`, any specialization constants in the bundle will have their default values.

```
template <bundle_state State>
kernel_bundle<State> get_kernel_bundle(const context& ctxt,
                                       const std::vector<device>& devs);
```

*Returns:* A kernel bundle in state `State` which contains all of the `kernels` in the application which are compatible with at least one of the devices in `devs`. This does not include any device built-in kernels. The bundle's set of associated devices is `devs` (with any duplicate devices removed).

Since the implementation may not represent all kernels in `bundle_state::input` or `bundle_state::object`, calling this function with one of those states may return a bundle that is missing some of the application's kernels.

*Throws:*

- An `exception` with the `errc::invalid` error code if any of the devices in `devs` is not one of devices contained by the context `ctxt` or is not a `descendent device` of some device in `ctxt`.
- An `exception` with the `errc::invalid` error code if the `devs` vector is empty.
- An `exception` with the `errc::invalid` error code if `State` is `bundle_state::input` and any device in `devs` does not have `aspect::online_compiler`.
- An `exception` with the `errc::invalid` error code if `State` is `bundle_state::object` and any device in `devs` does not have `aspect::online_linker`.

- An **exception** with the `errc::build` error code if `State` is `bundle_state::object` or `bundle_state::executable`, if the implementation needs to perform an online compile or link, and if the online compile or link fails.

```
template <bundle_state State>
kernel_bundle<State> get_kernel_bundle(const context& ctxt,
                                       const std::vector<device>& devs,
                                       const std::vector<kernel_id>& kernelIds);
```

*Returns:* A kernel bundle in state `State` which contains all of the device images that are compatible with at least one of the devices in `devs`, further filtered to contain only those device images that contain at least one of the kernels with the given identifiers. These identifiers may represent kernels that are defined in the application, device built-in kernels, or a mixture of the two. Since the device images may group many kernels together, the returned bundle may contain additional kernels beyond those that are requested in `kernelIds`. The bundle's set of associated devices is `devs` (with duplicate devices removed).

Since the implementation may not represent all kernels in `bundle_state::input` or `bundle_state::object`, calling this function with one of those states may return a bundle that is missing some of the kernels in `kernelIds`. The application can test for this via `kernel_bundle::has_kernel()`.

*Throws:*

- An **exception** with the `errc::invalid` error code if any of the kernels identified by `kernelIds` are incompatible with all devices in `devs`.
- An **exception** with the `errc::invalid` error code if any of the devices in `devs` is not one of devices contained by the context `ctxt` or is not a **descendent device** of some device in `ctxt`.
- An **exception** with the `errc::invalid` error code if the `devs` vector is empty.
- An **exception** with the `errc::invalid` error code if `State` is `bundle_state::input` and any device in `devs` does not have `aspect::online_compiler`.
- An **exception** with the `errc::invalid` error code if `State` is `bundle_state::object` and any device in `devs` does not have `aspect::online_linker`.
- An **exception** with the `errc::build` error code if `State` is `bundle_state::object` or `bundle_state::executable`, if the implementation needs to perform an online compile or link, and if the online compile or link fails.

```
template <bundle_state State, typename Selector>
kernel_bundle<State> get_kernel_bundle(const context& ctxt,
                                       const std::vector<device>& devs,
                                       Selector selector);
```

*Preconditions:* The `selector` must be a unary predicate whose return value is convertible to `bool` and whose parameter is `const device_image<State>&`.

*Effects:* The predicate function `selector` is called once for every device image in the application of state `State` which is compatible with at least one of the devices in `devs`. The function's return value determines whether a device image is included in the new kernel bundle. The `selector` is called only for device images that contain kernels defined in the application, not for device images that contain device built-in kernels.

*Returns:* A kernel bundle in state `State` which contains all of the device images for which the `selector` returns `true`. The bundle's set of associated devices is `devs` (with duplicate devices removed).

*Throws:*

- An **exception** with the `errc::invalid` error code if any of the devices in `devs` is not one of devices contained by the context `ctxt` or is not a **descendent device** of some device in `ctxt`.
- An **exception** with the `errc::invalid` error code if the `devs` vector is empty.
- An **exception** with the `errc::invalid` error code if `State` is `bundle_state::input` and any device in `devs` does not have `aspect::online_compiler`.
- An **exception** with the `errc::invalid` error code if `State` is `bundle_state::object` and any device in `devs` does not have `aspect::online_linker`.

This function is intended to be used in conjunction with backend specific APIs that allow the application to choose device images based on backend specific criteria.



This function does not call the online compiler or linker to translate device images into state `State`. If the application wants to select specific device images and also compile or link them into the desired state, it can do this by calling `compile()` or `link()` and then optionally joining several bundles together with `join()`.

```
template <bundle_state State> // (1)
kernel_bundle<State> get_kernel_bundle(const context& ctxt);

template <bundle_state State> // (2)
kernel_bundle<State> get_kernel_bundle(const context& ctxt,
                                       const std::vector<kernel_id>& kernelIds);

template <bundle_state State, typename Selector> // (3)
kernel_bundle<State> get_kernel_bundle(const context& ctxt, Selector selector);
```

1. Equivalent to `get_kernel_bundle<State>(ctxt, ctxt.get_devices())`.
2. Equivalent to `get_kernel_bundle<State>(ctxt, ctxt.get_devices(), kernelIds)`.
3. Equivalent to `get_kernel_bundle<State>(ctxt, ctxt.get_devices(), selector)`.

```
template <typename KernelName, bundle_state State> // (1)
kernel_bundle<State> get_kernel_bundle(const context& ctxt);

template <typename KernelName, bundle_state State> // (2)
kernel_bundle<State> get_kernel_bundle(const context& ctxt,
                                       const std::vector<device>& devs);
```

**Preconditions:** The template parameter `KernelName` must be the **type kernel name** of a kernel that is defined in the **SYCL application**. Since lambda expressions have no standard type name, kernels defined as lambda expressions must specify a `KernelName` in their **kernel invocation command** in order to use these functions. Applications which call these functions for a `KernelName` that is not defined are ill formed, and the implementation must issue a diagnostic in this case.

1. Equivalent to `get_kernel_bundle<State>(ctxt, ctxt.get_devices(), {get_kernel_id<KernelName>()})`.
2. Equivalent to `get_kernel_bundle<State>(ctxt, devs, {get_kernel_id<KernelName>()})`.

## 4.11.8. Querying if a kernel bundle exists

Most overloads of `get_kernel_bundle()` have a matching overload of the free function `has_kernel_bundle()` which checks to see if a kernel bundle with the requested characteristics exists.

```
template <bundle_state State>
bool has_kernel_bundle(const context& ctxt, const std::vector<device>& devs);
```

Returns: **true** only if all of the following are true:

- The application defines at least one **kernel** that is compatible with at least one of the devices in **devs**, and that kernel can be represented in a device image of state **State**.
- If **State** is **bundle\_state::input**, all devices in **devs** have **aspect::online\_compiler**.
- If **State** is **bundle\_state::object**, all devices in **devs** have **aspect::online\_linker**.

Throws:

- An **exception** with the **errc::invalid** error code if any of the devices in **devs** is not one of devices contained by the context **ctxt** or is not a **descendent device** of some device in **ctxt**.
- An **exception** with the **errc::invalid** error code if the **devs** vector is empty.

```
template <bundle_state State>
bool has_kernel_bundle(const context& ctxt, const std::vector<device>& devs,
                      const std::vector<kernel_id>& kernelIds);
```

Returns: **true** only if all of the following are true:

- Each of the kernels in **kernelIds** can be represented in a device image of state **State**.
- Each of the kernels in **kernelIds** is compatible with at least one of the devices in **devs**.
- If **State** is **bundle\_state::input**, all devices in **devs** have **aspect::online\_compiler**.
- If **State** is **bundle\_state::object**, all devices in **devs** have **aspect::online\_linker**.

Throws:

- An **exception** with the **errc::invalid** error code if any of the devices in **devs** is not one of devices contained by the context **ctxt** or is not a **descendent device** of some device in **ctxt**.
- An **exception** with the **errc::invalid** error code if the **devs** vector is empty.

```
template <bundle_state State> // (1)
bool has_kernel_bundle(const context& ctxt);

template <bundle_state State> // (2)
bool has_kernel_bundle(const context& ctxt,
                      const std::vector<kernel_id>& kernelIds);
```

1. Equivalent to **has\_kernel\_bundle(ctxt, ctxt.get\_devices())**.
2. Equivalent to **has\_kernel\_bundle<State>(ctxt, ctxt.get\_devices(), kernelIds)**.

```
template <typename KernelName, bundle_state State> // (1)
bool has_kernel_bundle(const context& ctxt);

template <typename KernelName, bundle_state State> // (2)
bool has_kernel_bundle(const context& ctxt, const std::vector<device>& devs);
```

Preconditions: The template parameter **KernelName** must be the **type kernel name** of a kernel that is

defined in the [SYCL application](#). Since lambda expressions have no standard type name, kernels defined as lambda expressions must specify a `KernelName` in their [kernel invocation command](#) in order to use these functions. Applications which call these functions for a `KernelName` that is not defined are ill formed, and the implementation must issue a diagnostic in this case.

1. Equivalent to `has_kernel_bundle<State>(ctxt, {get_kernel_id<KernelName>()})`.
2. Equivalent to `has_kernel_bundle<State>(ctxt, devs, {get_kernel_id<KernelName>()})`.

### 4.11.9. Querying if a kernel is compatible with a device

The following free functions allow an application to test whether a particular kernel is compatible with a device. A kernel that is defined in the application is compatible with a device unless:

- It uses optional features which are not supported on the device, as described in [Section 5.7](#); or
- It is decorated with a `[[sycl::device_has()]]` C++ attribute that lists an aspect that is not supported by the device, as described in [Section 5.8.1](#); or
- The translation unit containing the kernel was compiled in a compilation environment that does not support the device. Each implementation defines the specific criteria for which devices are supported in its compilation environment. For example, this might be dependent on options passed to the compiler.

A device built-in kernel is only compatible with the device for which it is built-in.

```
bool is_compatible(const std::vector<kernel_id>& kernelIds, const device& dev);
```

*Returns:* `true` if all of the kernels identified by `kernelIds` are compatible with the device `dev`.

```
template <typename KernelName> bool is_compatible(const device& dev);
```

*Preconditions:* The template parameter `KernelName` must be the [type kernel name](#) of a kernel that is defined in the [SYCL application](#). Since lambda expressions have no standard type name, kernels defined as lambda expressions must specify a `KernelName` in their [kernel invocation command](#) in order to use this function. Applications which call this function for a `KernelName` that is not defined are ill formed, and the implementation must issue a diagnostic in this case.

Equivalent to `is_compatible<State>({get_kernel_id<KernelName>()}, dev)`.

### 4.11.10. Joining kernel bundles

Two or more kernel bundles of the same state may be joined together into a single composite bundle. Joining bundles together is not the same as online compiling or linking because it produces a new bundle in the same state as its inputs. Rather, joining creates the union of all the device images from the input bundles, eliminates duplicate copies of the same device image, and creates a new bundle from the result.

```
template <bundle_state State>
kernel_bundle<State> join(const std::vector<kernel_bundle<State>>& bundles);
```

*Returns:* A new kernel bundle that contains a copy of all the device images in the input `bundles` with duplicates removed. The new bundle has the same associated context and the same set of associated devices as those in `bundles`.

*Throws:*



- An **exception** with the `errc::invalid` error code if the bundles in `bundles` do not all have the same associated context or do not all have the same set of associated devices.

### 4.11.11. Online compiling and linking

If the implementation provides an online compiler or linker, a **SYCL application** can use the free functions defined in this section to transform a kernel bundle from `bundle_state::input` into a bundle of state `bundle_state::object` or to transform a bundle from `bundle_state::object` into a bundle of state `bundle_state::executable`.

An application can query whether the implementation provides an online compiler or linker by querying a device for `aspect::online_compiler` or `aspect::online_linker`.

All of the functions in this section accept a `property_list` parameter, which can affect the semantics of the compilation or linking operation. The **core SYCL specification** does not currently define any such properties, but vendors may specify these properties as an extension.

```
kernel_bundle<bundle_state::object>
compile(const kernel_bundle<bundle_state::input>& inputBundle,
        const std::vector<device>& devs, const property_list& propList = {});
```

**Effects:** The device images from `inputBundle` are translated into one or more new device images of state `bundle_state::object`, and a new kernel bundle is created to contain these new device images. The new bundle represents all of the **kernels** in `inputBundles` that are compatible with at least one of the devices in `devs`. Any remaining kernels (those that are not compatible with any of the devices `devs`) are not compiled and not represented in the new kernel bundle.

The new bundle has the same associated context as `inputBundle`, and the new bundle's set of associated devices is `devs` (with duplicate devices removed).

**Returns:** The new kernel bundle.

**Throws:**

- An **exception** with the `errc::invalid` error code if any of the devices in `devs` are not in the set of associated devices for `inputBundle` (as defined by `kernel_bundle::get_devices()`) or if the `devs` vector is empty.
- An **exception** with the `errc::build` error code if the online compile operation fails.

```
kernel_bundle<bundle_state::executable>
link(const std::vector<kernel_bundle<bundle_state::object>>& objectBundles,
      const std::vector<device>& devs, const property_list& propList = {});
```

**Effects:** Duplicate device images from `objectBundles` are eliminated as though they were joined via `join()`, then the remaining device images are translated into one or more new device images of state `bundle_state::executable`, and a new kernel bundle is created to contain these new device images. The new bundle represents all of the **kernels** in `objectBundles` that are compatible with at least one of the devices in `devs`. Any remaining kernels (those that are not compatible with any of the devices in `devs`) are not linked and not represented in the new bundle.

The new bundle has the same associated context as those in `objectBundles`, and the new bundle's set of associated devices is `devs` (with duplicate devices removed).

**Returns:** The new kernel bundle.

*Throws:*

- An **exception** with the `errc::invalid` error code if the bundles in `objectBundles` do not all have the same associated context.
- An **exception** with the `errc::invalid` error code if any of the devices in `devs` are not in the set of associated devices for any of the bundles in `objectBundles` (as defined by `kernel_bundle::get_devices()`) or if the `devs` vector is empty.
- An **exception** with the `errc::build` error code if the online link operation fails.

```
kernel_bundle<bundle_state::executable>
build(const kernel_bundle<bundle_state::input>& inputBundle,
      const std::vector<device>& devs, const property_list& propList = {});
```

*Effects:* This function performs both an online compile and link operation, translating a kernel bundle of state `bundle_state::input` into a bundle of state `bundle_state::executable`. The device images from `inputBundle` are translated into one or more new device images of state `bundle_state::executable`, and a new bundle is created to contain these new device images. The new bundle represents all of the `kernels` in `inputBundle` that are compatible with at least one of the devices in `devs`. Any remaining kernels (those that are not compatible with any of the devices `devs`) are not compiled or linked and are not represented in the new bundle.

The new bundle has the same associated context as `inputBundle`, and the new bundle's set of associated devices is `devs` (with duplicate devices removed).

*Returns:* The new kernel bundle.

*Throws:*

- An **exception** with the `errc::invalid` error code if any of the devices in `devs` are not in the set of associated devices for `inputBundle` (as defined by `kernel_bundle::get_devices()`) or if the `devs` vector is empty.
- An **exception** with the `errc::build` error code if the online compile or link operations fail.

```
kernel_bundle<bundle_state::object> // (1)
compile(const kernel_bundle<bundle_state::input>& inputBundle,
        const property_list& propList = {});

kernel_bundle<bundle_state::executable> // (2)
link(const kernel_bundle<bundle_state::object>& objectBundle,
      const std::vector<device>& devs, const property_list& propList = {});

kernel_bundle<bundle_state::executable> // (3)
link(const std::vector<kernel_bundle<bundle_state::object>>& objectBundles,
      const property_list& propList = {});

kernel_bundle<bundle_state::executable> // (4)
link(const kernel_bundle<bundle_state::object>& objectBundle,
      const property_list& propList = {});

kernel_bundle<bundle_state::executable> // (5)
build(const kernel_bundle<bundle_state::input>& inputBundle,
      const property_list& propList = {});
```

1. Equivalent to `compile(inputBundle, inputBundle.get_devices(), propList)`.



2. Equivalent to `link({objectBundle}, devs, propList)`.
3. Equivalent to `link(objectBundles, devs, propList)`, where `devs` is the intersection of associated devices in common for all bundles in `objectBundles`.
4. Equivalent to `link({objectBundle}, objectBundle.get_devices(), propList)`.
5. Equivalent to `build(inputBundle, inputBundle.get_devices(), propList)`.

#### 4.11.12. The `kernel_bundle` class

A synopsis of the `kernel_bundle` class is shown below. Additionally, this class provides the common special member functions and common member functions that are listed in [Section 4.5.2](#) in [Table 7](#) and [Table 8](#), respectively.

As with all SYCL objects that have the common reference semantics, kernel bundles are equality comparable. Two bundles of the same `bundle_state` are considered to be equal if they are associated with the same context, have the same set of associated devices, and contain the same set of device images.

There is no public default constructor for this class.

```

1 namespace sycl {
2
3 class kernel { /* ... */
4 };
5
6 template <bundle_state State> class kernel_bundle {
7 public:
8     using device_image_iterator = __unspecified__;
9
10    kernel_bundle() = delete;
11
12    bool empty() const noexcept;
13
14    backend get_backend() const noexcept;
15
16    context get_context() const noexcept;
17
18    std::vector<device> get_devices() const noexcept;
19
20    bool has_kernel(const kernel_id& kernelId) const noexcept;
21
22    bool has_kernel(const kernel_id& kernelId, const device& dev) const noexcept;
23
24    template <typename KernelName> bool has_kernel() const noexcept;
25
26    template <typename KernelName>
27    bool has_kernel(const device& dev) const noexcept;
28
29    std::vector<kernel_id> get_kernel_ids() const;
30
31    /* Available only when: (State == bundle_state::executable) */
32    kernel get_kernel(const kernel_id& kernelId) const;
33
34    /* Available only when: (State == bundle_state::executable) */
35    template <typename KernelName> kernel get_kernel() const;
36
37    bool contains_specialization_constants() const noexcept;

```

```

38
39  bool native_specialization_constant() const noexcept;
40
41  template <auto& SpecName> bool has_specialization_constant() const noexcept;
42
43  /* Available only when: (State == bundle_state::input) */
44  template <auto& SpecName>
45  void set_specialization_constant(
46      typename std::remove_reference_t<decltype(SpecName)>::value_type value);
47
48  template <auto& SpecName>
49  typename std::remove_reference_t<decltype(SpecName)>::value_type
50  get_specialization_constant() const;
51
52  device_image_iterator begin() const;
53
54  device_image_iterator end() const;
55 };
56
57 } // namespace sycl

```

#### 4.11.12.1. Queries

The following member functions provide various queries for a [kernel bundle](#).

```
bool empty() const noexcept;
```

*Returns:* **true** only if the kernel bundle contains no device images.

```
backend get_backend() const noexcept;
```

*Returns:* The backend that is associated with the kernel bundle.

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

*Returns:* The context that is associated with the kernel bundle.

```
std::vector<device> get_devices() const noexcept;
```

*Returns:* The set of devices that is associated with the kernel bundle.

```

bool has_kernel(const kernel_id& kernelId) const noexcept; // (1)
bool has_kernel(const kernel_id& kernelId,
                const device& dev) const noexcept; // (2)

```

1. *Returns:* **true** only if the kernel bundle contains the kernel identified by **kernelId**.
2. *Returns:* **true** only if the kernel bundle contains the kernel identified by **kernelId** and if that kernel is compatible with the device **dev**.

```
template <typename KernelName> bool has_kernel() const noexcept; // (1)
```

```
template <typename KernelName>
bool has_kernel(const device& dev) const noexcept; // (2)
```

*Preconditions:* The template parameter `KernelName` must be the [type kernel name](#) of a kernel that is defined in the [SYCL application](#). Since lambda expressions have no standard type name, kernels defined as lambda expressions must specify a `KernelName` in their [kernel invocation command](#) in order to use these functions. Applications which call these functions for a `KernelName` that is not defined are ill formed, and the implementation must issue a diagnostic in this case.

1. *Returns:* `true` only if the kernel bundle contains the kernel identified by `KernelName`.
2. *Returns:* `true` only if the kernel bundle contains the kernel identified by `KernelName` and if that kernel is compatible with the device `dev`.

```
std::vector<kernel_id> get_kernel_ids() const;
```

*Returns:* A vector of the identifiers for all kernels that are contained in the kernel bundle.

```
kernel get_kernel(const kernel_id& kernelId) const;
```

*Preconditions:* This member function is only available if the kernel bundle's state is `bundle_state::executable`.

*Returns:* A `kernel` object representing the kernel identified by `kernelId`, which resides in the bundle.

*Throws:*

- An `exception` with the `errc::invalid` error code if the kernel bundle does not contain the kernel identified by `kernelId`.

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

*Preconditions:* This member function is only available if the kernel bundle's state is `bundle_state::executable`. The template parameter `KernelName` must be the [type kernel name](#) of a kernel that is defined in the [SYCL application](#). Since lambda expressions have no standard type name, kernels defined as lambda expressions must specify a `KernelName` in their [kernel invocation command](#) in order to use this function. Applications which call this function for a `KernelName` that is not defined are ill formed, and the implementation must issue a diagnostic in this case.

*Returns:* A `kernel` object representing the kernel identified by `KernelName`, which resides in the bundle.

*Throws:*

- An `exception` with the `errc::invalid` error code if the kernel bundle does not contain the kernel identified by `KernelName`.

#### 4.11.12.2. Specialization constant support

The following member functions allow an application to manipulate [specialization constants](#) that are used in the device images of a [kernel bundle](#). Applications can set the value of specialization constants in a kernel bundle whose state is `bundle_state::input` and then online compile that bundle into `bundle_state::object` or `bundle_state::executable`. The value of the specialization constants then become fixed in the compiled bundle and cannot be changed. Specialization constants that have not had their values set by the time the bundle is compiled take their default values.



It is expected that many implementations will use an intermediate language representation for a bundle in state `bundle_state::input` such as SPIR-V, and the intermediate language will have native support for specialization constants. However, implementations that do not have such native support must still support specialization constants in some other way.

```
bool contains_specialization_constants() const noexcept;
```

*Returns:* `true` only if the kernel bundle contains at least one device image which uses a specialization constant.

```
bool native_specialization_constant() const noexcept;
```

*Returns:* `true` only if the kernel bundle contains at least one device image which uses a specialization constant and all specialization constants used in all of the bundle's device images are [native specialization constants](#).

```
template <auto& SpecName> bool has_specialization_constant() const noexcept;
```

*Returns:* `true` if any device image in the kernel bundle uses the specialization constant whose address is `SpecName`.

```
template <auto& SpecName>
void set_specialization_constant(
    typename std::remove_reference_t<decltype(SpecName)>::value_type value);
```

*Preconditions:* This member function is only available if the kernel bundle's state is `bundle_state::input`.

*Effects:* Sets the value of the [specialization constant](#) whose address is `SpecName` for this bundle. If the specialization constant's value was previously set in this bundle, the value is overwritten.

The new value applies to all device images in the bundle. It is allowed to set the value of a specialization constant even if no device image in the bundle uses it; doing so has no effect on the execution of kernels from that bundle.

```
template <auto& SpecName>
typename std::remove_reference_t<decltype(SpecName)>::value_type
get_specialization_constant() const;
```

*Returns:* The value of the [specialization constant](#) whose address is `SpecName` for this kernel bundle. The value returned is as follows:

- If the value of this specialization constant was previously set in this bundle, that value is returned. Otherwise,
- If this bundle is the result of compiling, linking or joining another bundle and this specialization constant was set in that other bundle prior to compiling, linking or joining; then that value is returned. Otherwise,
- The specialization constant's default value is returned.

### 4.11.12.3. Device image support

The following member type and functions allow iteration over the [device images](#) contained by the kernel bundle.

```
using device_image_iterator = __unspecified__;
```

An iterator type that satisfies the C++ requirements of [LegacyForwardIterator](#). The iterator's referenced type is `const device_image<State>`, where `State` is the same state as the containing `kernel_bundle`.

```
device_image_iterator begin() const; // (1)
device_image_iterator end() const;  // (2)
```

1. *Returns:* An iterator to the first [device image](#) contained by the kernel bundle.
2. *Returns:* An iterator to one past the last [device image](#) contained by the kernel bundle.

### 4.11.13. The `kernel` class

A synopsis of the `kernel` class is shown below. Additionally, this class provides the common special member functions and common member functions that are listed in [Section 4.5.2](#) in [Table 7](#) and [Table 8](#), respectively.

There is no public default constructor for this class.

```
1 namespace sycl {
2
3 class kernel {
4 public:
5     kernel() = delete;
6
7     backend get_backend() const noexcept;
8
9     context get_context() const;
10
11     kernel_bundle<bundle_state::executable> get_kernel_bundle() const;
12
13     template <typename Param> typename Param::return_type get_info() const;
14
15     template <typename Param>
16     typename Param::return_type get_info(const device& dev) const;
17
18     template <typename Param>
19     typename Param::return_type get_backend_info() const;
20 };
21
22 } // namespace sycl
```

#### 4.11.13.1. Queries

The following member functions provide various queries for a [kernel](#).

```
backend get_backend() const noexcept;
```

*Returns:* The backend associated with this kernel.

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

*Returns:* The context associated with this kernel.

```
kernel_bundle<bundle_state::executable> get_kernel_bundle() const;
```

*Returns:* The kernel bundle that contains this kernel.

```
template <typename Param> typename Param::return_type get_info() const;
```

*Preconditions:* The `Param` must be one of the `info::kernel` descriptors defined in Table 115, and the type alias `Param::return_type` must be defined in accordance with that table.

*Returns:* Information about the kernel that is not specific to the device on which it is invoked.

```
template <typename Param>
typename Param::return_type get_info(const device& dev) const;
```

*Preconditions:* The `Param` must be one of the `info::kernel_device_specific` descriptors defined in Table 116, and the type alias `Param::return_type` must be defined in accordance with that table.

*Returns:* Information about the kernel that applies when the kernel is invoked on the device `dev`.

*Throws:*

- An `exception` with the `errc::invalid` error code if the kernel is not compatible with device `dev` (as defined by `is_compatible()`).

```
template <typename Param> typename Param::return_type get_backend_info() const;
```

*Preconditions:* The `Param` must be one of a descriptor defined by a SYCL backend specification.

*Returns:* Backend specific information about the kernel that is not specific to the device on which it is invoked.

*Throws:*

- An `exception` with the `errc::backend_mismatch` error code if the SYCL backend that corresponds with `Param` is different from the SYCL backend that is associated with this kernel bundle.

#### 4.11.13.2. Kernel information descriptors

A `kernel` can be queried for information using the `get_info()` member function, specifying one of the info parameters in `info::kernel`. All info parameters in `info::kernel` are specified in Table 115 and the synopsis for `info::kernel` is described in Section A.5.

*Table 115. Kernel class information descriptors*

Kernel Descriptors	Return type	Description
<code>info::kernel::num_args</code>	<code>uint32_t</code>	<p>This descriptor may only be used to query a kernel that resides in a kernel bundle that was constructed using a backend specific interoperability function or to query a device built-in kernel, and the semantics of this descriptor are defined by each SYCL backend specification.</p> <p>Attempting to use this descriptor for other kernels throws an <code>exception</code> with the <code>errc::invalid</code> error code.</p>
<code>info::kernel::attributes</code>	<code>std::string</code>	Return any attributes specified on a kernel function (as defined in <a href="#">Section 5.8</a> ).

A `kernel` can also be queried for device specific information using the `get_info()` member function, specifying one of the info parameters in `info::kernel_device_specific`. All info parameters in `info::kernel_device_specific` are specified in [Table 116](#). The synopsis for `info::kernel_device_specific` is described in [Section A.5](#).

Table 116. Device-specific kernel information descriptors

Device-specific Kernel Information Descriptors	Return type	Description
<code>info::kernel_device_specific::global_work_size</code>	<code>range&lt;3&gt;</code>	<p>This descriptor may only be used if the device type is <code>device_type::custom</code> or if the kernel is a built-in kernel. The exact semantics of this descriptor are defined by each SYCL backend specification, but the intent is to return the kernel's maximum global work size.</p> <p>Attempting to use this descriptor for other devices or kernels throws an <code>exception</code> with the <code>errc::invalid</code> error code.</p>
<code>info::kernel_device_specific::work_group_size</code>	<code>size_t</code>	Returns the maximum number of work-items in a work-group that can be used to execute this kernel on the given device. This value will always be less than or equal to the value returned from <code>info::device::max_work_group_size</code> .
<code>info::kernel_device_specific::compile_work_group_size</code>	<code>range&lt;3&gt;</code>	Returns the work-group size specified by the device compiler if applicable, otherwise returns <code>{0,0,0}</code> .
<code>info::kernel_device_specific::preferred_work_group_size_multiple</code>	<code>size_t</code>	Returns a value, of which work-group size is preferred to be a multiple, for executing a kernel on a particular device. This is a performance hint. The value must be less than or equal to that returned by <code>info::kernel_device_specific::work_group_size</code> .

Device-specific Kernel Information Descriptors	Return type	Description
<code>info::kernel_device_specific::private_mem_size</code>	<code>size_t</code>	Returns the minimum amount of private memory, in bytes, used by each work-item in the kernel. This value may include any private memory needed by an implementation to execute the kernel, including that used by the language built-ins and variables declared inside the kernel in the private address space.
<code>info::kernel_device_specific::max_num_sub_groups</code>	<code>uint32_t</code>	Returns the maximum number of sub-groups for this kernel.
<code>info::kernel_device_specific::compile_num_sub_groups</code>	<code>uint32_t</code>	Returns the number of sub-groups specified by the kernel, or 0 (if not specified).
<code>info::kernel_device_specific::max_sub_group_size</code>	<code>uint32_t</code>	Returns the maximum sub-group size for this kernel.
<code>info::kernel_device_specific::compile_sub_group_size</code>	<code>uint32_t</code>	Returns the required sub-group size specified by the kernel, or 0 (if not specified).

#### 4.11.14. The `device_image` class

A synopsis of the `device_image` class is shown below. Additionally, this class provides the common special member functions and common member functions that are listed in [Section 4.5.2](#) in [Table 7](#) and [Table 8](#), respectively.

```

1 namespace sycl {
2
3 template <bundle_state State> class device_image {
4 public:
5     device_image() = delete;
6
7     bool has_kernel(const kernel_id& kernelId) const noexcept;
8
9     bool has_kernel(const kernel_id& kernelId, const device& dev) const noexcept;
10 };
11
12 } // namespace sycl

```

There is no public constructor for this class.

```

bool has_kernel(const kernel_id& kernelId) const noexcept; // (1)
bool has_kernel(const kernel_id& kernelId,
                const device& dev) const noexcept; // (2)

```

1. *Returns:* `true` only if the device image contains the kernel identified by `kernelId`.



2. *Returns:* **true** only if the device image contains the kernel identified by **kernelId** and if that kernel is compatible with the device **dev**.

## 4.11.15. Example usage

This section provides some examples showing typical use cases for kernel bundles. These examples are intended to clarify the definition of the kernel bundle interfaces, but the content of this section is non-normative.

### 4.11.15.1. Controlling the timing of online compilation

In some cases an application may want to pre-compile its kernels before submitting them to a device. This gives the application control over when the overhead of online compilation happens, rather than relying on the default behavior (which may cause the online compilation to happen at the point when the kernel is submitted to a device). The following example shows how this can be achieved.

```

1 #include <sycl/sycl.hpp>
2 using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names
3
4 int main() {
5     queue myQueue;
6     auto myContext = myQueue.get_context();
7
8     // This call to get_kernel_bundle() forces an online compilation of all the
9     // application's kernels for the device in "myContext", unless those kernels
10    // were already compiled for that device by the ahead-of-time compiler.
11    auto myBundle = get_kernel_bundle<bundle_state::executable>(myContext);
12
13    myQueue.submit([&](handler& cgh) {
14        // Calling use_kernel_bundle() causes the parallel_for() below to use the
15        // pre-compiled kernel from "myBundle".
16        cgh.use_kernel_bundle(myBundle);
17
18        cgh.parallel_for(range{1024}, ([&](item index) {
19            // kernel code
20        }));
21    });
22
23    myQueue.wait();
24 }
```

### 4.11.15.2. Specialization constants

An application can use a kernel bundle to set the values of specialization constants in several kernels before any of them are submitted for execution.

```

1 #include <sycl/sycl.hpp>
2 using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names
3
4 // Forward declare names for our two kernels.
5 class MyKernel1;
6 class MyKernel2;
7
8 extern int get_width();
9 extern int get_height();
```

```

10
11 // Declare specialization constants used in our kernels.
12 constexpr specialization_id<int> width;
13 constexpr specialization_id<int> height;
14
15 int main() {
16     queue myQueue;
17     auto myContext = myQueue.get_context();
18
19     // Get the identifiers for our kernels, then get an input kernel bundle that
20     // contains our two kernels.
21     auto kernelIds = {get_kernel_id<MyKernel1>(), get_kernel_id<MyKernel2>()};
22     auto inputBundle =
23         get_kernel_bundle<bundle_state::input>(myContext, kernelIds);
24
25     // Set the values of the specialization constants.
26     inputBundle.set_specialization_constant<width>(get_width());
27     inputBundle.set_specialization_constant<height>(get_height());
28
29     // Build the kernel bundle into an executable form. The values of the
30     // specialization constants are compiled in.
31     auto exeBundle = build(inputBundle);
32
33     myQueue.submit([&](handler& cgh) {
34         // Use the kernel bundle we built in this command group.
35         cgh.use_kernel_bundle(exeBundle);
36         cgh.parallel_for<MyKernel1>(
37             range{1024}, ([=](item index, kernel_handler kh) {
38                 // Read the value of the specialization constant.
39                 int w = kh.get_specialization_constant<width>();
40                 // ...
41             }));
42     });
43
44     myQueue.submit([&](handler& cgh) {
45         // This command group uses the same kernel bundle.
46         cgh.use_kernel_bundle(exeBundle);
47         cgh.parallel_for<MyKernel2>(
48             range{1024}, ([=](item index, kernel_handler kh) {
49                 int h = kh.get_specialization_constant<height>();
50                 // ...
51             }));
52     });
53
54     myQueue.wait();
55 }

```

#### 4.11.15.3. Kernel introspection

Applications can use kernel bundles to introspect its kernels and use that information to tune the arguments passed when invoking it.

```

1 #include <sycl/sycl.hpp>
2 using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names
3

```

```

4 class MyKernel; // Forward declare the name of our kernel.
5
6 int main() {
7     size_t N = 1024;
8     queue myQueue;
9     auto myContext = myQueue.get_context();
10    auto myDev = myQueue.get_device();
11
12    // Get an executable kernel bundle containing our kernel.
13    kernel_id kernelId = get_kernel_id<MyKernel>();
14    auto myBundle =
15        get_kernel_bundle<bundle_state::executable>(myContext, {kernelId});
16
17    // Get the kernel's maximum work-group size when running on our device.
18    kernel myKernel = myBundle.get_kernel(kernelId);
19    size_t maxWgSize =
20        myKernel.get_info<info::kernel_device_specific::work_group_size>(myDev);
21
22    // Compute a good ND-range to use for iteration in the kernel
23    // based on the maximum work-group size.
24    std::array<size_t, 11> divisors = {1024, 512, 256, 128, 64, 32,
25                                     16, 8, 4, 2, 1};
26    size_t wgSize = *std::find_if(divisors.begin(), divisors.end(),
27                                  [=](auto d) { return (d <= maxWgSize); });
28    nd_range myRange{range{N}, range{wgSize}};
29
30    myQueue.submit([&](handler& cgh) {
31        // Use the kernel bundle we queried, so we are sure the queried work-group
32        // size matches the kernel we run.
33        cgh.use_kernel_bundle(myBundle);
34        cgh.parallel_for<MyKernel>(myRange, ([=](nd_item<1> index) {
35            // kernel code
36        })));
37    });
38
39    myQueue.wait();
40 }

```

#### 4.11.15.4. Invoking a device built-in kernel

An application can use kernel bundles to invoke a device's built-in kernels.

```

1 #include <sycl/sycl.hpp>
2 using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names
3
4 int main() {
5     queue myQueue;
6     auto myContext = myQueue.get_context();
7     auto myDevice = myQueue.get_device();
8
9     const std::vector<kernel_id> builtinKernelIds =
10         myDevice.get_info<info::device::built_in_kernel_ids>();
11
12    // Get an executable kernel_bundle containing all the built-in kernels
13    // supported by the device.

```

```

14  kernel_bundle<bundle_state::executable> myBundle =
15      get_kernel_bundle(myContext, {myDevice}, builtinKernelIds);
16
17  // Retrieve a kernel object that can be used to query for more information
18  // about the built-in kernel or to submit it to a command group. We assume
19  // here that the device supports at least one built-in kernel.
20  kernel builtinKernel = myBundle.get_kernel(builtinKernelIds[0]);
21
22  // Submit the built-in kernel.
23  myQueue.submit([&](handler& cgh) {
24      // Setting the arguments depends on the backend and the exact kernel used.
25      cgh.set_args(...);
26      cgh.parallel_for(range{1024}, builtinKernel);
27  });
28
29  myQueue.wait();
30 }

```

## 4.12. Defining kernels

In SYCL, functions that are executed on a SYCL device are referred to as [SYCL kernel functions](#). A [kernel](#) containing such a [SYCL kernel function](#) is enqueued on a device queue in order to be executed on that particular device.

The return type of the [SYCL kernel function](#) is `void`, and all memory accesses between host and device are through [accessors](#) or through [USM pointers](#).

There are two ways of defining kernels: as named function objects or as lambda expressions. A backend may also provide interoperability interfaces for defining kernels.

### 4.12.1. Defining kernels as named function objects

A kernel can be defined as a named function object type. These function objects provide the same functionality as any C++ function object, with the restriction that they need to follow SYCL rules to be [device copyable](#). The kernel function can be templated via templating the kernel function object type. For details on restrictions for kernel naming, please refer to [Section 5.2](#).

The `operator()` member function must be const-qualified, and it may take different parameters depending on the data accesses defined for the specific kernel. If the `operator()` function writes to any of the member variables, the behavior is undefined.

The following example defines a [SYCL kernel function](#), *RandomFiller*, which initializes all elements of a buffer with the same random number. The random number is generated during the construction of the function object while processing the command group. The `operator()` member function of the function object receives an `item` object. This member function will be called for each work-item of the execution range. The value of the random number will be assigned to each element of the buffer. In this case, the accessor and the scalar random number are members of the function object and therefore will be arguments to the device kernel. Usual restrictions of passing arguments to kernels apply.

```

1  class RandomFiller {
2  public:
3      RandomFiller(accessor<int> ptr) : ptr_{ptr} {
4          std::random_device hwRand;
5          std::uniform_int_distribution<> r{1, 100};
6          randomNum_ = r(hwRand);

```

```

7   }
8   void operator()(item<1> item) const { ptr_[item.get_id()] = randomNum_; }
9
10  private:
11      accessor<int> ptr_;
12      int randomNum_;
13 };
14
15 void workFunction(buffer<int, 1>& b, queue& q, const range<1> r) {
16     q.submit([&](handler& cgh) {
17         accessor ptr{b, cgh};
18         RandomFiller filler{ptr};
19
20         cgh.parallel_for(r, filler);
21     });
22 }

```

### 4.12.2. Defining kernels as lambda expressions

In C++, function objects can be defined using lambda expressions. Kernels may be defined as lambda expressions in SYCL. The name of a lambda expression in SYCL may optionally be specified by passing it as a template parameter to the invoking member function, and in that case, the lambda name is a C++ typename which must be forward declarable at namespace scope. If the lambda expression relies on template arguments, then if specified, the name of the lambda expression must contain those template arguments which must also be forward declarable at namespace scope. The class used for the name of a lambda expression is only used for naming purposes and is not required to be defined. For details on restrictions for kernel naming, please refer to [Section 5.2](#).

The kernel function for the lambda expression is the lambda expression itself. The kernel lambda must use copy for all of its captures (i.e. [=]), and the lambda must not use the `mutable` specifier.

```

1 // Explicit kernel names can be optionally forward declared at namespace scope
2 class MyKernel;
3
4 myQueue.submit([&](handler& h) {
5     // Explicitly name kernel with previously forward declared type
6     h.single_task<MyKernel>([=] {
7         // [kernel code]
8     });
9 });
10
11 // Explicitly name kernel without forward declaring type at
12 // namespace scope. Must still be forward declarable at
13 // namespace scope, even if not declared at that scope
14 myQueue.submit([&](handler& h) {
15     h.single_task<class MyOtherKernel>([=] {
16         // [kernel code]
17     });
18 });

```

Explicit lambda naming is shown in the following code example, including an illegal case that uses a class within the kernel name which is not forward declarable (`std::complex`).

```

1 // Explicit kernel names can be optionally forward declared at namespace scope

```

```

2  class MyForwardDeclName;
3
4  template <typename T>
5  class MyTemplatedKernelName;
6
7  // Define and launch templated kernel
8  template <typename T>
9  void templatedFunction() {
10     queue myQueue;
11
12     // Launch A: No explicit kernel name
13     myQueue.submit([&](handler& h) {
14         h.single_task( [= ] {
15             // [kernel code that depends on type T]
16         });
17     });
18
19     // Launch B: Name the kernel when invoking (this is optional)
20     myQueue.submit([&](handler& h) {
21         h.single_task<MyTemplatedKernelName<T>>( [= ] {
22             // The provided kernel name (MyTemplatedKernelName<T>) depends on T
23             // because the kernel does. T must also be forward declarable at
24             // namespace scope.
25
26             // [kernel code that depends on type T]
27         });
28     });
29 }
30
31 int main() {
32     queue myQueue;
33
34     myQueue.submit([&](handler& h) {
35         // Declare MyKernel within this kernel invocation. Legal because
36         // forward declaration at namespace scope is optional
37         h.single_task<class MyKernel>( [= ] {
38             // [kernel code]
39         });
40     });
41
42     myQueue.submit([&](handler& h) {
43         // Use kernel name that was forward declared at namespace scope
44         h.single_task<MyForwardDeclName>( [= ] {
45             // [kernel code]
46         });
47     });
48
49     templatedFunction<int>(); // OK
50
51     templatedFunction<std::complex<float>>>(); // Launch A is OK, Launch B illegal
52     // because std::complex is not forward declarable according to C++, and was
53     // used in an explicit kernel name which must be forward declarable.
54 }

```

### 4.12.3. `is_device_copyable` type trait

```
namespace sycl {
    template<typename T>
    struct is_device_copyable;

    template<typename T>
    inline constexpr bool is_device_copyable_v = is_device_copyable<T>::value;
};
```

`is_device_copyable` is a user specializable class template to indicate that a type `T` is [device copyable](#).

- `is_device_copyable` must meet the Cpp17UnaryTrait requirements.
- If `is_device_copyable` is specialized such that `is_device_copyable_v<T> == true` on a `T` that does not satisfy all the requirements of a device copyable type, the results are unspecified.

If the application defines a type `UDT` that satisfies the requirements of a [device copyable](#) type (as defined in [Section 3.13.1](#)) but the type is not implicitly device copyable as defined in that section, then the application must provide a specialization of `is_device_copyable` that derives from `std::true_type` in order to use that type in a context that requires a device copyable type. Such a specialization can be declared like this:

```
template<>
struct sycl::is_device_copyable<UDT> : std::true_type {};
```

It is legal to provide this specialization even if the implementation does not define `SYCL_DEVICE_COPYABLE` to `1`, but the type cannot be used as a device copyable type in that case and the specialization is ignored.

### 4.12.4. Rules for parameter passing to kernels

A SYCL application passes parameters to a kernel in different ways depending on whether the kernel is a named function object or a lambda expression. If the kernel is a named function object, the `operator()` member function (or other member functions that it calls) may reference member variables inside the same named function object. Any such member variables become parameters to the kernel. If the kernel is a lambda expression, any variables captured by the lambda become parameters to the kernel.

Regardless of how the parameter is passed, the following rules define the allowable types for a kernel parameter:

- Any [device copyable](#) type is a legal parameter type.
- The following SYCL types are legal parameter types:
  - `accessor` when templated with `target::device`;
  - `accessor` when templated with any of the deprecated parameters: `target::global_buffer`, `target::constant_buffer`, or `target::local`;
  - `local_accessor`;
  - `unsampled_image_accessor` when templated with `image_target::device`;
  - `sampled_image_accessor` when templated with `image_target::device`;
  - `stream`;
  - `id`;
  - `range`;

- `marray<T, NumElements>` when `T` is [device copyable](#);
- `vec<T, NumElements>`.
- An array of element types `T` is a legal parameter type if `T` is a legal parameter type.
- A class type `S` with a non-static member variable of type `T` is a legal parameter type if `T` is a legal parameter type and if `S` would otherwise be a legal parameter type aside from this member variable.
- A class type `S` with a non-virtual base class of type `T` is a legal parameter type if `T` is a legal parameter type and if `S` would otherwise be a legal parameter type aside from this base class.



Pointer types are trivially copyable, so they may be passed as kernel parameters. However, only the pointer value itself is passed to the kernel. Dereferencing the pointer on the kernel results in undefined behavior unless the pointer points to an address within a [USM](#) memory region that is accessible on the device.

Reference types are not trivially copyable, so they may not be passed as kernel parameters.



The `reducer` class is a special type of kernel parameter which is passed to a kernel in a different way. [Section 4.9.2](#) describes how this parameter type is used.

## 4.13. Error handling

### 4.13.1. Error handling rules

Error handling in a SYCL application (host code) uses C++ exceptions. If an error occurs, it will be thrown by the API function call and may be caught by the user through standard C++ exception handling mechanisms.

SYCL applications are asynchronous in the sense that host and device code executions are decoupled from one another except at specific points. For example, device code executions often begin when dependencies in the SYCL task graph are satisfied, which occurs asynchronously from host code execution. As a result of this the errors that occur on a device cannot be thrown directly from a host API call, because the call enqueueing a device action has typically already returned by the time that the error occurs. Such errors are not detected until the error-causing task executes or tries to execute, and we refer to these as [asynchronous errors](#).

#### 4.13.1.1. Asynchronous error handler

The queue and context classes can optionally take an asynchronous handler object [async\\_handler](#) on construction, which is a callable such as a function class or lambda, with an `exception_list` as a parameter. Invocation of an [async\\_handler](#) may be triggered by the queue member functions `queue::wait_and_throw()` or `queue::throw_asynchronous()`, by the event member function `event::wait_and_throw()`, or automatically on destruction of a queue or context that contains unconsumed asynchronous errors. When invoked, an [async\\_handler](#) is called and receives an `exception_list` argument containing a list of exception objects representing any unconsumed [asynchronous errors](#) associated with the queue or context.

When an [asynchronous error](#) instance has been passed to an [async\\_handler](#), then that instance of the error has been consumed for handling and is not reported on any subsequent invocations of the [async\\_handler](#).

The [async\\_handler](#) may be a named function object type, a lambda expression or a `std::function`. The `exception_list` object passed to the [async\\_handler](#) is constructed by the [SYCL runtime](#).



#### 4.13.1.2. Behavior without an async handler

If an asynchronous error occurs in a queue or context that has no user-supplied asynchronous error handler object `async_handler`, then an implementation-defined default `async_handler` is called to handle the error in the same situations that a user-supplied `async_handler` would be, as defined in [Section 4.13.1.1](#). The default `async_handler` must in some way report all errors passed to it, when possible, and must then invoke `std::terminate` or equivalent.

#### 4.13.1.3. Priorities of async handlers

If the SYCL runtime can associate an `asynchronous error` with a specific queue, then:

- If the queue was constructed with an `async_handler`, that handler is invoked to handle the error.
- Otherwise if the context enclosed by the queue was constructed with an `async_handler`, that handler is invoked to handle the error.
- Otherwise when no handler was passed to either queue or context on construction, then a default handler is invoked to handle the error, as described by [Section 4.13.1.2](#).
- All handler invocations in this list occur at times as defined by [Section 4.13.1.1](#).

If the SYCL runtime cannot associate an `asynchronous error` with a specific queue, then:

- If the context in which the error occurred was constructed with an `async_handler`, then that handler is invoked to handle the error.
- Otherwise when no handler was passed to the associated context on construction, then a default handler is invoked to handle the error, as described by [Section 4.13.1.2](#).
- All handler invocations in this list occur at times as defined by [Section 4.13.1.1](#).

#### 4.13.1.4. Asynchronous errors with a secondary queue

If an `asynchronous error` occurs when running or enqueueing a command group which has a secondary queue specified, then the command group may be enqueued to the secondary queue instead of the primary queue. The error handling in this case is also configured using the `async_handler` provided for both queues. If there is no `async_handler` given on any of the queues, then the asynchronous error handling proceeds through the contexts associated with the queues, and if they were also constructed without `async_handlers`, then the default handler will be used. If the primary queue fails and there is an `async_handler` given at this queue's construction, which populates the `exception_list` parameter, then any errors will be added and can be thrown whenever the user chooses to handle those exceptions. Since there were errors on the primary queue and a secondary queue was given, then the execution of the kernel is re-scheduled to the secondary queue and any error reporting for the kernel execution on that queue is done through that queue, in the same way as described above. The secondary queue may fail as well, and the errors will be thrown if there is an `async_handler` and either `wait_and_throw()` or `throw()` are called on that queue. If no `async_handler` was specified, then the one associated with the queue's context will be used and if the context was also constructed without an `async_handler`, then the default handler will be used. The `command group function object` event returned by that function will be relevant to the queue where the kernel has been enqueued.

Below is an example of catching a SYCL `exception` and printing out the error message.

```
1 void catch_any_errors(sycl::context const& ctx) {
2     try {
3         do_something_to_invoke_error(ctx);
4     } catch (sycl::exception const& e) {
5         std::cerr << e.what();
6     }
```

```
7 }
```

Below is an example of catching a SYCL **exception** with the **errc::invalid** error code and printing out the error message.

```
1 void catch_invalid_errors(sycl::context const& ctx) {
2     try {
3         do_something_to_invoke_error(ctx);
4     } catch (sycl::exception const& e) {
5         if (e.code() == sycl::errc::invalid) {
6             std::cerr << "Invalid error: " << e.what();
7         } else {
8             throw;
9         }
10    }
11 }
```

## 4.13.2. Exception class interface

```
1 namespace sycl {
2
3 using async_handler = std::function<void(sycl::exception_list)>;
4
5 class exception : public virtual std::exception {
6 public:
7     exception(std::error_code ec, const std::string& what_arg);
8     exception(std::error_code ec, const char* what_arg);
9     exception(std::error_code ec);
10    exception(int ev, const std::error_category& ec,
11              const std::string& what_arg);
12    exception(int ev, const std::error_category& ec, const char* what_arg);
13    exception(int ev, const std::error_category& ec);
14
15    exception(context ctx, std::error_code ec, const std::string& what_arg);
16    exception(context ctx, std::error_code ec, const char* what_arg);
17    exception(context ctx, std::error_code ec);
18    exception(context ctx, int ev, const std::error_category& ec,
19              const std::string& what_arg);
20    exception(context ctx, int ev, const std::error_category& ec,
21              const char* what_arg);
22    exception(context ctx, int ev, const std::error_category& ec);
23
24    const std::error_code& code() const noexcept;
25    const std::error_category& category() const noexcept;
26
27    const char* what() const;
28
29    bool has_context() const noexcept;
30    context get_context() const;
31 };
32
33 class exception_list {
34     // Used as a container for a list of asynchronous exceptions
35 public:
```

```

36 using value_type = std::exception_ptr;
37 using reference = value_type&
38 using const_reference = const value_type&
39 using size_type = std::size_t;
40 using iterator = /*unspecified*/;
41 using const_iterator = /*unspecified*/;
42
43 size_type size() const;
44 iterator begin() const; // first asynchronous exception
45 iterator end() const;   // refer to past-the-end last asynchronous exception
46 };
47
48 enum class errc : /* unspecified */ {
49     success = 0,
50     runtime,
51     kernel,
52     accessor,
53     nd_range,
54     event,
55     kernel_argument,
56     build,
57     invalid,
58     memory_allocation,
59     platform,
60     profiling,
61     feature_not_supported,
62     kernel_not_supported,
63     backend_mismatch
64 };
65
66 std::error_code make_error_code(errc e) noexcept;
67
68 const std::error_category& sycl_category() noexcept;
69
70 } // namespace sycl
71
72 namespace std {
73
74 template <> struct is_error_code_enum</* see below */> : true_type {};
75
76 } // namespace std

```

The SYCL `exception_list` class is also available in order to provide a list of synchronous and asynchronous exceptions.

Errors can occur both in the SYCL library and SYCL host side, or may come directly from a [SYCL backend](#). The member functions on these exceptions provide the corresponding information. [SYCL backends](#) can provide additional exception class objects as long as they derive from `sycl::exception` object, or any of its derived classes.

A specialization of `std::is_error_code_enum` must be defined for `sycl::errc` that inherits from `std::true_type`.

Table 117. Member functions of the SYCL `exception` class

Member function	Description
<code>exception(std::error_code ec, const std::string&amp; what_arg)</code>	Constructs an <b>exception</b> . The string returned by <code>what()</code> is guaranteed to contain <code>what_arg</code> as a substring.
<code>exception(std::error_code ec, const char* what_arg)</code>	Constructs an <b>exception</b> . The string returned by <code>what()</code> is guaranteed to contain <code>what_arg</code> as a substring.
<code>exception(std::error_code ec)</code>	Constructs an <b>exception</b> .
<code>exception(int ev, const std::error_category&amp; ecac, const std::string&amp; what_arg)</code>	Constructs an <b>exception</b> with the error code <code>ev</code> and the underlying error category <code>ecac</code> . The string returned by <code>what()</code> is guaranteed to contain <code>what_arg</code> as a substring.
<code>exception(int ev, const std::error_category&amp; ecac, const char* what_arg)</code>	Constructs an <b>exception</b> with the error code <code>ev</code> and the underlying error category <code>ecac</code> . The string returned by <code>what()</code> is guaranteed to contain <code>what_arg</code> as a substring.
<code>exception(int ev, const std::error_category&amp; ecac)</code>	Constructs an <b>exception</b> with the error code <code>ev</code> and the underlying error category <code>ecac</code> .
<code>exception(context ctx, std::error_code ec, const std::string&amp; what_arg)</code>	Constructs an <b>exception</b> with an associated SYCL context <code>ctx</code> . The string returned by <code>what()</code> is guaranteed to contain <code>what_arg</code> as a substring.
<code>exception(context ctx, std::error_code ec, const char* what_arg)</code>	Constructs an <b>exception</b> with an associated SYCL context <code>ctx</code> . The string returned by <code>what()</code> is guaranteed to contain <code>what_arg</code> as a substring.
<code>exception(context ctx, std::error_code ec)</code>	Constructs an <b>exception</b> with an associated SYCL context <code>ctx</code> .
<code>exception(context ctx, int ev, const std::error_category&amp; ecac, const std::string&amp; what_arg)</code>	Constructs an <b>exception</b> with an associated SYCL context <code>ctx</code> , the error code <code>ev</code> and the underlying error category <code>ecac</code> . The string returned by <code>what()</code> is guaranteed to contain <code>what_arg</code> as a substring.
<code>exception(context ctx, int ev, const std::error_category&amp; ecac, const char* what_arg)</code>	Constructs an <b>exception</b> with an associated SYCL context <code>ctx</code> , the error code <code>ev</code> and the underlying error category <code>ecac</code> . The string returned by <code>what()</code> is guaranteed to contain <code>what_arg</code> as a substring.

Member function	Description
<code>exception(context ctx, int ev, const std::error_category&amp;ecat)</code>	Constructs an <b>exception</b> with an associated SYCL context <b>ctx</b> , the error code <b>ev</b> and the underlying error category <b>ecat</b> .
<code>const std::error_code&amp; code() const noexcept</code>	Returns the error code stored inside the exception.
<code>const std::error_category&amp; category() const noexcept</code>	Returns the error category of the error code stored inside the exception.
<code>const char* what() const</code>	Returns an implementation-defined non-null constant C-style string that describes the error that triggered the exception.
<code>bool has_context() const noexcept</code>	Returns <b>true</b> if this SYCL <b>exception</b> has an associated SYCL <b>context</b> and <b>false</b> if it does not.
<code>context get_context() const</code>	Returns the SYCL <b>context</b> that is associated with this SYCL <b>exception</b> if one is available. Must throw an <b>exception</b> with the <b>errc::invalid</b> error code if this SYCL <b>exception</b> does not have a SYCL <b>context</b> .

Table 118. Member functions of the **exception\_list**

Member function	Description
<code>size_t size() const</code>	Returns the size of the list
<code>iterator begin() const</code>	Returns an iterator to the beginning of the list of asynchronous exceptions.
<code>iterator end() const</code>	Returns an iterator to the end of the list of asynchronous exceptions.

Table 119. Values of the SYCL **errc** enum

Standard SYCL Error Codes	Description
success	The implementation never throws an exception with this error code, but it is defined to ensure that no other error code has the value zero. An application can construct an <b>std::error_code</b> with this code to indicate "not an error".
runtime	Generic runtime error.

Standard SYCL Error Codes	Description
<b>kernel</b>	Error that occurred before or while enqueueing the SYCL kernel.
<b>nd_range</b>	Error regarding the SYCL <b>nd_range</b> specified for the SYCL kernel
<b>accessor</b>	Error regarding the SYCL <b>accessor</b> objects defined.
<b>event</b>	Error regarding associated SYCL <b>event</b> objects.
kernel_argument	The application has passed an invalid argument to a <a href="#">SYCL kernel function</a> . This includes captured variables if the <a href="#">SYCL kernel function</a> is a lambda expression.
<b>build</b>	Error from an online compile or link operation when compiling, linking, or building a kernel bundle for a device.
invalid	A catchall error which is used when the application passes an invalid value as a parameter to a SYCL API function or calls a SYCL API function in some invalid way.
memory_allocation	Error on memory allocation on the SYCL device for a SYCL kernel.
<b>platform</b>	The SYCL platform will trigger this exception on error.
profiling	The <a href="#">SYCL runtime</a> will trigger this error if there is an error when profiling info is enabled.
feature_not_supported	Exception thrown when host code uses an optional feature that is not supported by a device.
kernel_not_supported	Exception thrown when a kernel uses an optional feature that is not supported on the device to which it is enqueued. This exception is also thrown if a <a href="#">command group</a> is bound to a <a href="#">kernel bundle</a> , and the bundle does not contain the kernel invoked by the command group.

Standard SYCL Error Codes	Description
backend_mismatch	The application has called a back-end interoperability function with mismatched backend information. For example, requesting information specific to backend A from a SYCL object that comes from backend B causes this error.

Table 120. SYCL error code helper functions

SYCL Error Code Helpers	Description
<code>const std::error_category&amp; <b>sycl_category</b>() noexcept;</code>	Obtains a reference to the static error category object for SYCL errors. This object overrides the virtual function <code>error_category::name()</code> to return a pointer to the string "sycl". When the implementation throws an <code>sycl::exception</code> object <code>ex</code> with this category, the error code value contained by the exception ( <code>ex.code().value()</code> ) is one of the enumerated values in <code>sycl::errc</code> .
<code>std::error_code <b>make_error_code</b>(errc e) noexcept;</code>	Constructs an error code using <code>e</code> and <code>sycl_category()</code> .

## 4.14. Data types

SYCL as a C++ programming model supports the C++ core language data types, and it also provides the ability for all SYCL applications to be executed on SYCL compatible devices. The scalar and vector data types that are supported by the SYCL system are defined below. More details about the SYCL device compiler support for fundamental and backend interoperability types are found in [Section 5.5](#).

### 4.14.1. Scalar data types

The fundamental C++ data types which are supported in SYCL are described in [Table 153](#). Note these types are fundamental and therefore do not exist within the `sycl` namespace.

Additional scalar data types which are supported by SYCL within the `sycl` namespace are described in [Table 121](#).

Table 121. Additional scalar data types supported by SYCL

Scalar data type	Description
byte	An alias to <code>std::uint8_t</code> . This is deprecated in SYCL 2020 since C++17 <code>std::byte</code> can be used instead.

Scalar data type	Description
<code>half</code>	A 16-bit floating-point. The half data type must conform to the IEEE 754-2008 half precision storage format. This type is only supported on devices that have <code>aspect::fp16</code> . <code>std::numeric_limits</code> must be specialized for the half data type.

## 4.14.2. Vector types

SYCL provides a cross-platform class template that works efficiently on SYCL devices as well as in host C++ code. This type allows sharing of vectors between the host and its SYCL devices. The vector supports member functions that allow construction of a new vector from a swizzled set of component elements.

The `vec` class is templated on its number of elements and its element type. The number of elements parameter, `NumElements`, must be one of: 1, 2, 3, 4, 8 or 16. Any other value shall produce a compilation failure. The element type parameter, `DataT`, must be the cv-unqualified version of one of the following: one of the built-in scalar data types listed in [Section 5.5](#), `half`, or `sycl::byte`.

An instance of the SYCL `vec` class template can be implicitly converted to an instance of the data type when the number of elements is 1 in order to allow single element vectors and scalars to be convertible with each other.

### 4.14.2.1. Vec interface

The constructors, member functions and non-member functions of the SYCL `vec` class template are listed in [Table 122](#), [Table 123](#) and [Table 124](#) respectively.

```

1 namespace sycl {
2
3 enum class rounding_mode : /* unspecified */ { automatic, rte, rtz, rtp, rtn };
4
5 struct elem {
6     static constexpr int x = 0;
7     static constexpr int y = 1;
8     static constexpr int z = 2;
9     static constexpr int w = 3;
10    static constexpr int r = 0;
11    static constexpr int g = 1;
12    static constexpr int b = 2;
13    static constexpr int a = 3;
14    static constexpr int s0 = 0;
15    static constexpr int s1 = 1;
16    static constexpr int s2 = 2;
17    static constexpr int s3 = 3;
18    static constexpr int s4 = 4;
19    static constexpr int s5 = 5;
20    static constexpr int s6 = 6;
21    static constexpr int s7 = 7;
22    static constexpr int s8 = 8;
23    static constexpr int s9 = 9;
24    static constexpr int sA = 10;
25    static constexpr int sB = 11;
26    static constexpr int sC = 12;

```



```

27 static constexpr int sD = 13;
28 static constexpr int sE = 14;
29 static constexpr int sF = 15;
30 };
31
32 template <typename DataT, int NumElements> class vec {
33 public:
34     using element_type = DataT;
35     using value_type = DataT;
36
37     vec();
38
39     // Available only when: NumElements > 1
40     explicit constexpr vec(const DataT& arg);
41
42     // Available only when: NumElements == 1
43     constexpr vec(const DataT& arg);
44
45     template <typename... ArgTN> constexpr vec(const ArgTN&... args);
46
47     constexpr vec(const vec<DataT, NumElements>& rhs);
48
49     // Available only when: NumElements == 1
50     operator DataT() const;
51
52     // Available only when: NumElements == 1 and T is explicitly convertible to DataT
53     template<typename T>
54     explicit operator T() const;
55
56     static constexpr size_t byte_size() noexcept;
57
58     static constexpr size_t size() noexcept;
59
60     // Deprecated
61     size_t get_size() const;
62
63     // Deprecated
64     size_t get_count() const;
65
66     template <typename ConvertT,
67             rounding_mode RoundingMode = rounding_mode::automatic>
68     vec<ConvertT, NumElements> convert() const;
69
70     template <typename AsT> AsT as() const;
71
72     // Available on when the number of swizzleIndexes template parameters is
73     // 1, 2, 3, 4, 8, or 16.
74     // Available only when each of the swizzleIndexes template parameters is
75     // greater or equal to 0 and less than NumElements.
76     template <int... swizzleIndexes> __writeable_swizzle__ swizzle();
77     template <int... swizzleIndexes> __const_swizzle__ swizzle() const;
78
79     // Available only when NumElements <= 4.
80     // XYZW_ACCESS is: x, y, z, w, subject to NumElements.
81     DataT& XYZW_ACCESS();
82     const DataT& XYZW_ACCESS() const;

```

```

83
84 // Available only NumElements == 4.
85 // RGBA_ACCESS is: r, g, b, a.
86 DataT& RGBA_ACCESS();
87 const DataT& RGBA_ACCESS() const;
88
89 // INDEX_ACCESS is: s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, sA, sB, sC, sD,
90 // sE, sF, subject to NumElements.
91 DataT& INDEX_ACCESS();
92 const DataT& INDEX_ACCESS() const;
93
94 #ifdef SYCL_SIMPLE_SWIZZLES
95 // Available only when NumElements <= 4.
96 // XYZW_SWIZZLE is all permutations with repetition of: x, y, z, w, subject to
97 // NumElements.
98 __writeable_swizzle__ XYZW_SWIZZLE();
99 __const_swizzle__ XYZW_SWIZZLE() const;
100
101 // Available only when NumElements == 4.
102 // RGBA_SWIZZLE is all permutations with repetition of: r, g, b, a.
103 __writeable_swizzle__ RGBA_SWIZZLE();
104 __const_swizzle__ RGBA_SWIZZLE() const;
105
106 #endif // #ifdef SYCL_SIMPLE_SWIZZLES
107
108 // Available only when: NumElements > 1.
109 __writeable_swizzle__ lo();
110 __const_swizzle__ lo() const;
111
112 __writeable_swizzle__ hi();
113 __const_swizzle__ hi() const;
114
115 __writeable_swizzle__ odd();
116 __const_swizzle__ odd() const;
117
118 __writeable_swizzle__ even();
119 __const_swizzle__ even() const;
120
121 // load and store member functions
122 template <access::address_space AddressSpace, access::decorated IsDecorated>
123 void load(size_t offset,
124          multi_ptr<const DataT, AddressSpace, IsDecorated> ptr);
125
126 void load(size_t offset, const DataT* ptr);
127
128 template <access::address_space AddressSpace, access::decorated IsDecorated>
129 void store(size_t offset,
130           multi_ptr<DataT, AddressSpace, IsDecorated> ptr) const;
131
132 void store(size_t offset, DataT* ptr) const;
133
134 // subscript operator
135 DataT& operator[](int index);
136 const DataT& operator[](int index) const;
137
138 vec& operator=(const vec& rhs);

```

```

139
140 // Available only when: T is convertible to DataT
141 template<typename T>
142 vec& operator=(const T& rhs);
143
144 // OP is: +, -, *, /, %
145 //
146 // Available only when: T is convertible to DataT
147 // If OP is %, available only when: DataT != float && DataT != double && DataT != half
148 friend vec operatorOP(const vec& lhs, const vec& rhs);
149
150 template<typename T>
151 friend vec operatorOP(const vec& lhs, const T& rhs);
152
153 template<typename T>
154 friend vec operatorOP(const T& lhs, const vec& rhs);
155
156 // OP is: +=, -=, *=, /=, %=
157 //
158 // Available only when: T is convertible to DataT
159 // If OP is %=, available only when: DataT != float && DataT != double && DataT != half
160 friend vec& operatorOP(vec& lhs, const vec& rhs);
161
162 template<typename T>
163 friend vec& operatorOP(vec& lhs, const T& rhs);
164
165 // OP is prefix ++, --
166 //
167 // Available only when: DataT != bool
168 friend vec& operatorOP(vec& rhs);
169
170 // OP is postfix ++, --
171 //
172 // Available only when: DataT != bool
173 friend vec operatorOP(vec& lhs, int);
174
175 // OP is unary +, -
176 friend vec operatorOP(const vec& rhs);
177
178 // OP is: &, |, ^
179 //
180 // Available only when: T is convertible to DataT
181 // Available only when: DataT != float && DataT != double && DataT != half
182 friend vec operatorOP(const vec& lhs, const vec& rhs);
183
184 template<typename T>
185 friend vec operatorOP(const vec& lhs, const T& rhs);
186
187 template<typename T>
188 friend vec operatorOP(const T& lhs, const vec& rhs);
189
190 // OP is: &=, |=, ^=
191 //
192 // Available only when: T is convertible to DataT
193 // Available only when: DataT != float && DataT != double && DataT != half
194 friend vec& operatorOP(vec& lhs, const vec& rhs);

```

```

195
196 template<typename T>
197 friend vec& operatorOP(vec& lhs, const T& rhs);
198
199 // OP is: <<, >>
200 //
201 // Available only when: T is convertible to DataT
202 // Available only when: DataT != float && DataT != double && DataT != half
203 friend vec operatorOP(const vec& lhs, const vec& rhs);
204
205 template<typename T>
206 friend vec operatorOP(const vec& lhs, const T& rhs);
207
208 template<typename T>
209 friend vec operatorOP(const T& lhs, const vec& rhs);
210
211 // OP is: <=, >=
212 //
213 // Available only when: T is convertible to DataT
214 // Available only when: DataT != float && DataT != double && DataT != half
215 friend vec& operatorOP(vec& lhs, const vec& rhs);
216
217 template<typename T>
218 friend vec& operatorOP(vec& lhs, const T& rhs);
219
220 // OP is: &&, ||
221 //
222 // Available only when: T is convertible to DataT
223 friend vec<RET, NumElements> operatorOP(const vec& lhs, const vec& rhs);
224
225 template<typename T>
226 friend vec<RET, NumElements> operatorOP(const vec& lhs, const T& rhs);
227
228 template<typename T>
229 friend vec<RET, NumElements> operatorOP(const T& lhs, const vec& rhs);
230
231 // OP is: ==, !=, <, >, <=, >=
232 //
233 // Available only when: T is convertible to DataT
234 friend vec<RET, NumElements> operatorOP(const vec& lhs, const vec& rhs);
235
236 template<typename T>
237 friend vec<RET, NumElements> operatorOP(const vec& lhs, const T& rhs);
238
239 template<typename T>
240 friend vec<RET, NumElements> operatorOP(const T& lhs, const vec& rhs);
241
242 // Available only when: DataT != float && DataT != double && DataT != half
243 friend vec operator~(const vec& v);
244
245 friend vec<RET, NumElements> operator!(const vec& v);
246 };
247
248 // Deduction guides
249 // Available only when: (std::is_same_v<T, U> && ...)
250 template <class T, class... U> vec(T, U...) -> vec<T, sizeof...(U) + 1>;

```

```

251
252 } // namespace sycl

```

Table 122. Constructors of the SYCL `vec` class template

Constructor	Description
<code>vec()</code>	Default construct a vector with element type <code>DataT</code> and with <code>NumElements</code> dimensions by default construction of each of its elements.
<code>explicit constexpr vec(const DataT&amp; arg)</code>	<p><i>Constraints:</i> <code>NumElements</code> is greater than 1.</p> <p>Constructs a <code>vec</code> object by assigning each element to <code>arg</code>.</p>
<code>constexpr vec(const DataT&amp; arg)</code>	<p><i>Constraints:</i> <code>NumElements</code> is equal to 1.</p> <p>Constructs a 1-element <code>vec</code> object by assigning the element to <code>arg</code>.</p>
<code>template &lt;typename... ArgTN&gt; constexpr vec(const ArgTN&amp;... args)</code>	<p><i>Constraints:</i></p> <ul style="list-style-type: none"> <li>The total number of elements from all parameters is greater than 1 and sums to <code>NumElements</code>, and</li> <li>Each type <code>ArgTN</code> is one of the following: <ul style="list-style-type: none"> <li>A type that is implicitly convertible to <code>DataT</code>, or</li> <li>A <code>vec</code> type whose element type is <code>DataT</code>, or</li> <li>A <code>__writeable_swizzle__</code> type whose element type is <code>DataT</code>, or</li> <li>A <code>__const_swizzle__</code> type whose element type is <code>DataT</code>.</li> </ul> </li> </ul> <p>Constructs a <code>vec</code> object from a combination of scalar and vector parameters.</p>
<code>constexpr vec(const vec&lt;DataT, NumElements&gt;&amp; rhs)</code>	Construct a vector of element type <code>DataT</code> and number of elements <code>NumElements</code> by copy from another similar vector.

Table 123. Member functions for the SYCL `vec` class template

Member function	Description
<code>operator DataT() const</code>	<p>Available only when: <code>NumElements == 1</code>.</p> <p>Converts this SYCL <code>vec</code> instance to an instance of <code>DataT</code> with the value of the single element in this SYCL <code>vec</code> instance.</p> <p>The SYCL <code>vec</code> instance shall be implicitly convertible to the same data types, to which <code>DataT</code> is implicitly convertible. Note that conversion operator shall not be templated to allow standard conversion sequence for implicit conversion.</p>
<pre>template&lt;typename T&gt; explicit operator T() const</pre>	<p><i>Constraints:</i></p> <ul style="list-style-type: none"> <li>• <code>NumElements</code> is equal to 1, and</li> <li>• <code>DataT</code> can be explicitly converted to <code>T</code> via <code>static_cast&lt;T&gt;</code>, and</li> <li>• <code>T</code> is not <code>DataT</code>.</li> </ul> <p>Returns the value of the vector's element converted to <code>T</code>.</p>
<code>static constexpr size_t size() noexcept</code>	Returns the number of elements of this SYCL <code>vec</code> .
<code>size_t get_count() const</code>	Returns the same value as <code>size()</code> . Deprecated.
<code>static constexpr size_t byte_size() noexcept</code>	<p>Returns the size of this SYCL <code>vec</code> in bytes.</p> <p>3-element vector size matches 4-element vector size to provide interoperability with OpenCL vector types. The same rule applies to vector alignment as described in <a href="#">Section 4.14.2.6</a>.</p>
<code>size_t get_size() const</code>	Returns the same value as <code>byte_size()</code> . Deprecated.
<pre>template &lt;typename ConvertT,           rounding_mode RoundingMode = rounding_mode ::automatic&gt; vec&lt;ConvertT, NumElements&gt; convert() const</pre>	<p>Converts this SYCL <code>vec</code> to a SYCL <code>vec</code> of a different element type specified by <code>ConvertT</code> using the rounding mode specified by <code>RoundingMode</code>. The new SYCL <code>vec</code> type must have the same number of elements as this SYCL <code>vec</code>. The different rounding modes are described in <a href="#">Table 125</a>.</p>

Member function	Description
<pre>template &lt;typename asT&gt; asT as() const</pre>	<p>Bitwise reinterprets this SYCL <b>vec</b> as a SYCL <b>vec</b> of a different element type and number of elements specified by <b>asT</b>. The new SYCL <b>vec</b> type must have the same storage size in bytes as this SYCL <b>vec</b>, and the size of the elements in the new SYCL <b>vec</b> (<math>\text{NumElements} * \text{sizeof}(\text{DataT})</math>) must be the same as the size of the elements in this SYCL <b>vec</b>.</p>
<pre>template &lt;int... swizzleIndexes&gt; __writeable_swizzle__ swizzle() template &lt;int... swizzleIndexes&gt; __const_swizzle__ swizzle() const</pre>	<p>Available only when: The number of <b>swizzleIndexes</b> template parameters is 1, 2, 3, 4, 8, or 16. Available only when: Each of the <b>swizzleIndexes</b> template parameters is greater or equal to 0 and less than <b>NumElements</b>.</p> <p>Return an instance of the implementation-defined <b>__writeable_swizzle__</b> or <b>__const_swizzle__</b> class representing a swizzled view of the vector as described in <a href="#">Section 4.14.2.4</a>. The <b>swizzleIndexes</b> argument pack specifies the elements in the swizzle.</p>
<pre>DataT&amp; XYZW_ACCESS() const DataT&amp; XYZW_ACCESS() const</pre>	<p>Available only when: <b>NumElements</b> <math>\leq 4</math>.</p> <p>Return a reference to the element identified by <b>XYZW_ACCESS</b>.</p> <p>Where <b>XYZW_ACCESS</b> is: <b>x</b> for <b>NumElements</b> == 1, <b>x</b>, <b>y</b> for <b>NumElements</b> == 2, <b>x</b>, <b>y</b>, <b>z</b> for <b>NumElements</b> == 3 and <b>x</b>, <b>y</b>, <b>z</b>, <b>w</b> for <b>NumElements</b> == 4.</p>
<pre>DataT&amp; RGBA_ACCESS() const DataT&amp; RGBA_ACCESS() const</pre>	<p>Available only when: <b>NumElements</b> == 4.</p> <p>Return a reference to the element identified by <b>RGBA_ACCESS</b>.</p> <p>Where <b>RGBA_ACCESS</b> is: <b>r</b>, <b>g</b>, <b>b</b>, <b>a</b>.</p>

Member function	Description
<pre> DataT&amp; INDEX_ACCESS() const DataT&amp; INDEX_ACCESS() const </pre>	<p>Return a reference to the element identified by <code>INDEX_ACCESS</code>.</p> <p>Where <code>INDEX_ACCESS</code> is: <code>s0</code> for <code>NumElements == 1</code>, <code>s0</code>, <code>s1</code> for <code>NumElements == 2</code>, <code>s0</code>, <code>s1</code>, <code>s2</code> for <code>NumElements == 3</code>, <code>s0</code>, <code>s1</code>, <code>s2</code>, <code>s3</code> for <code>NumElements == 4</code>, <code>s0</code>, <code>s1</code>, <code>s2</code>, <code>s3</code>, <code>s4</code>, <code>s5</code>, <code>s6</code>, <code>s7</code>, <code>s8</code> for <code>NumElements == 8</code> and <code>s0</code>, <code>s1</code>, <code>s2</code>, <code>s3</code>, <code>s4</code>, <code>s5</code>, <code>s6</code>, <code>s7</code>, <code>s8</code>, <code>s9</code>, <code>sA</code>, <code>sB</code>, <code>sC</code>, <code>sD</code>, <code>sE</code>, <code>sF</code> for <code>NumElements == 16</code>.</p>
<pre> __writeable_swizzle__ XYZW_SWIZZLE() __const_swizzle__ XYZW_SWIZZLE() const </pre>	<p>Available only when: <code>NumElements &lt;= 4</code>, and when the macro <code>SYCL_SIMPLE_SWIZZLES</code> is defined before including <code>&lt;sycl/sycl.hpp&gt;</code>.</p> <p>Return an instance of the implementation-defined <code>__writeable_swizzle__</code> or <code>__const_swizzle__</code> class representing a swizzled view of the vector as described in <a href="#">Section 4.14.2.4</a>.</p> <p>Where <code>XYZW_SWIZZLE</code> is all permutations with repetition, of any subset with length greater than 1, of <code>x</code>, <code>y</code> for <code>NumElements == 2</code>, <code>x</code>, <code>y</code>, <code>z</code> for <code>NumElements == 3</code> and <code>x</code>, <code>y</code>, <code>z</code>, <code>w</code> for <code>NumElements == 4</code>. For example a four element <code>vec</code> provides permutations including <code>xzyw</code>, <code>xyyy</code> and <code>xz</code>.</p>
<pre> __writeable_swizzle__ RGBA_SWIZZLE() __const_swizzle__ RGBA_SWIZZLE() const </pre>	<p>Available only when: <code>NumElements == 4</code>, and when the macro <code>SYCL_SIMPLE_SWIZZLES</code> is defined before including <code>&lt;sycl/sycl.hpp&gt;</code>.</p> <p>Return an instance of the implementation-defined <code>__writeable_swizzle__</code> or <code>__const_swizzle__</code> class representing a swizzled view of the vector as described in <a href="#">Section 4.14.2.4</a>.</p> <p>Where <code>RGBA_SWIZZLE</code> is all permutations with repetition, of any subset with length greater than 1, of <code>r</code>, <code>g</code>, <code>b</code>, <code>a</code>. For example a four element <code>vec</code> provides permutations including <code>rbga</code>, <code>rggg</code> and <code>rb</code>.</p>



Member function	Description
<pre>__writeable_swizzle__ lo() __const_swizzle__ lo() <b>const</b></pre>	<p>Available only when: <code>NumElements &gt; 1</code>.</p> <p>Return an instance of the implementation-defined <code>__writeable_swizzle__</code> or <code>__const_swizzle__</code> class representing a swizzled view of the vector as described in <a href="#">Section 4.14.2.4</a>. The swizzle consists of the lower half of the elements in the vector. When <code>NumElements == 3</code>, the vector is treated as though <code>NumElements == 4</code> with the fourth element undefined.</p>
<pre>__writeable_swizzle__ hi() __const_swizzle__ hi() <b>const</b></pre>	<p>Available only when: <code>NumElements &gt; 1</code>.</p> <p>Return an instance of the implementation-defined <code>__writeable_swizzle__</code> or <code>__const_swizzle__</code> class representing a swizzled view of the vector as described in <a href="#">Section 4.14.2.4</a>. The swizzle consists of the upper half of the elements in the vector. When <code>NumElements == 3</code>, the vector is treated as though <code>NumElements == 4</code> with the fourth element undefined.</p>
<pre>__writeable_swizzle__ odd() __const_swizzle__ odd() <b>const</b></pre>	<p>Available only when: <code>NumElements &gt; 1</code>.</p> <p>Return an instance of the implementation-defined <code>__writeable_swizzle__</code> or <code>__const_swizzle__</code> class representing a swizzled view of the vector as described in <a href="#">Section 4.14.2.4</a>. The swizzle consists of the elements in the vector with an odd numbered index. When <code>NumElements == 3</code>, the vector is treated as though <code>NumElements == 4</code> with the fourth element undefined.</p>

Member function	Description
<pre>__writeable_swizzle__ even() __const_swizzle__ even() <b>const</b></pre>	<p>Available only when: <b>NumElements</b> &gt; 1.</p> <p>Return an instance of the implementation-defined <b>__writeable_swizzle__</b> or <b>__const_swizzle__</b> class representing a swizzled view of the vector as described in <a href="#">Section 4.14.2.4</a>. The swizzle consists of the elements in the vector with an even numbered index. When <b>NumElements</b> == 3, the vector is treated as though <b>NumElements</b> == 4 with the fourth element undefined.</p>
<pre>template &lt;access::address_space AddressSpace,           access::decorated IsDecorated&gt; void load(     size_t offset,     multi_ptr&lt;const DataT, AddressSpace, IsDecorated&gt; ptr)  void load(size_t offset, <b>const</b> DataT* ptr)</pre>	<p>Loads <b>NumElements</b> elements into the components of this SYCL <b>vec</b>. These elements are loaded from consecutive addresses, where the starting address is computed by adding <b>offset * NumElements * sizeof(DataT)</b> bytes to the address specified by the <b>ptr</b>. The <b>ptr</b> must be aligned to <b>alignof(DataT)</b>.</p>
<pre>template &lt;access::address_space AddressSpace,           access::decorated IsDecorated&gt; void store(     size_t offset,     multi_ptr&lt;DataT, AddressSpace, IsDecorated&gt; ptr) <b>const</b>  void store(size_t offset, DataT* ptr) <b>const</b></pre>	<p>Stores <b>NumElements</b> components of this SYCL <b>vec</b> into consecutive addresses, with the starting address determined by adding <b>offset * NumElements * sizeof(DataT)</b> to the address specified by the <b>ptr</b>. The <b>ptr</b> must be aligned to <b>alignof(DataT)</b>.</p>
<pre>DataT&amp; operator[](int index)</pre>	<p>Returns a reference to the element stored within this SYCL <b>vec</b> at the index specified by <b>index</b>.</p>
<pre><b>const</b> DataT&amp; operator[](int index) <b>const</b></pre>	<p>Returns a <b>const</b> reference to the element stored within this SYCL <b>vec</b> at the index specified by <b>index</b>.</p>
<pre>vec&amp; operator=(<b>const</b> vec&amp; rhs)</pre>	<p>Assign each element of the <b>rhs</b> SYCL <b>vec</b> to each element of this SYCL <b>vec</b> and return a reference to this SYCL <b>vec</b>.</p>
<pre>template&lt;typename T&gt; vec&amp; operator=(<b>const</b> T&amp; rhs)</pre>	<p><b>Constraints:</b> <b>T</b> is implicitly convertible to <b>DataT</b>.</p> <p>Assign the <b>rhs</b> scalar to each element of this SYCL <b>vec</b> and return a reference to this SYCL <b>vec</b>.</p>

Table 124. Hidden friend functions of the **vec** class template

Hidden friend function	Description
<pre>vec operatorOP(const vec&amp; lhs, const vec&amp; rhs)</pre>	<p>If <i>OP</i> is %, available only when: <i>DataT</i> != float &amp;&amp; <i>DataT</i> != double &amp;&amp; <i>DataT</i> != half.</p> <p>Construct a new instance of the SYCL <i>vec</i> class template with the same template parameters as <i>lhs vec</i> with each element of the new SYCL <i>vec</i> instance the result of an element-wise <i>OP</i> arithmetic operation between each element of <i>lhs vec</i> and each element of the <i>rhs</i> SYCL <i>vec</i>.</p> <p>Where <i>OP</i> is: +, -, *, /, %.</p>
<pre>template&lt;typename T&gt; vec operatorOP(const vec&amp; lhs, const T&amp; rhs)</pre>	<p><i>Constraints:</i></p> <ul style="list-style-type: none"> <li>• <i>T</i> is implicitly convertible to <i>DataT</i>, and</li> <li>• If <i>OP</i> is %, <i>DataT</i> != float &amp;&amp; <i>DataT</i> != double &amp;&amp; <i>DataT</i> != half.</li> </ul> <p>Construct a new instance of the SYCL <i>vec</i> class template with the same template parameters as <i>lhs vec</i> with each element of the new SYCL <i>vec</i> instance the result of an element-wise <i>OP</i> arithmetic operation between each element of <i>lhs vec</i> and the <i>rhs</i> scalar.</p> <p>Where <i>OP</i> is: +, -, *, /, %.</p>
<pre>template&lt;typename T&gt; vec operatorOP(const T&amp; lhs, const vec&amp; rhs)</pre>	<p><i>Constraints:</i></p> <ul style="list-style-type: none"> <li>• <i>T</i> is implicitly convertible to <i>DataT</i>, and</li> <li>• If <i>OP</i> is %, <i>DataT</i> != float &amp;&amp; <i>DataT</i> != double &amp;&amp; <i>DataT</i> != half.</li> </ul> <p>Construct a new instance of the SYCL <i>vec</i> class template with the same template parameters as the <i>rhs</i> SYCL <i>vec</i> with each element of the new SYCL <i>vec</i> instance the result of an element-wise <i>OP</i> arithmetic operation between the <i>lhs</i> scalar and each element of the <i>rhs</i> SYCL <i>vec</i>.</p> <p>Where <i>OP</i> is: +, -, *, /, %.</p>

Hidden friend function	Description
<pre>vec&amp; operatorOP(vec&amp; lhs, const vec&amp; rhs)</pre>	<p>If <b>OP</b> is <b>%=</b>, available only when: <b>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</b>.</p> <p>Perform an in-place element-wise <b>OP</b> arithmetic operation between each element of <b>lhs vec</b> and each element of the <b>rhs SYCL vec</b> and return <b>lhs vec</b>.</p> <p>Where <b>OP</b> is: <b>+=, -=, *=, /=, %=</b>.</p>
<pre>template&lt;typename T&gt; vec&amp; operatorOP(vec&amp; lhs, const T&amp; rhs)</pre>	<p><b>Constraints:</b></p> <ul style="list-style-type: none"> <li>• <b>T</b> is implicitly convertible to <b>DataT</b>, and</li> <li>• If <b>OP</b> is <b>%=</b>, <b>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</b>.</li> </ul> <p>Perform an in-place element-wise <b>OP</b> arithmetic operation between each element of <b>lhs vec</b> and <b>rhs scalar</b> and return <b>lhs vec</b>.</p> <p>Where <b>OP</b> is: <b>+=, -=, *=, /=, %=</b>.</p>
<pre>vec&amp; operatorOP(vec&amp; v)</pre>	<p>Available only when: <b>DataT != bool</b>.</p> <p>Perform an in-place element-wise <b>OP</b> prefix arithmetic operation on each element of <b>v</b> and return <b>v</b>.</p> <p>Where <b>OP</b> is: <b>++, --</b>.</p>
<pre>vec operatorOP(vec&amp; v, int)</pre>	<p>Available only when: <b>DataT != bool</b>.</p> <p>Perform an in-place element-wise <b>OP</b> postfix arithmetic operation on each element of <b>v</b> and return a copy of <b>v</b> before the operation is performed.</p> <p>Where <b>OP</b> is: <b>++, --</b>.</p>
<pre>vec operatorOP(const vec&amp; v)</pre>	<p>Construct a new instance of the SYCL <b>vec</b> class template with the same template parameters as this SYCL <b>vec</b> with each element of the new SYCL <b>vec</b> instance the result of an element-wise <b>OP</b> unary arithmetic operation on each element of this SYCL <b>vec</b>.</p> <p>Where <b>OP</b> is: <b>+, -</b>.</p>

Hidden friend function	Description
<pre data-bbox="161 185 778 219">vec operatorOP(const vec&amp; lhs, const vec&amp; rhs)</pre>	<p data-bbox="1002 159 1460 253">Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</p> <p data-bbox="1002 286 1460 607">Construct a new instance of the SYCL <code>vec</code> class template with the same template parameters as <code>lhs vec</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise <code>OP</code> bitwise operation between each element of <code>lhs vec</code> and each element of the <code>rhs</code> SYCL <code>vec</code>.</p> <p data-bbox="1002 640 1241 674">Where <code>OP</code> is: <code>&amp;</code>, <code> </code>, <code>^</code>.</p>
<pre data-bbox="161 719 751 786">template&lt;typename T&gt; vec operatorOP(const vec&amp; lhs, const T&amp; rhs)</pre>	<p data-bbox="1002 692 1158 725"><i>Constraints:</i></p> <ul data-bbox="1018 752 1460 909" style="list-style-type: none"> <li>• <code>T</code> is implicitly convertible to <code>DataT</code>, and</li> <li>• <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</li> </ul> <p data-bbox="1002 943 1460 1223">Construct a new instance of the SYCL <code>vec</code> class template with the same template parameters as <code>lhs vec</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise <code>OP</code> bitwise operation between each element of <code>lhs vec</code> and the <code>rhs</code> scalar.</p> <p data-bbox="1002 1256 1241 1290">Where <code>OP</code> is: <code>&amp;</code>, <code> </code>, <code>^</code>.</p>
<pre data-bbox="161 1332 751 1400">template&lt;typename T&gt; vec operatorOP(const T&amp; lhs, const vec&amp; rhs)</pre>	<p data-bbox="1002 1305 1158 1339"><i>Constraints:</i></p> <ul data-bbox="1018 1366 1460 1523" style="list-style-type: none"> <li>• <code>T</code> is implicitly convertible to <code>DataT</code>, and</li> <li>• <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</li> </ul> <p data-bbox="1002 1556 1460 1877">Construct a new instance of the SYCL <code>vec</code> class template with the same template parameters as the <code>rhs</code> SYCL <code>vec</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise <code>OP</code> bit-wise operation between the <code>lhs</code> scalar and each element of the <code>rhs</code> SYCL <code>vec</code>.</p> <p data-bbox="1002 1910 1241 1944">Where <code>OP</code> is: <code>&amp;</code>, <code> </code>, <code>^</code>.</p>

Hidden friend function	Description
<pre>vec&amp; operatorOP(vec&amp; lhs, const vec&amp; rhs)</pre>	<p>Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</p> <p>Perform an in-place element-wise <code>OP</code> bitwise operation between each element of <code>lhs vec</code> and the <code>rhs SYCL vec</code> and return <code>lhs vec</code>.</p> <p>Where <code>OP</code> is: <code>&amp;=</code>, <code> =</code>, <code>^=</code>.</p>
<pre>template&lt;typename T&gt; vec&amp; operatorOP(vec&amp; lhs, const T&amp; rhs)</pre>	<p><i>Constraints:</i></p> <ul style="list-style-type: none"> <li>• <code>T</code> is implicitly convertible to <code>DataT</code>, and</li> <li>• <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</li> </ul> <p>Perform an in-place element-wise <code>OP</code> bitwise operation between each element of <code>lhs vec</code> and the <code>rhs</code> scalar and return a <code>lhs vec</code>.</p> <p>Where <code>OP</code> is: <code>&amp;=</code>, <code> =</code>, <code>^=</code>.</p>
<pre>vec operatorOP(const vec&amp; lhs, const vec&amp; rhs)</pre>	<p>Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</p> <p>Construct a new instance of the SYCL <code>vec</code> class template with the same template parameters as <code>lhs vec</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise <code>OP</code> bitshift operation between each element of <code>lhs vec</code> and each element of the <code>rhs SYCL vec</code>. If <code>OP</code> is <code>&gt;&gt;</code>, <code>DataT</code> is a signed type and <code>lhs vec</code> has a negative value any vacated bits viewed as an unsigned integer must be assigned the value <code>1</code>, otherwise any vacated bits viewed as an unsigned integer must be assigned the value <code>0</code>.</p> <p>Where <code>OP</code> is: <code>&lt;&lt;</code>, <code>&gt;&gt;</code>.</p>

Hidden friend function	Description
<pre>template&lt;typename T&gt; vec operatorOP(const vec&amp; lhs, const T&amp; rhs)</pre>	<p><i>Constraints:</i></p> <ul style="list-style-type: none"> <li>• <b>T</b> is implicitly convertible to <b>DataT</b>, and</li> <li>• <b>DataT</b> <b>!=</b> <b>float</b> <b>&amp;&amp;</b> <b>DataT</b> <b>!=</b> <b>double</b> <b>&amp;&amp;</b> <b>DataT</b> <b>!=</b> <b>half</b>.</li> </ul> <p>Construct a new instance of the SYCL <b>vec</b> class template with the same template parameters as <b>lhs vec</b> with each element of the new SYCL <b>vec</b> instance the result of an element-wise <b>OP</b> bitshift operation between each element of <b>lhs vec</b> and the <b>rhs</b> scalar. If <b>OP</b> is <b>&gt;&gt;</b>, <b>DataT</b> is a signed type and <b>lhs vec</b> has a negative value any vacated bits viewed as an unsigned integer must be assigned the value <b>1</b>, otherwise any vacated bits viewed as an unsigned integer must be assigned the value <b>0</b>.</p> <p>Where <b>OP</b> is: <b>&lt;&lt;</b>, <b>&gt;&gt;</b>.</p>
<pre>template&lt;typename T&gt; vec operatorOP(const T&amp; lhs, const vec&amp; rhs)</pre>	<p><i>Constraints:</i></p> <ul style="list-style-type: none"> <li>• <b>T</b> is implicitly convertible to <b>DataT</b>, and</li> <li>• <b>DataT</b> <b>!=</b> <b>float</b> <b>&amp;&amp;</b> <b>DataT</b> <b>!=</b> <b>double</b> <b>&amp;&amp;</b> <b>DataT</b> <b>!=</b> <b>half</b>.</li> </ul> <p>Construct a new instance of the SYCL <b>vec</b> class template with the same template parameters as the <b>rhs</b> SYCL <b>vec</b> with each element of the new SYCL <b>vec</b> instance the result of an element-wise <b>OP</b> bitshift operation between the <b>lhs</b> scalar and each element of the <b>rhs</b> SYCL <b>vec</b>. If <b>OP</b> is <b>&gt;&gt;</b>, <b>DataT</b> is a signed type and this SYCL <b>vec</b> has a negative value any vacated bits viewed as an unsigned integer must be assigned the value <b>1</b>, otherwise any vacated bits viewed as an unsigned integer must be assigned the value <b>0</b>.</p> <p>Where <b>OP</b> is: <b>&lt;&lt;</b>, <b>&gt;&gt;</b>.</p>

Hidden friend function	Description
<pre>vec&amp; operatorOP(vec&amp; lhs, const vec&amp; rhs)</pre>	<p>Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</p> <p>Perform an in-place element-wise <code>OP</code> bitshift operation between each element of <code>lhs vec</code> and the <code>rhs SYCL vec</code> and returns <code>lhs vec</code>. If <code>OP</code> is <code>&gt;&gt;=</code>, <code>DataT</code> is a signed type and <code>lhs vec</code> has a negative value any vacated bits viewed as an unsigned integer must be assigned the value <code>1</code>, otherwise any vacated bits viewed as an unsigned integer must be assigned the value <code>0</code>.</p> <p>Where <code>OP</code> is: <code>&lt;&lt;=</code>, <code>&gt;&gt;=</code>.</p>
<pre>template&lt;typename T&gt; vec&amp; operatorOP(vec&amp; lhs, const T&amp; rhs)</pre>	<p><i>Constraints:</i></p> <ul style="list-style-type: none"> <li><code>T</code> is implicitly convertible to <code>DataT</code>, and</li> <li><code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</li> </ul> <p>Perform an in-place element-wise <code>OP</code> bitshift operation between each element of <code>lhs vec</code> and the <code>rhs scalar</code> and returns a reference to this SYCL <code>vec</code>. If <code>OP</code> is <code>&gt;&gt;=</code>, <code>DataT</code> is a signed type and <code>lhs vec</code> has a negative value any vacated bits viewed as an unsigned integer must be assigned the value <code>1</code>, otherwise any vacated bits viewed as an unsigned integer must be assigned the value <code>0</code>.</p> <p>Where <code>OP</code> is: <code>&lt;&lt;=</code>, <code>&gt;&gt;=</code>.</p>



Hidden friend function	Description
<pre data-bbox="164 185 962 253">vec&lt;RET, NumElements&gt; operatorOP(const vec&amp; lhs, const vec&amp; rhs)</pre>	<p data-bbox="1005 159 1460 483">Construct a new instance of the SYCL <code>vec</code> class template with the same template parameters as <code>lhs vec</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise <code>OP</code> logical operation between each element of <code>lhs vec</code> and each element of the <code>rhs</code> SYCL <code>vec</code>.</p> <p data-bbox="1005 510 1460 909">The element type of the returned <code>vec</code>, <code>RET</code>, varies depending on the size of the <code>DataT</code> template parameter of the input <code>vec</code>. If <code>sizeof(DataT)</code> is 1, <code>RET</code> is <code>int8_t</code>. If <code>sizeof(DataT)</code> is 2, <code>RET</code> is <code>int16_t</code>. If <code>sizeof(DataT)</code> is 4, <code>RET</code> is <code>int32_t</code>. If <code>sizeof(DataT)</code> is 8, <code>RET</code> is <code>int64_t</code>. If <code>sizeof(DataT)</code> is any other value, <code>RET</code> is an implementation defined integer type.</p> <p data-bbox="1005 936 1241 969">Where <code>OP</code> is: <code>&amp;&amp;</code>, <code>  </code>.</p>
<pre data-bbox="164 1014 930 1126">template&lt;typename T&gt; vec&lt;RET, NumElements&gt; operatorOP(const vec&amp; lhs, const T&amp; rhs)</pre>	<p data-bbox="1005 992 1460 1059"><i>Constraints:</i> <code>T</code> is implicitly convertible to <code>DataT</code>.</p> <p data-bbox="1005 1086 1460 1373">Construct a new instance of the SYCL <code>vec</code> class template with the same template parameters as this SYCL <code>vec</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise <code>OP</code> logical operation between each element of <code>lhs vec</code> and the <code>rhs</code> scalar.</p> <p data-bbox="1005 1400 1460 1798">The element type of the returned <code>vec</code>, <code>RET</code>, varies depending on the size of the <code>DataT</code> template parameter of the input <code>vec</code>. If <code>sizeof(DataT)</code> is 1, <code>RET</code> is <code>int8_t</code>. If <code>sizeof(DataT)</code> is 2, <code>RET</code> is <code>int16_t</code>. If <code>sizeof(DataT)</code> is 4, <code>RET</code> is <code>int32_t</code>. If <code>sizeof(DataT)</code> is 8, <code>RET</code> is <code>int64_t</code>. If <code>sizeof(DataT)</code> is any other value, <code>RET</code> is an implementation defined integer type.</p> <p data-bbox="1005 1825 1241 1859">Where <code>OP</code> is: <code>&amp;&amp;</code>, <code>  </code>.</p>

Hidden friend function	Description
<pre data-bbox="161 185 930 286"> template&lt;typename T&gt; vec&lt;RET, NumElements&gt; operatorOP(const T&amp; lhs, const vec&amp; rhs) </pre>	<p data-bbox="1005 159 1457 226"><i>Constraints:</i> <b>T</b> is implicitly convertible to <b>DataT</b>.</p> <p data-bbox="1005 259 1457 577">Construct a new instance of the SYCL <b>vec</b> class template with the same template parameters as the <b>rhs</b> SYCL <b>vec</b> with each element of the new SYCL <b>vec</b> instance the result of an element-wise <b>OP</b> logical operation between the <b>lhs</b> scalar and each element of the <b>rhs</b> SYCL <b>vec</b>.</p> <p data-bbox="1005 611 1457 1003">The element type of the returned <b>vec</b>, <b>RET</b>, varies depending on the size of the <b>DataT</b> template parameter of the input <b>vec</b>. If <b>sizeof(DataT)</b> is 1, <b>RET</b> is <b>int8_t</b>. If <b>sizeof(DataT)</b> is 2, <b>RET</b> is <b>int16_t</b>. If <b>sizeof(DataT)</b> is 4, <b>RET</b> is <b>int32_t</b>. If <b>sizeof(DataT)</b> is 8, <b>RET</b> is <b>int64_t</b>. If <b>sizeof(DataT)</b> is any other value, <b>RET</b> is an implementation defined integer type.</p> <p data-bbox="1005 1037 1241 1070">Where <b>OP</b> is: <b>&amp;&amp;</b>, <b>  </b>.</p>

Hidden friend function	Description
<pre data-bbox="161 185 959 253">vec&lt;RET, NumElements&gt; operatorOP(const vec&amp; lhs, const vec&amp; rhs)</pre>	<p data-bbox="1002 159 1458 808">Construct a new instance of the SYCL <code>vec</code> class template with the element type <code>RET</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise <code>OP</code> relational operation between each element of <code>lhs vec</code> and each element of the <code>rhs</code> SYCL <code>vec</code>. Each element of the SYCL <code>vec</code> that is returned must be <code>-1</code> if the operation results in <code>true</code> and <code>0</code> if the operation results in <code>false</code>. The <code>==</code>, <code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code> and <code>&gt;=</code> operations result in <code>false</code> if either the <code>lhs</code> element or the <code>rhs</code> element is a NaN. The <code>!=</code> operation results in <code>true</code> if either the <code>lhs</code> element or the <code>rhs</code> element is a NaN.</p> <p data-bbox="1002 837 1458 1234">The element type of the returned <code>vec</code>, <code>RET</code>, varies depending on the size of the <code>DataT</code> template parameter of the input <code>vec</code>. If <code>sizeof(DataT)</code> is 1, <code>RET</code> is <code>int8_t</code>. If <code>sizeof(DataT)</code> is 2, <code>RET</code> is <code>int16_t</code>. If <code>sizeof(DataT)</code> is 4, <code>RET</code> is <code>int32_t</code>. If <code>sizeof(DataT)</code> is 8, <code>RET</code> is <code>int64_t</code>. If <code>sizeof(DataT)</code> is any other value, <code>RET</code> is an implementation defined integer type.</p> <p data-bbox="1002 1263 1378 1296">Where <code>OP</code> is: <code>==</code>, <code>!=</code>, <code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code>, <code>&gt;=</code>.</p>

Hidden friend function	Description
<pre>template&lt;typename T&gt; vec&lt;RET, NumElements&gt; operatorOP(const vec&amp; lhs, const T&amp; rhs)</pre>	<p><i>Constraints:</i> <b>T</b> is implicitly convertible to <b>DataT</b>.</p> <p>Construct a new instance of the SYCL <b>vec</b> class template with the <b>DataT</b> parameter of <b>RET</b> with each element of the new SYCL <b>vec</b> instance the result of an element-wise <b>OP</b> relational operation between each element of <b>lhs vec</b> and the <b>rhs</b> scalar. Each element of the SYCL <b>vec</b> that is returned must be <b>-1</b> if the operation results in <b>true</b> and <b>0</b> if the operation results in <b>false</b>. The <b>==</b>, <b>&lt;</b>, <b>&gt;</b>, <b>&lt;=</b> and <b>&gt;=</b> operations result in <b>false</b> if either the <b>lhs</b> element or the <b>rhs</b> is a NaN. The <b>!=</b> operation results in <b>true</b> if either the <b>lhs</b> element or the <b>rhs</b> is a NaN.</p> <p>The element type of the returned <b>vec</b>, <b>RET</b>, varies depending on the size of the <b>DataT</b> template parameter of the input <b>vec</b>. If <b>sizeof(DataT)</b> is 1, <b>RET</b> is <b>int8_t</b>. If <b>sizeof(DataT)</b> is 2, <b>RET</b> is <b>int16_t</b>. If <b>sizeof(DataT)</b> is 4, <b>RET</b> is <b>int32_t</b>. If <b>sizeof(DataT)</b> is 8, <b>RET</b> is <b>int64_t</b>. If <b>sizeof(DataT)</b> is any other value, <b>RET</b> is an implementation defined integer type.</p> <p>Where <b>OP</b> is: <b>==</b>, <b>!=</b>, <b>&lt;</b>, <b>&gt;</b>, <b>&lt;=</b>, <b>&gt;=</b>.</p>

Hidden friend function	Description
<pre>template&lt;typename T&gt; vec&lt;RET, NumElements&gt; operatorOP(const T&amp; lhs, const vec&amp; rhs)</pre>	<p><i>Constraints:</i> <b>T</b> is implicitly convertible to <b>DataT</b>.</p> <p>Construct a new instance of the SYCL <b>vec</b> class template with the element type <b>RET</b> with each element of the new SYCL <b>vec</b> instance the result of an element-wise <b>OP</b> relational operation between the <b>lhs</b> scalar and each element of the <b>rhs</b> SYCL <b>vec</b>. Each element of the SYCL <b>vec</b> that is returned must be <b>-1</b> if the operation results in <b>true</b> and <b>0</b> if the operation results in <b>false</b>. The <b>==</b>, <b>&lt;</b>, <b>&gt;</b>, <b>&lt;=</b> and <b>&gt;=</b> operations result in <b>false</b> if either the <b>lhs</b> or the <b>rhs</b> element is a NaN. The <b>!=</b> operation results in <b>true</b> if either the <b>lhs</b> or the <b>rhs</b> element is a NaN.</p> <p>The element type of the returned <b>vec</b>, <b>RET</b>, varies depending on the size of the <b>DataT</b> template parameter of the input <b>vec</b>. If <b>sizeof(DataT)</b> is 1, <b>RET</b> is <b>int8_t</b>. If <b>sizeof(DataT)</b> is 2, <b>RET</b> is <b>int16_t</b>. If <b>sizeof(DataT)</b> is 4, <b>RET</b> is <b>int32_t</b>. If <b>sizeof(DataT)</b> is 8, <b>RET</b> is <b>int64_t</b>. If <b>sizeof(DataT)</b> is any other value, <b>RET</b> is an implementation defined integer type.</p> <p>Where <b>OP</b> is: <b>==</b>, <b>!=</b>, <b>&lt;</b>, <b>&gt;</b>, <b>&lt;=</b>, <b>&gt;=</b>.</p>
<pre>vec operator~(const vec&amp; v)</pre>	<p>Available only when: <b>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</b>.</p> <p>Construct a new instance of the SYCL <b>vec</b> class template with the same template parameters as <b>v vec</b> with each element of the new SYCL <b>vec</b> instance the result of an element-wise bitwise NOT operation on each element of <b>v vec</b>.</p>

Hidden friend function	Description
<pre data-bbox="161 185 766 219">vec&lt;RET, NumElements&gt; operator!(const vec&amp; v)</pre>	<p data-bbox="1002 159 1460 589">Construct a new instance of the SYCL <code>vec</code> class template with the same template parameters as <code>v vec</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise logical NOT operation on each element of <code>v vec</code>. Each element of the SYCL <code>vec</code> that is returned must be <code>-1</code> if the operation results in <code>true</code> and <code>0</code> if the operation results in <code>false</code> or this SYCL <code>vec</code> is a NaN.</p> <p data-bbox="1002 618 1460 1010">The element type of the returned <code>vec</code>, <code>RET</code>, varies depending on the size of the <code>DataT</code> template parameter of the input <code>vec</code>. If <code>sizeof(DataT)</code> is 1, <code>RET</code> is <code>int8_t</code>. If <code>sizeof(DataT)</code> is 2, <code>RET</code> is <code>int16_t</code>. If <code>sizeof(DataT)</code> is 4, <code>RET</code> is <code>int32_t</code>. If <code>sizeof(DataT)</code> is 8, <code>RET</code> is <code>int64_t</code>. If <code>sizeof(DataT)</code> is any other value, <code>RET</code> is an implementation defined integer type.</p>

#### 4.14.2.2. Aliases

The SYCL programming API provides all permutations of the type alias:

```
using <type><elems> = vec<<storage-type>, <elems>>
```

where `<elems>` is 2, 3, 4, 8 and 16, and pairings of `<type>` and `<storage-type>` for integral types are `char` and `int8_t`, `uchar` and `uint8_t`, `short` and `int16_t`, `ushort` and `uint16_t`, `int` and `int32_t`, `uint` and `uint32_t`, `long` and `int64_t`, `ulong` and `uint64_t`, and for floating point types are both `half`, `float` and `double`.

For example `uint4` is the alias to `vec<uint32_t, 4>` and `float16` is the alias to `vec<float, 16>`.

#### 4.14.2.3. Swizzles

Swizzle operations can be performed in two ways. Firstly by calling the `swizzle` member function template, which takes a variadic number of integer template arguments between 0 and `NumElements-1`, specifying swizzle indexes. Secondly by calling one of the simple swizzle member functions defined in Table 123 as `XYZW_SWIZZLE` and `RGBA_SWIZZLE`. Note that the simple swizzle functions are only available for up to 4 element vectors and are only available when the macro `SYCL_SIMPLE_SWIZZLES` is defined before including `<sycl/sycl.hpp>`.

In both cases the return value is an instance of either the `__writeable_swizzle__` or the `__const_swizzle__` class template. These classes have an implementation-defined name, and they represent a view of the original `vec` object with the swizzle operation applied. The `__writeable_swizzle__` class represents a writeable view of the `vec` object, while the `__const_swizzle__` class represents a read-only view. Since the swizzle operation may result in a different number of elements, these views may represent a different number of elements than the original `vec` object.

Both the `swizzle` member function template and the simple swizzle member functions allow swizzle indexes to be repeated.

A series of static constexpr values are provided within the `elem` struct to allow specifying named swizzle indexes when calling the `swizzle` member function template.

#### 4.14.2.4. The swizzled vector classes

The `__writeable_swizzle__` and `__const_swizzle__` classes are each views over a `vec` object which captures the effects of a swizzle operation without actually performing that operation. The tables below define the interfaces to these classes, but in general `__writeable_swizzle__` supports the same interface as `vec`, while `__const_swizzle__` supports only the non-mutating operations of `vec`. Member functions and operators that read elements from these views return elements from the underlying `vec` as translated by the captured swizzle operation. Member functions and operators that modify elements of these views modify corresponding elements of the underlying `vec` as translated by the captured swizzle operation. The following example illustrates this behavior:

```
1 #include <sycl/sycl.hpp>
2 using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names
3
4 int main() {
5     vec v{5, 6, 7, 8};
6     vec<int, 2> slice = v.swizzle<2, 1>(); // slice has the value {7,6}
7     slice = v.swizzle<2, 2>();           // slice is now {7,7}
8     int i = v.swizzle<2, 1>()[1];        // i has the value 6
9     v.swizzle<2>() = 0;                  // v is now {5,6,0,8}
10    v.swizzle<1>()++;                     // v is now {5,7,0,8}
11    v.swizzle<2, 3>(); // Has no effect because result of swizzle is
12                      // neither read nor modified
13 }
```

Synopses of the `__writeable_swizzle__` and `__const_swizzle__` classes are shown below. The member type aliases are described in [Section 4.14.2.4.1](#), the constructors are described in [Section 4.14.2.4.2](#), the member functions are described in [Section 4.14.2.4.4](#), and the hidden friend functions are described in [Section 4.14.2.4.5](#).

The `__writeable_swizzle__` and `__const_swizzle__` classes are not user constructible.

The `__writeable_swizzle__` and `__const_swizzle__` types are class templates, but the template parameters are unspecified. The description below describes the member functions and hidden friend functions using two exposition-only private members named `DataT` and `NumElements`. The type alias `DataT` represents the element type of the underlying `vec`. The constant `NumElements` represents the number of elements in the result of the swizzle operation, which could be different from the number of elements in the underlying `vec`.

Because the template parameters to the `__writeable_swizzle__` and `__const_swizzle__` classes affect the C++ argument dependent lookup (ADL) behavior, implementations do not have complete freedom when choosing these template parameters. Implementations must choose template parameters that result in the following ADL behavior:

- When an argument is the `__writeable_swizzle__` type, the ADL search set must include only the class template definition for `__writeable_swizzle__`, the class definition for `DataT` (if it is a class), and the namespace that contains `DataT`.
- When an argument is the `__const_swizzle__` type, the ADL search set must include only the class template definition for `__const_swizzle__`, the class definition for `DataT` (if it is a class), and the namespace that contains `DataT`.

When the string `__writeable_swizzle__` or `__const_swizzle__` is used inside the class definition in the synopses below, it refers to the instantiation with the same set of template parameters as the enclosing class.

This is consistent with C++ syntax. When the string `__writeable_swizzle__` or `__const_swizzle__` is used inside the class definition, it refers to a possibly different instantiation of the `__writeable_swizzle__` or `__const_swizzle__` class.

Although the synopses below illustrate `__writeable_swizzle__` and `__const_swizzle__` as two separate classes, this is exposition only. An implementation could instead implement a single combined class with additional constraints on the member functions.

```

1 namespace /*unspecified*/ {
2
3 template </*unspecified*/>
4 class __writeable_swizzle__ {
5 private:
6     // The "DataT" and "NumElements" members are exposition-only
7     using DataT = /* element type of the underlying vec */;
8     static constexpr int NumElements = /* number of elements produced by the swizzle */;
9
10 public:
11     // Public members as defined in the tables below
12     // Hidden friend functions as defined in the table below
13 };
14
15 template </*unspecified*/>
16 class __const_swizzle__ {
17 private:
18     // The "DataT" and "NumElements" members are exposition-only
19     using DataT = /* element type of the underlying vec */;
20     static constexpr int NumElements = /* number of elements produced by the swizzle */;
21
22 public:
23     // Public members as defined in the tables below
24     // Hidden friend functions as defined in the table below
25 };
26
27 } // namespace

```

#### 4.14.2.4.1. Member type aliases for the swizzled vector class templates

```

using element_type = DataT
using value_type = DataT

```

Each of these type aliases tells the type of an element in the underlying `vec`.

#### 4.14.2.4.2. Constructors for the swizzled vector class templates

```

__writeable_swizzle__() = delete
__writeable_swizzle__(const __writeable_swizzle__&) = delete

__const_swizzle__() = delete
__const_swizzle__(const __const_swizzle__&) = delete

```

The default constructor and copy constructor are deleted.



## 4.14.2.4.3. Destructors for the swizzled vector class templates

```
~__writeable_swizzle__()
~__const_swizzle__()
```

The destructors have no visible effect.

## 4.14.2.4.4. Member functions for the swizzled vector class templates

```
operator DataT() const

template<typename T>
explicit operator T() const

static constexpr size_t byte_size() noexcept
static constexpr size_t size() noexcept

size_t get_size() const // Deprecated
size_t get_count() const // Deprecated

template <typename ConvertT,
          rounding_mode RoundingMode = rounding_mode::automatic>
vec<ConvertT, NumElements> convert() const

template <typename asT> asT as() const

template <access::address_space AddressSpace, access::decorated IsDecorated>
void store(size_t offset, multi_ptr<DataT, AddressSpace, IsDecorated> ptr) const
```

*Availability:* These functions are available in both `__writeable_swizzle__` and `__const_swizzle__`.

*Constraints:* These functions have the same constraints as the equivalent member functions of the `vec` class.

*Effects:* The effect of these functions is the same as if they were called on a temporary `vec` object that contains the result of the captured swizzle operation.

```
operator vec<DataT, NumElements>() const
```

*Availability:* These functions are available in both `__writeable_swizzle__` and `__const_swizzle__`.

*Constraints:* Available only when `NumElements > 1`.

*Returns:* A new `vec` object that contains the result of the captured swizzle operation.

```

DataT& XYZW_ACCESS() const      (1)
DataT& RGBA_ACCESS() const    (2)
DataT& INDEX_ACCESS() const    (3)

const DataT& XYZW_ACCESS() const (4)
const DataT& RGBA_ACCESS() const (5)
const DataT& INDEX_ACCESS() const (6)

```

**Availability:** Functions (1) - (3) are available only in `__writeable_swizzle__`. Functions (4) - (6) are available only in `__const_swizzle__`.

**Constraints:** These functions have the same constraints as the equivalent member functions of the `vec` class.

**Returns:** A reference to the element of the underlying `vec` object that corresponds to the position of the swizzle operation identified by `XYZW_ACCESS`, `RGBA_ACCESS`, or `INDEX_ACCESS`.

```

template <int... swizzleIndexes> __writeable_swizzle__/*unspecified*/ swizzle() const (1)

#ifdef SYCL_SIMPLE_SWIZZLES
__writeable_swizzle__/*unspecified*/ XYZW_SWIZZLE() const (2)
__writeable_swizzle__/*unspecified*/ RGBA_SWIZZLE() const (3)
#endif

__writeable_swizzle__/*unspecified*/ lo() const (4)
__writeable_swizzle__/*unspecified*/ hi() const (5)
__writeable_swizzle__/*unspecified*/ odd() const (6)
__writeable_swizzle__/*unspecified*/ even() const (7)

template <int... swizzleIndexes> __const_swizzle__/*unspecified*/ swizzle() const (8)

#ifdef SYCL_SIMPLE_SWIZZLES
__const_swizzle__/*unspecified*/ XYZW_SWIZZLE() const (9)
__const_swizzle__/*unspecified*/ RGBA_SWIZZLE() const (10)
#endif

__const_swizzle__/*unspecified*/ lo() const (11)
__const_swizzle__/*unspecified*/ hi() const (12)
__const_swizzle__/*unspecified*/ odd() const (13)
__const_swizzle__/*unspecified*/ even() const (14)

```

**Availability:** Functions (1) - (7) are available only in `__writeable_swizzle__`. Functions (8) - (14) are available only in `__const_swizzle__`.

**Constraints:** These functions have the same constraints as the equivalent member functions of the `vec` class.

**Returns:** A new view of the underlying `vec` object, where the view represents the composition of two swizzle operations. The first is the swizzle operation represented by the `__writeable_swizzle__` or `__const_swizzle__` view. The second is the swizzle operation defined by the member function. The indices used by the second swizzle are the indices produced by the first swizzle. For example, if the second swiz-

zle references the first element, this means the element of the underlying **vec** that corresponds to the first element produced by the first swizzle.

```
template <access::address_space AddressSpace, access::decorated IsDecorated>
void load(size_t offset, multi_ptr<const DataT, AddressSpace, IsDecorated> ptr) const
```

*Availability:* Available only in **\_\_writeable\_swizzle\_\_**.

*Constraints:* Available only when the **\_\_writeable\_swizzle\_\_** view does not contain any repeated elements.

*Effects:* Loads values from memory into elements of the underlying **vec** object. A total of **NumElements** values are loaded from memory, starting at the the address **ptr + offset\*sizeof(DataT)**. The first value from memory is written to the element in **vec** that corresponds to the first element of the swizzle operation. The second value from memory is written to the element in **vec** that corresponds to the second element of the swizzle operation, etc.

```
DataT& operator[](int index) const (1)
const DataT& operator[](int index) const (2)
```

*Availability:* Functions (1) is available only in **\_\_writeable\_swizzle\_\_**. Functions (2) is available only in **\_\_const\_swizzle\_\_**.

*Returns:* A reference to the element of the underlying **vec** object that corresponds to the position **index** of the swizzle operation.

```
const __writeable_swizzle__&
operator=(const __writeable_swizzle__& rhs) const
```

*Availability:* Available only in **\_\_writeable\_swizzle\_\_**.

*Constraints:* Available only when the **\_\_writeable\_swizzle\_\_** view does not contain any repeated elements.

*Effects:* Assigns elements from the right hand side **\_\_writeable\_swizzle\_\_** view to elements of the left hand side **\_\_writeable\_swizzle\_\_** view. The value corresponding to the first element of the **rhs** swizzle operation is assigned to the element of the underlying **vec** object that corresponds to the first element of the left hand side swizzle operation, etc.

*Returns:* A reference to the left hand side **\_\_writeable\_swizzle\_\_** view.

```
const __const_swizzle__&
operator=(const __const_swizzle__& rhs) const = delete;
```

The copy assignment operator is deleted for the **\_\_const\_swizzle\_\_** class.

```
template</*unspecified*/>
const __writeable_swizzle__&
operator=(const __writeable_swizzle__</*unspecified*/>& rhs) const

template</*unspecified*/>
const __writeable_swizzle__&
operator=(const __const_swizzle__</*unspecified*/>& rhs) const
```

*Availability:* Available only in `__writeable_swizzle__`.

*Constraints:* Available only when all of the following conditions are met:

- The element data type of `rhs` is the same as `DataT`;
- The number of elements in the `rhs` view is equal to `NumElements`; and
- The `__writeable_swizzle__` view (i.e. the left hand side of the assignment) does not contain any repeated elements.

*Effects:* Assigns elements from the right hand side `__writeable_swizzle__` or `__const_swizzle__` view to elements of the left hand side `__writeable_swizzle__` view. The value corresponding to the first element of the `rhs` swizzle operation is assigned to the element of the underlying `vec` object that corresponds to the first element of the left hand side swizzle operation, etc.

*Returns:* A reference to the left hand side `__writeable_swizzle__` view.

```
const __writeable_swizzle__& operator=(const DataT& rhs) const
```

*Availability:* Available only in `__writeable_swizzle__`.

*Constraints:* Available only when the `__writeable_swizzle__` view does not contain any repeated elements.

*Effects:* Assigns the value `rhs` to those elements of the underlying `vec` object that have corresponding elements in the `__writeable_swizzle__` view. Elements in the underlying `vec` object that do not have elements in the `__writeable_swizzle__` view are not assigned.

*Returns:* A reference to the `__writeable_swizzle__` view.

```
const __writeable_swizzle__& operator=(const vec<DataT, NumElements>& rhs) const
```

*Availability:* Available only in `__writeable_swizzle__`.

*Constraints:* Available only when the `__writeable_swizzle__` view does not contain any repeated elements.

*Effects:* Assigns elements from `rhs` to elements of the `vec` object that underlies this `__writeable_swizzle__` view. The first element of `rhs` is assigned to the element of the underlying `vec` object that corresponds to the first element of the swizzle operation, etc.

*Returns:* A reference to the `__writeable_swizzle__` view.

## 4.14.2.4.5. Hidden friend functions of the swizzled vector class templates

```
template</*unspecified*/> (1)
friend vec<DataT, NumElements>
operatorOP(const __writeable_swizzle__& lhs,
           const __writeable_swizzle__</*unspecified*/>& rhs)
```

```
template</*unspecified*/> (2)
friend vec<DataT, NumElements>
operatorOP(const __writeable_swizzle__& lhs,
           const __const_swizzle__</*unspecified*/>& rhs)
```

```
template</*unspecified*/> (3)
friend vec<DataT, NumElements>
operatorOP(const __const_swizzle__</*unspecified*/>& lhs,
           const __writeable_swizzle__& rhs)
```

```
template</*unspecified*/> (4)
friend vec<DataT, NumElements>
operatorOP(const __const_swizzle__& lhs, const __const_swizzle__</*unspecified*/>& rhs)
```

```
friend vec<DataT, NumElements> (5)
operatorOP(const vec<DataT, NumElements>& lhs, const __writeable_swizzle__& rhs)
```

```
friend vec<DataT, NumElements> (6)
operatorOP(const vec<DataT, NumElements>& lhs, const __const_swizzle__& rhs)
```

```
friend vec<DataT, NumElements> (7)
operatorOP(const __writeable_swizzle__& lhs, const vec<DataT, NumElements>& rhs)
```

```
friend vec<DataT, NumElements> (8)
operatorOP(const __const_swizzle__& lhs, const vec<DataT, NumElements>& rhs)
```

```
template<typename T> (9)
friend vec<DataT, NumElements>
operatorOP(const __writeable_swizzle__& lhs, const T& rhs)
```

```
template<typename T> (10)
friend vec<DataT, NumElements>
operatorOP(const __const_swizzle__& lhs, const T& rhs)
```

```
template<typename T> (11)
friend vec<DataT, NumElements>
operatorOP(const T& lhs, const __writeable_swizzle__& rhs)
```

```
template<typename T> (12)
friend vec<DataT, NumElements>
operatorOP(const T& lhs, const __const_swizzle__& rhs)
```

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

*Availability:* Overloads (1), (2), (3), (5), (7), (9), and (11) are hidden friends of `__writeable_swizzle__`. Overloads (4), (6), (8), (10), and (12) are hidden friends of `__const_swizzle__`.

*Constraints:* If **OP** is one of the following: %, &, |, ^, <<, >>; available only when: `DataT != float && DataT !=`

`double && DataT != half.`

Overloads (1) - (4) are available only when the element data type of `lhs` is the same as the element data type of `rhs` and when the number of elements in the `lhs` view is equal to the number of elements in the `rhs` view.

Overloads (9) - (12) are available only when `T` is implicitly convertible to `DataT` and when `T` is not one of the swizzled vector class templates.

*Effects:* These functions behave as though the swizzle operation represented by each `__writeable_swizzle__` or `__const_swizzle__` parameter was first evaluated into a temporary `vec` object, and then `operatorOP` was called with the temporary `vec` object.

*Returns:* A new `vec` object that represents the result of the operation.

```

template</*unspecified*/> (1)
friend const __writeable_swizzle__&
operatorOP(const __writeable_swizzle__& lhs,
           const __writeable_swizzle__</*unspecified*/>& rhs)

template</*unspecified*/> (2)
friend const __writeable_swizzle__&
operatorOP(const __writeable_swizzle__& lhs,
           const __const_swizzle__</*unspecified*/>& rhs)

friend const __writeable_swizzle__& (3)
operatorOP(const __writeable_swizzle__& lhs, const vec<DataT, NumElements>& rhs)

template<typename T>
friend const __writeable_swizzle__& (4)
operatorOP(const __writeable_swizzle__& lhs, const T& rhs)

```

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

*Availability:* These are hidden friend functions only in `__writeable_swizzle__`.

*Constraints:* Available only when the left hand side `__writeable_swizzle__` view does not contain any repeated elements.

If `OP` is one of the following: `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`; available only when: `DataT != float && DataT != double && DataT != half`.

Overloads (1) and (2) are available only when the element data type of `lhs` is the same as the element data type of `rhs` and when the number of elements in the `lhs` view is equal to the number of elements in the `rhs` view.

Overloads (4) is available only when `T` is implicitly convertible to `DataT` and when `T` is not one of the swizzled vector class templates.

*Effects:* These functions operate as follow.

A left hand side value is computed from `lhs` by applying the swizzle operation on the underlying `vec` object. If the `rhs` is a `__writeable_swizzle__` view, the right hand side value is computed the same way. Otherwise, the right hand side value is the same as `rhs`.

The non-assignment part of the operation is performed on these two values, producing a result. This

result is assigned to **lhs** as follows.

The first element of the result is assigned to the **vec** element that corresponds to the first element of the left-hand-side swizzle. The second element of the result is assigned to the **vec** element that corresponds to the second element of the left-hand-side swizzle, etc.

*Returns:* A reference to the **lhs**.

```
friend const __writeable_swizzle__& operatorOP(const __writeable_swizzle__& sv)
```

Where **OP** is: **++**, **--**.

*Availability:* These are hidden friend functions only in **\_\_writeable\_swizzle\_\_**.

*Constraints:* Available only when the **\_\_writeable\_swizzle\_\_** view does not contain any repeated elements. Available only when **DataT** is not **bool**.

*Effects:* Perform an in-place element-wise **OP** prefix arithmetic operation on those elements of the **vec** object that have corresponding elements in the **sv** view. Elements in the underlying **vec** object that do not have elements in the **sv** view are not modified.

*Returns:* A reference to the **sv** view.

```
friend vec<DataT, NumElements> operatorOP(const __writeable_swizzle__& sv, int)
```

Where **OP** is: **++**, **--**.

*Availability:* These are hidden friend functions only in **\_\_writeable\_swizzle\_\_**.

*Constraints:* Available only when the **\_\_writeable\_swizzle\_\_** view does not contain any repeated elements. Available only when **DataT** is not **bool**.

*Effects:* Perform an in-place element-wise **OP** postfix arithmetic operation on those elements of the **vec** object that have corresponding elements in the **sv** view. Elements in the underlying **vec** object that do not have elements in the **sv** view are not modified.

*Returns:* A new **vec** object that represents the elements of **sv** after the swizzle operation is applied and before the postfix arithmetic operation is applied.

```
friend vec<DataT, NumElements> operatorOP(const __writeable_swizzle__& sv) (1)
friend vec<DataT, NumElements> operatorOP(const __const_swizzle__& sv) (2)
```

Where **OP** is: **+**, **-**.

*Availability:* Functions (1) are hidden friends in **\_\_writeable\_swizzle\_\_**. Functions (2) are hidden friends in **\_\_const\_swizzle\_\_**.

*Effects:* These functions behave as though the swizzle operation represented by the **sv** parameter was first evaluated into a temporary **vec** object, and then **operatorOP** was applied to the temporary **vec** object.

Returns: A **vec** object that represents the result of the operation.

```
template</*unspecified*/> (1)
friend vec<RET, NumElements>
operatorOP(const __writeable_swizzle__& lhs,
           const __writeable_swizzle__</*unspecified*/>& rhs)
```

```
template</*unspecified*/> (2)
friend vec<RET, NumElements>
operatorOP(const __writeable_swizzle__& lhs,
           const __const_swizzle__</*unspecified*/>& rhs)
```

```
template</*unspecified*/> (3)
friend vec<RET, NumElements>
operatorOP(const __const_swizzle__</*unspecified*/>& lhs,
           const __writeable_swizzle__& rhs)
```

```
template</*unspecified*/> (4)
friend vec<RET, NumElements>
operatorOP(const __const_swizzle__& lhs, const __const_swizzle__</*unspecified*/>& rhs)
```

```
friend vec<RET, NumElements> (5)
operatorOP(const vec<DataT, NumElements>& lhs, const __writeable_swizzle__& rhs)
```

```
friend vec<RET, NumElements> (6)
operatorOP(const vec<DataT, NumElements>& lhs, const __const_swizzle__& rhs)
```

```
friend vec<RET, NumElements> (7)
operatorOP(const __writeable_swizzle__& lhs, const vec<DataT, NumElements>& rhs)
```

```
friend vec<RET, NumElements> (8)
operatorOP(const __const_swizzle__& lhs, const vec<DataT, NumElements>& rhs)
```

```
template<typename T> (9)
friend vec<RET, NumElements>
operatorOP(const __writeable_swizzle__& lhs, const T& rhs)
```

```
template<typename T> (10)
friend vec<RET, NumElements>
operatorOP(const __const_swizzle__& lhs, const T& rhs)
```

```
template<typename T> (11)
friend vec<RET, NumElements>
operatorOP(const T& lhs, const __writeable_swizzle__& rhs)
```

```
template<typename T> (12)
friend vec<RET, NumElements>
operatorOP(const T& lhs, const __const_swizzle__& rhs)
```

Where **OP** is: **&&**, **||**, **==**, **!=**, **<**, **>**, **<=**, **>=**.

Availability: Overloads (1), (2), (3), (5), (7), (9), and (11) are hidden friends of **\_\_writeable\_swizzle\_\_**. Overloads (4), (6), (8), (10), and (12) are hidden friends of **\_\_const\_swizzle\_\_**.



**Constraints:** Overloads (1) - (4) are available only when the element data type of `lhs` is the same as the element data type of `rhs` and when the number of elements in the `lhs` view is equal to the number of elements in the `rhs` view.

Overloads (9) - (12) are available only when `T` is implicitly convertible to `DataT` and when `T` is not one of the swizzled vector class templates.

**Effects:** These functions behave as though the swizzle operation represented by each `__writeable_swizzle__` or `__const_swizzle__` parameter was first evaluated into a temporary `vec` object, and then `operatorOP` was called with the temporary `vec` object.

**Returns:** A `vec` object that represents the result of the operation.

```
friend vec<DataT, NumElements> operator~(const __writeable_swizzle__& sv) (1)
friend vec<DataT, NumElements> operator~(const __const_swizzle__& sv) (2)

friend vec<RET, NumElements> operator!(const __writeable_swizzle__& sv) (3)
friend vec<RET, NumElements> operator!(const __const_swizzle__& sv) (4)
```

**Availability:** Overloads (1) and (3) are hidden friends of `__writeable_swizzle__`. Overloads (2) and (4) are hidden friends of `__const_swizzle__`.

**Constraints:** Overloads (1) - (2) are available only when: `DataT != float && DataT != double && DataT != half`.

**Effects:** These functions behave as though the swizzle operation represented by the `__writeable_swizzle__` or `__const_swizzle__` parameter was first evaluated into a temporary `vec` object, and then the bitwise or logical NOT operation was applied to the temporary `vec` object.

**Returns:** A `vec` object that represents the result of the operation.

#### 4.14.2.5. Rounding modes

The various rounding modes that can be used in the `convert` member function template are described in [Table 125](#).

Table 125. Rounding modes for the SYCL `vec` class template

Rounding mode	Description
<code>automatic</code>	Default rounding mode for the SYCL <code>vec</code> class element type. <code>rtz</code> (round toward zero) for integer types and <code>rte</code> (round to nearest even) for floating-point types.
<code>rte</code>	Round to nearest even.
<code>rtz</code>	Round toward zero.
<code>rtp</code>	Round toward positive infinity.

Rounding mode	Description
rtn	Round toward negative infinity.

#### 4.14.2.6. Memory layout and alignment

The elements of an instance of the SYCL `vec` class template are stored in memory sequentially and contiguously and are aligned to the size of the element type in bytes multiplied by the number of elements:

$$\text{sizeof}(\text{DataT}) \cdot \text{NumElements}$$

The exception to this is when the number of element is three in which case the SYCL `vec` is aligned to the size of the element type in bytes multiplied by four:

$$\text{sizeof}(\text{DataT}) \cdot 4$$

This is true for both host and device code in order to allow for instances of the `vec` class template to be passed to SYCL kernel functions.

In no case, however, is the alignment guaranteed to be greater than 64 bytes.



The alignment guarantee is limited to 64 bytes because some host compilers (e.g. on Microsoft Windows) limit the maximum alignment of function parameters to this value.

#### 4.14.2.7. Performance note

The usage of the subscript `operator[]` may not be efficient on some devices.

### 4.14.3. Math array types

SYCL provides an `marray<typename DataT, std::size_t NumElements>` class template to represent a contiguous fixed-size container. This type allows sharing of containers between the host and its SYCL devices.

The `marray` class is templated on its element type and number of elements. The number of elements parameter, `NumElements`, is a positive value of the `std::size_t` type. The element type parameter, `DataT`, must be a *numeric type* as it is defined by C++ standard.

An instance of the `marray` class template can also be implicitly converted to an instance of the data type when the number of elements is `1` in order to allow single element arrays and scalars to be convertible with each other.

Logical and comparison operators for `marray` class template return `marray<bool, NumElements>`.

#### 4.14.3.1. Math array interface

The constructors, member functions and non-member functions of the SYCL `marray` class template are listed in [Table 126](#), [Table 127](#) and [Table 128](#) respectively.

```

1 namespace sycl {
2
3 template <typename DataT, std::size_t NumElements> class marray {
4 public:
5     using value_type = DataT;
6     using reference = DataT&
7     using const_reference = const DataT&
8     using iterator = DataT*;

```

```

9  using const_iterator = const DataT*;
10
11  marray();
12
13  explicit constexpr marray(const DataT& arg);
14
15  template <typename... ArgTN> constexpr marray(const ArgTN&... args);
16
17  constexpr marray(const marray<DataT, NumElements>& rhs);
18  constexpr marray(marray<DataT, NumElements>&& rhs);
19
20  // Available only when: NumElements == 1
21  operator DataT() const;
22
23  static constexpr std::size_t size() noexcept;
24
25  // subscript operator
26  reference operator[](std::size_t index);
27  const_reference operator[](std::size_t index) const;
28
29  marray& operator=(const marray<DataT, NumElements>& rhs);
30  marray& operator=(const DataT& rhs);
31
32  // iterator functions
33  iterator begin();
34  const_iterator begin() const;
35
36  iterator end();
37  const_iterator end() const;
38
39  // OP is: +, -, *, /, %
40  /* If OP is %, available only when: DataT != float && DataT != double && DataT
41   * != half. */
42  friend marray operatorOP(const marray& lhs, const marray& rhs) { /* ... */
43  }
44  friend marray operatorOP(const marray& lhs, const DataT& rhs) { /* ... */
45  }
46
47  // OP is: +=, -=, *=, /=, %=
48  /* If OP is %, available only when: DataT != float && DataT != double &&
49   * DataT != half. */
50  friend marray& operatorOP(marray& lhs, const marray& rhs) { /* ... */
51  }
52  friend marray& operatorOP(marray& lhs, const DataT& rhs) { /* ... */
53  }
54
55  // OP is prefix ++, --
56  friend marray& operatorOP(marray& v) { /* ... */
57  }
58
59  // OP is postfix ++, --
60  friend marray operatorOP(marray& v, int) { /* ... */
61  }
62
63  // OP is unary +, -
64  friend marray operatorOP(marray& v) { /* ... */

```

```

65 }
66
67 // OP is: &, |, ^
68 /* Available only when: DataT != float && DataT != double && DataT != half. */
69 friend marray operatorOP(const marray& lhs, const marray& rhs) { /* ... */
70 }
71 friend marray operatorOP(const marray& lhs, const DataT& rhs) { /* ... */
72 }
73
74 // OP is: &=, |=, ^=
75 /* Available only when: DataT != float && DataT != double && DataT != half. */
76 friend marray& operatorOP(marray& lhs, const marray& rhs) { /* ... */
77 }
78 friend marray& operatorOP(marray& lhs, const DataT& rhs) { /* ... */
79 }
80
81 // OP is: &&, ||
82 friend marray<bool, NumElements> operatorOP(const marray& lhs,
83                                             const marray& rhs) {
84 /* ... */ }
85 friend marray<bool, NumElements> operatorOP(const marray& lhs,
86                                             const DataT& rhs) {
87 /* ... */ }
88
89 // OP is: <<, >>
90 /* Available only when: DataT != float && DataT != double && DataT != half.
91 */
92 friend marray operatorOP(const marray& lhs, const marray& rhs) { /* ... */
93 }
94 friend marray operatorOP(const marray& lhs, const DataT& rhs) { /* ... */
95 }
96
97 // OP is: <=, >=
98 /* Available only when: DataT != float && DataT != double && DataT != half.
99 */
100 friend marray& operatorOP(marray& lhs, const marray& rhs) { /* ... */
101 }
102 friend marray& operatorOP(marray& lhs, const DataT& rhs) { /* ... */
103 }
104
105 // OP is: ==, !=, <, >, <=, >=
106 friend marray<bool, NumElements> operatorOP(const marray& lhs,
107                                             const marray& rhs) {
108 /* ... */ }
109 friend marray<bool, NumElements> operatorOP(const marray& lhs,
110                                             const DataT& rhs) {
111 /* ... */ }
112
113 /* Available only when: DataT != float && DataT != double && DataT != half.
114 */
115 friend marray operator~(const marray& v) { /* ... */
116 }
117
118 // OP is: +, -, *, /, %
119 /* operator% is only available when: DataT != float && DataT != double &&
120 * DataT != half. */

```

```

121 friend marray operatorOP(const DataT& lhs, const marray& rhs) { /* ... */
122 }
123
124 // OP is: &, |, ^
125 /* Available only when: DataT != float && DataT != double
126 && DataT != half. */
127 friend marray operatorOP(const DataT& lhs, const marray& rhs) { /* ... */
128 }
129
130 // OP is: &&, ||
131 friend marray<bool, NumElements> operatorOP(const DataT& lhs,
132                                             const marray& rhs) {
133 /* ... */ }
134
135 // OP is: <<, >>
136 /* Available only when: DataT != float && DataT != double && DataT != half.
137 */
138 friend marray operatorOP(const DataT& lhs, const marray& rhs) { /* ... */
139 }
140
141 // OP is: ==, !=, <, >, <=, >=
142 friend marray<bool, NumElements> operatorOP(const DataT& lhs,
143                                             const marray& rhs) {
144 /* ... */ }
145
146 friend marray<bool, NumElements> operator!(const marray& v) { /* ... */
147 }
148 };
149
150 } // namespace sycl

```

Table 126. Constructors of the SYCL `marray` class template

Constructor	Description
<code>marray()</code>	Default construct an array with element type <code>DataT</code> and with <code>NumElements</code> dimensions by default construction of each of its elements.
<code>explicit constexpr marray(const DataT&amp; arg)</code>	Construct an array of element type <code>DataT</code> and <code>NumElements</code> dimensions by setting each value to <code>arg</code> by assignment.
<code>template &lt;typename... ArgTN&gt; constexpr marray(const ArgTN &amp;... args)</code>	Construct a SYCL <code>marray</code> instance from any combination of scalar and SYCL <code>marray</code> parameters of the same element type, providing the total number of elements for all parameters sum to <code>NumElements</code> of this <code>marray</code> specialization.
<code>constexpr marray(const marray&lt;DataT, NumElements&gt;&amp; rhs)</code>	Construct an array of element type <code>DataT</code> and number of elements <code>NumElements</code> by copy from another similar vector.

Constructor	Description
<code>constexpr marray(marray&lt;DataT, NumElements&gt;&amp;&amp; rhs)</code>	Construct an array of element type <code>DataT</code> and number of elements <code>NumElements</code> by moving from another similar vector.

Table 127. Member functions for the SYCL `marray` class template

Member function	Description
<code>operator DataT() const</code>	Available only when: <code>NumElements == 1</code> .  Converts this SYCL <code>marray</code> instance to an instance of <code>DataT</code> with the value of the single element in this SYCL <code>marray</code> instance.  The SYCL <code>marray</code> instance shall be implicitly convertible to the same data types, to which <code>DataT</code> is implicitly convertible. Note that conversion operator shall not be templated to allow standard conversion sequence for implicit conversion.
<code>static constexpr std::size_t size() noexcept</code>	Returns the number of elements in this SYCL <code>marray</code> (i.e., <code>NumElements</code> ).
<code>DataT&amp; operator[](std::size_t index)</code>	Returns a reference to the element stored within this SYCL <code>marray</code> at the index specified by <code>index</code> .
<code>const DataT&amp; operator[](std::size_t index) const</code>	Returns a const reference to the element stored within this SYCL <code>marray</code> at the index specified by <code>index</code> .
<code>marray&amp; operator=(const marray&amp; rhs)</code>	Assign each element of the <code>rhs</code> SYCL <code>marray</code> to each element of this SYCL <code>marray</code> and return a reference to this SYCL <code>marray</code> .
<code>marray&amp; operator=(const DataT&amp; rhs)</code>	Assign each element of the <code>rhs</code> scalar to each element of this SYCL <code>marray</code> and return a reference to this SYCL <code>marray</code> .
<code>iterator begin()</code>	Returns an iterator referring to the first element stored within the <code>marray</code> .
<code>const_iterator begin() const</code>	Returns a const iterator referring to the first element stored within the <code>marray</code> .
<code>iterator end()</code>	Returns an iterator referring to the one past the last element stored within the <code>marray</code> .

Member function	Description
<code>const_iterator end() const</code>	Returns a const iterator referring to the one past the last element stored within the <code>marray</code> .

Table 128. Hidden friend functions of the `marray` class template

Hidden friend function	Description
<code>marray operatorOP(const marray&amp; lhs, const marray&amp; rhs)</code>	<p>If <code>OP</code> is <code>%</code>, available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</p> <p>Construct a new instance of the SYCL <code>marray</code> class template with the same template parameters as <code>lhs marray</code> with each element of the new SYCL <code>marray</code> instance the result of an element-wise <code>OP</code> arithmetic operation between each element of <code>lhs marray</code> and each element of the <code>rhs</code> SYCL <code>marray</code>.</p> <p>Where <code>OP</code> is: <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%</code>.</p>
<code>marray operatorOP(const marray&amp; lhs, const DataT&amp; rhs)</code>	<p>If <code>OP</code> is <code>%</code>, available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</p> <p>Construct a new instance of the SYCL <code>marray</code> class template with the same template parameters as <code>lhs marray</code> with each element of the new SYCL <code>marray</code> instance the result of an element-wise <code>OP</code> arithmetic operation between each element of <code>lhs marray</code> and the <code>rhs</code> scalar.</p> <p>Where <code>OP</code> is: <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%</code>.</p>
<code>marray&amp; operatorOP(marray&amp; lhs, const marray&amp; rhs)</code>	<p>If <code>OP</code> is <code>%=</code>, available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</p> <p>Perform an in-place element-wise <code>OP</code> arithmetic operation between each element of <code>lhs marray</code> and each element of the <code>rhs</code> SYCL <code>marray</code> and return <code>lhs marray</code>.</p> <p>Where <code>OP</code> is: <code>+=</code>, <code>-=</code>, <code>*=</code>, <code>/=</code>, <code>%=</code>.</p>

Hidden friend function	Description
<code>marray&amp; operatorOP(marray&amp; lhs, const DataT&amp; rhs)</code>	<p>If <b>OP</b> is <b>%=</b>, available only when: <b>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</b>.</p> <p>Perform an in-place element-wise <b>OP</b> arithmetic operation between each element of <b>lhs marray</b> and <b>rhs</b> scalar and return <b>lhs marray</b>.</p> <p>Where <b>OP</b> is: <b>+=, -=, *=, /=, %=</b>.</p>
<code>marray&amp; operatorOP(marray&amp; v)</code>	<p>Perform an in-place element-wise <b>OP</b> prefix arithmetic operation on each element of <b>v marray</b>, assigning the result of each element to the corresponding element of <b>v marray</b> and return <b>v marray</b>.</p> <p>Where <b>OP</b> is: <b>++, --</b>.</p>
<code>marray operatorOP(marray&amp; v, int)</code>	<p>Perform an in-place element-wise <b>OP</b> postfix arithmetic operation on each element of <b>v marray</b>, assigning the result of each element to the corresponding element of <b>v marray</b> and returns a copy of <b>v marray</b> before the operation is performed.</p> <p>Where <b>OP</b> is: <b>++, --</b>.</p>
<code>marray operatorOP(marray&amp; v)</code>	<p>Construct a new instance of the SYCL <b>marray</b> class template with the same template parameters as this SYCL <b>marray</b> with each element of the new SYCL <b>marray</b> instance the result of an element-wise <b>OP</b> unary arithmetic operation on each element of this SYCL <b>marray</b>.</p> <p>Where <b>OP</b> is: <b>+, -</b>.</p>
<code>marray operatorOP(const marray&amp; lhs, const marray&amp; rhs)</code>	<p>Available only when: <b>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</b>.</p> <p>Construct a new instance of the SYCL <b>marray</b> class template with the same template parameters as <b>lhs marray</b> with each element of the new SYCL <b>marray</b> instance the result of an element-wise <b>OP</b> bit-wise operation between each element of <b>lhs marray</b> and each element of the <b>rhs</b> SYCL <b>marray</b>.</p> <p>Where <b>OP</b> is: <b>&amp;,  , ^</b>.</p>



Hidden friend function	Description
<pre data-bbox="161 185 887 219">marray operatorOP(const marray&amp; lhs, const DataT&amp; rhs)</pre>	<p data-bbox="1002 159 1460 255">Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</p> <p data-bbox="1002 288 1460 607">Construct a new instance of the SYCL <code>marray</code> class template with the same template parameters as <code>lhs marray</code> with each element of the new SYCL <code>marray</code> instance the result of an element-wise <code>OP</code> bit-wise operation between each element of <code>lhs marray</code> and the <code>rhs</code> scalar.</p> <p data-bbox="1002 640 1241 674">Where <code>OP</code> is: <code>&amp;,  , ^</code>.</p>
<pre data-bbox="161 719 831 752">marray&amp; operatorOP(marray&amp; lhs, const marray&amp; rhs)</pre>	<p data-bbox="1002 692 1460 788">Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</p> <p data-bbox="1002 822 1460 960">Perform an in-place element-wise <code>OP</code> bitwise operation between each element of <code>lhs marray</code> and the <code>rhs</code> SYCL <code>marray</code> and return <code>lhs marray</code>.</p> <p data-bbox="1002 994 1283 1028">Where <code>OP</code> is: <code>&amp;=,  =, ^=</code>.</p>
<pre data-bbox="161 1070 820 1104">marray&amp; operatorOP(marray&amp; lhs, const DataT&amp; rhs)</pre>	<p data-bbox="1002 1043 1460 1140">Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</p> <p data-bbox="1002 1173 1460 1312">Perform an in-place element-wise <code>OP</code> bitwise operation between each element of <code>lhs marray</code> and the <code>rhs</code> scalar and return a <code>lhs marray</code>.</p> <p data-bbox="1002 1346 1283 1379">Where <code>OP</code> is: <code>&amp;=,  =, ^=</code>.</p>
<pre data-bbox="161 1422 898 1489">marray&lt;bool, NumElements&gt; operatorOP(const marray&amp; lhs, const marray&amp; rhs)</pre>	<p data-bbox="1002 1395 1460 1713">Construct a new instance of the <code>marray</code> class template with <code>DataT = bool</code> and same <code>NumElements</code> as <code>lhs marray</code> with each element of the new <code>marray</code> instance the result of an element-wise <code>OP</code> logical operation between each element of <code>lhs marray</code> and each element of the <code>rhs marray</code>.</p> <p data-bbox="1002 1747 1241 1780">Where <code>OP</code> is: <code>&amp;&amp;,   </code>.</p>

Hidden friend function	Description
<pre data-bbox="161 188 900 253">marray&lt;bool, NumElements&gt; operatorOP(const marray&amp; lhs, const DataT&amp; rhs)</pre>	<p data-bbox="1002 159 1460 448">Construct a new instance of the <code>marray</code> class template with <code>DataT = bool</code> and same <code>NumElements</code> as <code>lhs marray</code> with each element of the new <code>marray</code> instance the result of an element-wise <code>OP</code> logical operation between each element of <code>lhs marray</code> and the <code>rhs</code> scalar.</p> <p data-bbox="1002 477 1241 510">Where <code>OP</code> is: <code>&amp;&amp;,   </code>.</p>
<pre data-bbox="161 557 900 591">marray operatorOP(const marray&amp; lhs, const marray&amp; rhs)</pre>	<p data-bbox="1002 528 1460 627">Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</p> <p data-bbox="1002 656 1460 1232">Construct a new instance of the SYCL <code>marray</code> class template with the same template parameters as <code>lhs marray</code> with each element of the new SYCL <code>marray</code> instance the result of an element-wise <code>OP</code> bit-shift operation between each element of <code>lhs marray</code> and each element of the <code>rhs</code> SYCL <code>marray</code>. If <code>OP</code> is <code>&gt;&gt;</code>, <code>DataT</code> is a signed type and <code>lhs marray</code> has a negative value any vacated bits viewed as an unsigned integer must be assigned the value <code>1</code>, otherwise any vacated bits viewed as an unsigned integer must be assigned the value <code>0</code>.</p> <p data-bbox="1002 1261 1241 1294">Where <code>OP</code> is: <code>&lt;&lt;, &gt;&gt;</code>.</p>
<pre data-bbox="161 1341 887 1375">marray operatorOP(const marray&amp; lhs, const DataT&amp; rhs)</pre>	<p data-bbox="1002 1312 1460 1411">Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</p> <p data-bbox="1002 1440 1460 2016">Construct a new instance of the SYCL <code>marray</code> class template with the same template parameters as <code>lhs marray</code> with each element of the new SYCL <code>marray</code> instance the result of an element-wise <code>OP</code> bit-shift operation between each element of <code>lhs marray</code> and the <code>rhs</code> scalar. If <code>OP</code> is <code>&gt;&gt;</code>, <code>DataT</code> is a signed type and <code>lhs marray</code> has a negative value any vacated bits viewed as an unsigned integer must be assigned the value <code>1</code>, otherwise any vacated bits viewed as an unsigned integer must be assigned the value <code>0</code>.</p> <p data-bbox="1002 2045 1241 2078">Where <code>OP</code> is: <code>&lt;&lt;, &gt;&gt;</code>.</p>

Hidden friend function	Description
<pre data-bbox="161 185 831 219">marray&amp; operatorOP(marray&amp; lhs, const marray&amp; rhs)</pre>	<p data-bbox="1002 159 1461 253">Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</p> <p data-bbox="1002 286 1461 678">Perform an in-place element-wise <code>OP</code> bitshift operation between each element of <code>lhs marray</code> and the <code>rhs SYCL marray</code> and returns <code>lhs marray</code>. If <code>OP</code> is <code>&gt;&gt;=</code>, <code>DataT</code> is a signed type and <code>lhs marray</code> has a negative value any vacated bits viewed as an unsigned integer must be assigned the value <code>1</code>, otherwise any vacated bits viewed as an unsigned integer must be assigned the value <code>0</code>.</p> <p data-bbox="1002 712 1270 745">Where <code>OP</code> is: <code>&lt;&lt;=</code>, <code>&gt;&gt;=</code>.</p>
<pre data-bbox="161 790 818 824">marray&amp; operatorOP(marray&amp; lhs, const DataT&amp; rhs)</pre>	<p data-bbox="1002 763 1461 857">Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</p> <p data-bbox="1002 891 1461 1323">Perform an in-place element-wise <code>OP</code> bitshift operation between each element of <code>lhs marray</code> and the <code>rhs scalar</code> and returns a reference to this SYCL <code>marray</code>. If <code>OP</code> is <code>&gt;&gt;=</code>, <code>DataT</code> is a signed type and <code>lhs marray</code> has a negative value any vacated bits viewed as an unsigned integer must be assigned the value <code>1</code>, otherwise any vacated bits viewed as an unsigned integer must be assigned the value <code>0</code>.</p> <p data-bbox="1002 1357 1270 1391">Where <code>OP</code> is: <code>&lt;&lt;=</code>, <code>&gt;&gt;=</code>.</p>
<pre data-bbox="161 1429 898 1503">marray&lt;bool, NumElements&gt; operatorOP(const marray&amp; lhs, const marray&amp; rhs)</pre>	<p data-bbox="1002 1402 1461 1906">Construct a new instance of the <code>marray</code> class template with <code>DataT = bool</code> and same <code>NumElements</code> as <code>lhs marray</code> with each element of the new <code>marray</code> instance is the result of an element-wise <code>OP</code> relational operation between each element of <code>lhs marray</code> and each element of the <code>rhs marray</code>. The <code>==</code>, <code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code> and <code>&gt;=</code> operations result in <code>false</code> if either the <code>lhs</code> element or the <code>rhs</code> element is a NaN. The <code>!=</code> operation results in <code>true</code> if either the <code>lhs</code> element or the <code>rhs</code> element is a NaN.</p> <p data-bbox="1002 1939 1377 1973">Where <code>OP</code> is: <code>==</code>, <code>!=</code>, <code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code>, <code>&gt;=</code>.</p>

Hidden friend function	Description
<pre data-bbox="161 185 900 253">marray&lt;bool, NumElements&gt; operatorOP(const marray&amp; lhs, const DataT&amp; rhs)</pre>	<p data-bbox="1002 159 1460 622">Construct a new instance of the <code>marray</code> class template with <code>DataT = bool</code> and same <code>NumElements</code> as <code>lhs marray</code> with each element of the new <code>marray</code> instance the result of an element-wise <code>OP</code> relational operation between each element of <code>lhs marray</code> and the <code>rhs</code> scalar. The <code>==</code>, <code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code> and <code>&gt;=</code> operations result in <code>false</code> if either the <code>lhs</code> element or the <code>rhs</code> is a NaN. The <code>!=</code> operation results in <code>true</code> if either the <code>lhs</code> element or the <code>rhs</code> is a NaN.</p> <p data-bbox="1002 656 1378 685">Where <code>OP</code> is: <code>==</code>, <code>!=</code>, <code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code>, <code>&gt;=</code>.</p>
<pre data-bbox="161 734 887 763">marray operatorOP(const DataT&amp; lhs, const marray&amp; rhs)</pre>	<p data-bbox="1002 707 1460 804">If <code>OP</code> is <code>%</code>, available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</p> <p data-bbox="1002 837 1460 1155">Construct a new instance of the SYCL <code>marray</code> class template with the same template parameters as the <code>rhs</code> SYCL <code>marray</code> with each element of the new SYCL <code>marray</code> instance the result of an element-wise <code>OP</code> arithmetic operation between the <code>lhs</code> scalar and each element of the <code>rhs</code> SYCL <code>marray</code>.</p> <p data-bbox="1002 1189 1299 1218">Where <code>OP</code> is: <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%</code>.</p>
<pre data-bbox="161 1265 887 1294">marray operatorOP(const DataT&amp; lhs, const marray&amp; rhs)</pre>	<p data-bbox="1002 1238 1460 1335">Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>.</p> <p data-bbox="1002 1368 1460 1686">Construct a new instance of the SYCL <code>marray</code> class template with the same template parameters as the <code>rhs</code> SYCL <code>marray</code> with each element of the new SYCL <code>marray</code> instance the result of an element-wise <code>OP</code> bitwise operation between the <code>lhs</code> scalar and each element of the <code>rhs</code> SYCL <code>marray</code>.</p> <p data-bbox="1002 1720 1243 1749">Where <code>OP</code> is: <code>&amp;</code>, <code> </code>, <code>^</code>.</p>

Hidden friend function	Description
<pre data-bbox="161 185 884 253">marray&lt;bool, NumElements&gt; operatorOP(const DataT&amp; lhs, const marray&amp; rhs)</pre>	<p data-bbox="1002 159 1460 443">Construct a new instance of the <code>marray</code> class template with <code>DataT = bool</code> and same <code>NumElements</code> as <code>rhs marray</code> with each element of the new <code>marray</code> instance the result of an element-wise <code>OP</code> logical operation between the <code>lhs</code> scalar and each element of the <code>rhs marray</code>.</p> <p data-bbox="1002 472 1241 506">Where <code>OP</code> is: <code>&amp;&amp;</code>, <code>  </code>.</p>
<pre data-bbox="161 555 884 589">marray operatorOP(const DataT&amp; lhs, const marray&amp; rhs)</pre>	<p data-bbox="1002 528 1460 1099">Construct a new instance of the SYCL <code>marray</code> class template with the same template parameters as the <code>rhs</code> SYCL <code>marray</code> with each element of the new SYCL <code>marray</code> instance the result of an element-wise <code>OP</code> bitshift operation between the <code>lhs</code> scalar and each element of the <code>rhs</code> SYCL <code>marray</code>. If <code>OP</code> is <code>&gt;&gt;</code>, <code>DataT</code> is a signed type and this SYCL <code>marray</code> has a negative value any vacated bits viewed as an unsigned integer must be assigned the value <code>1</code>, otherwise any vacated bits viewed as an unsigned integer must be assigned the value <code>0</code>.</p> <p data-bbox="1002 1128 1241 1162">Where <code>OP</code> is: <code>&lt;&lt;</code>, <code>&gt;&gt;</code>.</p>
<pre data-bbox="161 1211 884 1279">marray&lt;bool, NumElements&gt; operatorOP(const DataT&amp; lhs, const marray&amp; rhs)</pre>	<p data-bbox="1002 1184 1460 1682">Construct a new instance of the <code>marray</code> class template with <code>DataT = bool</code> and same <code>NumElements</code> as <code>rhs marray</code> with each element of the new SYCL <code>marray</code> instance the result of an element-wise <code>OP</code> relational operation between the <code>lhs</code> scalar and each element of the <code>rhs marray</code>. The <code>==</code>, <code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code> and <code>&gt;=</code> operations result in <code>false</code> if either the <code>lhs</code> or the <code>rhs</code> element is a NaN. The <code>!=</code> operation results in <code>true</code> if either the <code>lhs</code> or the <code>rhs</code> element is a NaN.</p> <p data-bbox="1002 1711 1377 1744">Where <code>OP</code> is: <code>==</code>, <code>!=</code>, <code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code>, <code>&gt;=</code>.</p>

Hidden friend function	Description
<code>marray&amp; operator~(const marray&amp; v)</code>	Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code> .  Construct a new instance of the SYCL <code>marray</code> class template with the same template parameters as <code>v</code> <code>marray</code> with each element of the new SYCL <code>marray</code> instance the result of an element-wise <code>OP</code> bit-wise operation on each element of <code>v marray</code> .
<code>marray&lt;bool, NumElements&gt; operator!(const marray&amp; v)</code>	Construct a new instance of the <code>marray</code> class template with <code>DataT = bool</code> and same <code>NumElements</code> as <code>v marray</code> with each element of the new <code>marray</code> instance the result of an element-wise logical <code>!</code> operation on each element of <code>v marray</code> .

#### 4.14.3.2. Aliases

The SYCL programming API provides all permutations of the type alias:

```
using m<type><elems> = marray<<storage-type>, <elems>>
```

where `<elems>` is 2, 3, 4, 8 and 16, and pairings of `<type>` and `<storage-type>` for integral types are `char` and `int8_t`, `uchar` and `uint8_t`, `short` and `int16_t`, `ushort` and `uint16_t`, `int` and `int32_t`, `uint` and `uint32_t`, `long` and `int64_t`, `ulong` and `uint64_t`, for floating point types are both `half`, `float` and `double`, and for boolean type `bool`.

For example `muint4` is the alias to `marray<uint32_t, 4>` and `mfloat16` is the alias to `marray<float, 16>`.

#### 4.14.3.3. Memory layout and alignment

The elements of an instance of the `marray` class template as if stored in `std::array<DataT, NumElements>`.

## 4.15. Synchronization and atomics

The available features are:

- Accessor classes: Accessor classes specify acquisition and release of buffer and image data structures to provide points at which a SYCL runtime must guarantee memory consistency.
- Atomic operations: SYCL devices support a restricted subset of C++ atomics and SYCL uses the library syntax from the next C++ specification to make this available.
- Fences: Fence primitives are made available to order loads and stores. They are exposed through the `atomic_fence` function. Fences can have acquire semantics, release semantics or both.
- Barriers: Barrier primitives are made available as a coordination mechanism for work-items within individual `groups`. They are exposed through the `group_barrier` function.
- Hierarchical parallel dispatch: In the hierarchical parallelism model of describing computations, work-items within a work-group may coordinate via multiple instances of the `parallel_for_work_item` function call, rather than through the use of explicit `work-group barrier` operations.
- Device event: they are used inside SYCL kernel functions to wait for asynchronous operations within

a SYCL kernel function to complete.

### 4.15.1. Barriers and fences

A [group barrier](#) or [mem-fence](#) provides memory ordering semantics over both the local address space and global address space. A [mem-fence](#) provides control over the re-ordering of memory load and store operations, subject to the associated memory [order](#) and memory [scope](#), when paired with synchronization through an atomic object.

```
1 namespace sycl {
2
3 void atomic_fence(memory_order order, memory_scope scope);
4
5 } // namespace sycl
```

The effects of a call to `atomic_fence` depend on the value of the `order` parameter:

- `memory_order::relaxed`: No effect
- `memory_order::acquire`: Acquire fence
- `memory_order::release`: Release fence
- `memory_order::acq_rel`: Both an acquire fence and a release fence
- `memory_order::seq_cst`: A sequentially consistent acquire and release fence

A [group barrier](#) acts as both an acquire fence and a release fence: all work-items in the group execute a release fence prior to signaling arrival at the barrier, and all work-items in the group execute an acquire fence afterwards. A [group barrier](#) provides implicit atomic synchronization as if through an internal atomic object, such that the acquire and release fences associated with the barrier synchronize with each other, without an explicit atomic operation being required on an atomic object to synchronize the fences.

### 4.15.2. `device_event` class

The SYCL `device_event` class encapsulates a single SYCL device event which is available only within SYCL kernel functions and can be used to wait for asynchronous operations within a SYCL kernel function to complete.

All member functions of the `device_event` class must not throw a SYCL exception.

A synopsis of the SYCL `device_event` class is provided below. The constructors and member functions of the SYCL `device_event` class are listed in [Table 130](#) and [Table 129](#) respectively.

```
1 namespace sycl {
2 class device_event {
3
4     device_event(__unspecified__);
5
6 public:
7     void wait() noexcept;
8 };
9 } // namespace sycl
```

Table 129. Member functions of the SYCL `device_event` class

Member function	Description
<code>void wait() noexcept</code>	Waits for the asynchronous operation associated with this SYCL <code>device_event</code> to complete.

Table 130. Constructors of the `device_event` class

Constructor	Description
<code>device_event(____unspecified____)</code>	Unspecified implementation-defined constructor.

### 4.15.3. Atomic references

The `sycl::atomic_ref` class provides the ability to perform atomic operations in device code with a syntax similar to the C++ standard `std::atomic_ref`. The `sycl::atomic_ref` class must not be used in host code.

Unlike `std::atomic_ref`, `sycl::atomic_ref` does not provide a default memory ordering for its operations. Instead, the application must specify a default ordering via the `DefaultOrder` template parameter. This ordering is used as a default for most of the atomic operations, but most member functions also provide an optional parameter that allows the application to override this default. The set of supported orderings is specific to a device, but every device is guaranteed to support at least `memory_order::relaxed`. If the default order is set to `memory_order::relaxed`, all memory order arguments default to `memory_order::relaxed`. If the default order is set to `memory_order::acq_rel`, memory order arguments default to `memory_order::acquire` for load operations, `memory_order::release` for store operations and `memory_order::acq_rel` for read-modify-write operations. If the default order is set to `memory_order::seq_cst`, all memory order arguments default to `memory_order::seq_cst`.

The `sycl::atomic_ref` class has a template parameter `DefaultScope`, which allows the application to define a default memory scope for the atomic operations. Most member functions also provide an optional parameter that allows the application to override this default.

The `sycl::atomic_ref` class also has a template parameter `AddressSpace`, which allows the application to make an assertion about the address space of the object of type `T` that it references. The default value for this parameter is `access::address_space::generic_space`, which indicates that the object could be in either the global or local address spaces. If the application knows the address space, it can set this template parameter to either `access::address_space::global_space` or `access::address_space::local_space` as an assertion to the implementation. Specifying the address space via this template parameter may allow the implementation to perform certain optimizations. Specifying an address space that does not match the object's actual address space results in undefined behavior.

The template parameter `T` must be one of the following types:

- `int`,
- `unsigned int`,
- `long`,
- `unsigned long`,
- `long long`,
- `unsigned long long`,
- `float`,
- `double`, or
- Any pointer-to-object type.



In addition, the type `T` must satisfy one of the following conditions:

- `sizeof(T) == 4`, or
- `sizeof(T) == 8` and the code containing this `atomic_ref` was submitted to a device that has `aspect::atomic64`.

For floating-point types, the member functions of the `atomic_ref` class may be emulated, and they may use a different floating-point environment from those defined by `info::device::single_fp_config` and `info::device::double_fp_config` (i.e. floating-point atomics may use different rounding modes and may have different exception behavior).

The atomic types are defined as follows.

```

1 namespace sycl {
2
3 // Exposition only
4 template <memory_order ReadModifyWriteOrder> struct memory_order_traits;
5
6 template <> struct memory_order_traits<memory_order::relaxed> {
7     static constexpr memory_order read_order = memory_order::relaxed;
8     static constexpr memory_order write_order = memory_order::relaxed;
9 };
10
11 template <> struct memory_order_traits<memory_order::acq_rel> {
12     static constexpr memory_order read_order = memory_order::acquire;
13     static constexpr memory_order write_order = memory_order::release;
14 };
15
16 template <> struct memory_order_traits<memory_order::seq_cst> {
17     static constexpr memory_order read_order = memory_order::seq_cst;
18     static constexpr memory_order write_order = memory_order::seq_cst;
19 };
20
21 template <typename T, memory_order DefaultOrder, memory_scope DefaultScope,
22         access::address_space AddressSpace = access::address_space::generic_space>
23 class atomic_ref {
24 public:
25     using value_type = T;
26     static constexpr size_t required_alignment = /* implementation-defined */;
27     static constexpr bool is_always_lock_free = /* implementation-defined */;
28     static constexpr memory_order default_read_order =
29         memory_order_traits<DefaultOrder>::read_order;
30     static constexpr memory_order default_write_order =
31         memory_order_traits<DefaultOrder>::write_order;
32     static constexpr memory_order default_read_modify_write_order = DefaultOrder;
33     static constexpr memory_scope default_scope = DefaultScope;
34
35     bool is_lock_free() const noexcept;
36
37     explicit atomic_ref(T&);
38     atomic_ref(const atomic_ref&) noexcept;
39     atomic_ref& operator=(const atomic_ref&) = delete;
40
41     void store(T operand, memory_order order = default_write_order,
42             memory_scope scope = default_scope) const noexcept;
43

```

```

44 T operator=(T desired) const noexcept;
45
46 T load(memory_order order = default_read_order,
47        memory_scope scope = default_scope) const noexcept;
48
49 operator T() const noexcept;
50
51 T exchange(T operand, memory_order order = default_read_modify_write_order,
52            memory_scope scope = default_scope) const noexcept;
53
54 bool compare_exchange_weak(T& expected, T desired, memory_order success,
55                             memory_order failure,
56                             memory_scope scope = default_scope) const noexcept;
57
58 bool
59 compare_exchange_weak(T& expected, T desired,
60                       memory_order order = default_read_modify_write_order,
61                       memory_scope scope = default_scope) const noexcept;
62
63 bool
64 compare_exchange_strong(T& expected, T desired, memory_order success,
65                         memory_order failure,
66                         memory_scope scope = default_scope) const noexcept;
67
68 bool
69 compare_exchange_strong(T& expected, T desired,
70                         memory_order order = default_read_modify_write_order,
71                         memory_scope scope = default_scope) const noexcept;
72 };
73
74 // Partial specialization for integral types
75 template <memory_order DefaultOrder, memory_scope DefaultScope,
76           access::address_space AddressSpace = access::address_space::generic_space>
77 class atomic_ref<Integral, DefaultOrder, DefaultScope, AddressSpace> {
78
79     /* All other members from atomic_ref<T> are available */
80
81     using difference_type = value_type;
82
83     Integral fetch_add(Integral operand,
84                       memory_order order = default_read_modify_write_order,
85                       memory_scope scope = default_scope) const noexcept;
86
87     Integral fetch_sub(Integral operand,
88                       memory_order order = default_read_modify_write_order,
89                       memory_scope scope = default_scope) const noexcept;
90
91     Integral fetch_and(Integral operand,
92                       memory_order order = default_read_modify_write_order,
93                       memory_scope scope = default_scope) const noexcept;
94
95     Integral fetch_or(Integral operand,
96                      memory_order order = default_read_modify_write_order,
97                      memory_scope scope = default_scope) const noexcept;
98
99     Integral fetch_xor(Integral operand,

```

```

100         memory_order order = default_read_modify_write_order,
101         memory_scope scope = default_scope) const noexcept;
102
103 Integral fetch_min(Integral operand,
104         memory_order order = default_read_modify_write_order,
105         memory_scope scope = default_scope) const noexcept;
106
107 Integral fetch_max(Integral operand,
108         memory_order order = default_read_modify_write_order,
109         memory_scope scope = default_scope) const noexcept;
110
111 Integral operator++(int) const noexcept;
112 Integral operator--(int) const noexcept;
113 Integral operator++() const noexcept;
114 Integral operator--() const noexcept;
115 Integral operator+=(Integral) const noexcept;
116 Integral operator-=(Integral) const noexcept;
117 Integral operator&=(Integral) const noexcept;
118 Integral operator|=(Integral) const noexcept;
119 Integral operator^=(Integral) const noexcept;
120 };
121
122 // Partial specialization for floating-point types
123 template <memory_order DefaultOrder, memory_scope DefaultScope,
124         access::address_space AddressSpace = access::address_space::generic_space>
125 class atomic_ref<Floating, DefaultOrder, DefaultScope, AddressSpace> {
126
127     /* All other members from atomic_ref<T> are available */
128
129     using difference_type = value_type;
130
131     Floating fetch_add(Floating operand,
132         memory_order order = default_read_modify_write_order,
133         memory_scope scope = default_scope) const noexcept;
134
135     Floating fetch_sub(Floating operand,
136         memory_order order = default_read_modify_write_order,
137         memory_scope scope = default_scope) const noexcept;
138
139     Floating fetch_min(Floating operand,
140         memory_order order = default_read_modify_write_order,
141         memory_scope scope = default_scope) const noexcept;
142
143     Floating fetch_max(Floating operand,
144         memory_order order = default_read_modify_write_order,
145         memory_scope scope = default_scope) const noexcept;
146
147     Floating operator+=(Floating) const noexcept;
148     Floating operator-=(Floating) const noexcept;
149 };
150
151 // Partial specialization for pointers
152 template <typename T, memory_order DefaultOrder, memory_scope DefaultScope,
153         access::address_space AddressSpace = access::address_space::generic_space>
154 class atomic_ref<T*, DefaultOrder, DefaultScope, AddressSpace> {
155

```

```

156 using value_type = T*;
157 using difference_type = ptrdiff_t;
158 static constexpr size_t required_alignment = /* implementation-defined */;
159 static constexpr bool is_always_lock_free = /* implementation-defined */;
160 static constexpr memory_order default_read_order =
161     memory_order_traits<DefaultOrder>::read_order;
162 static constexpr memory_order default_write_order =
163     memory_order_traits<DefaultOrder>::write_order;
164 static constexpr memory_order default_read_modify_write_order = DefaultOrder;
165 static constexpr memory_scope default_scope = DefaultScope;
166
167 bool is_lock_free() const noexcept;
168
169 explicit atomic_ref(T*&);
170 atomic_ref(const atomic_ref&) noexcept;
171 atomic_ref& operator=(const atomic_ref&) = delete;
172
173 void store(T* operand, memory_order order = default_write_order,
174     memory_scope scope = default_scope) const noexcept;
175
176 T* operator=(T* desired) const noexcept;
177
178 T* load(memory_order order = default_read_order,
179     memory_scope scope = default_scope) const noexcept;
180
181 operator T*() const noexcept;
182
183 T* exchange(T* operand, memory_order order = default_read_modify_write_order,
184     memory_scope scope = default_scope) const noexcept;
185
186 bool compare_exchange_weak(T*& expected, T* desired, memory_order success,
187     memory_order failure,
188     memory_scope scope = default_scope) const noexcept;
189
190 bool
191 compare_exchange_weak(T*& expected, T* desired,
192     memory_order order = default_read_modify_write_order,
193     memory_scope scope = default_scope) const noexcept;
194
195 bool
196 compare_exchange_strong(T*& expected, T* desired, memory_order success,
197     memory_order failure,
198     memory_scope scope = default_scope) const noexcept;
199
200 bool
201 compare_exchange_strong(T*& expected, T* desired,
202     memory_order order = default_read_modify_write_order,
203     memory_scope scope = default_scope) const noexcept;
204
205 T* fetch_add(difference_type,
206     memory_order order = default_read_modify_write_order,
207     memory_scope scope = default_scope) const noexcept;
208
209 T* fetch_sub(difference_type,
210     memory_order order = default_read_modify_write_order,
211     memory_scope scope = default_scope) const noexcept;

```

```

212
213  T* operator++(int) const noexcept;
214  T* operator--(int) const noexcept;
215  T* operator++() const noexcept;
216  T* operator--() const noexcept;
217  T* operator+=(difference_type) const noexcept;
218  T* operator-=(difference_type) const noexcept;
219 };
220
221 } // namespace sycl

```

The constructors and member functions for instances of the SYCL `atomic_ref` class using any compatible type are listed in Table 131 and Table 132 respectively. Additional member functions for integral, floating-point and pointer types are listed in Table 133, Table 134 and Table 135 respectively.

The static member `required_alignment` describes the minimum required alignment in bytes of an object that can be referenced by an `atomic_ref<T>`, which must be at least `alignof(T)`.

The static member `is_always_lock_free` is true if all atomic operations for type `T` are always lock-free. A SYCL implementation is not guaranteed to support atomic operations that are not lock-free.

The static members `default_read_order`, `default_write_order` and `default_read_modify_write_order` reflect the default memory order values for each type of atomic operation, consistent with the `DefaultOrder` template.

The atomic operations and member functions behave as described in the C++ specification, barring the restrictions discussed above.



Care must be taken when using atomics for work-item coordination, because work-items are not required to provide stronger than weakly parallel forward progress guarantees. Operations that block a work-item, such as continuously checking the value of an atomic variable until some condition holds, or using atomic operations that are not lock-free, may prevent overall progress.

Table 131. Constructors of the SYCL `atomic_ref` class template

Constructor	Description
<code>atomic_ref(T&amp; ref)</code>	Constructs an instance of SYCL <code>atomic_ref</code> which is associated with the reference <code>ref</code> .

Table 132. Member functions available on any object of type `atomic_ref<T>`

Member function	Description
<code>bool is_lock_free() const</code>	Return <code>true</code> if the atomic operations provided by this <code>atomic_ref</code> are lock-free.

Member function	Description
<pre>void store(T operand, memory_order order = default_write_order, memory_scope scope = default_scope) const</pre>	Atomically stores <code>operand</code> to the object referenced by this <code>atomic_ref</code> . The memory order of this atomic operation must be <code>memory_order::relaxed</code> , <code>memory_order::release</code> or <code>memory_order::seq_cst</code> . This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code> .
<pre>T operator=(T desired) const</pre>	Equivalent to <code>store(desired)</code> . Returns <code>desired</code> .
<pre>T load(memory_order order = default_read_order, memory_scope scope = default_scope) const</pre>	Atomically loads the value of the object referenced by this <code>atomic_ref</code> . The memory order of this atomic operation must be <code>memory_order::relaxed</code> , <code>memory_order::acquire</code> , or <code>memory_order::seq_cst</code> . This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code> .
<pre>operator T() const</pre>	Equivalent to <code>load()</code> .
<pre>T exchange(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const</pre>	Atomically replaces the value of the object referenced by this <code>atomic_ref</code> with value <code>operand</code> and returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code> .

Member function	Description
<pre> bool compare_exchange_weak(T&amp; expected, T desired, memory_order success, memory_order failure, memory_scope scope = default_scope) const </pre>	<p>Atomically compares the value of the object referenced by this <code>atomic_ref</code> against the value of <code>expected</code>. If the values are equal, attempts to replace the value of the referenced object with the value of <code>desired</code>; otherwise assigns the original value of the referenced object to <code>expected</code>.</p> <p>Returns <code>true</code> if the comparison operation and replacement operation were successful. The <code>failure</code> memory order of this atomic operation must be <code>memory_order::relaxed</code>, <code>memory_order::acquire</code> or <code>memory_order::seq_cst</code>.</p> <p>This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</p>
<pre> bool compare_exchange_weak(T&amp; expected, T desired, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const </pre>	<p>Equivalent to <code>compare_exchange_weak(expected, desired, order, order, scope)</code>.</p>
<pre> bool compare_exchange_strong(T&amp; expected, T desired, memory_order success, memory_order failure, memory_scope scope = default_scope) const </pre>	<p>Atomically compares the value of the object referenced by this <code>atomic_ref</code> against the value of <code>expected</code>. If the values are equal, replaces the value of the referenced object with the value of <code>desired</code>; otherwise assigns the original value of the referenced object to <code>expected</code>.</p> <p>Returns <code>true</code> if the comparison operation was successful. The <code>failure</code> memory order of this atomic operation must be <code>memory_order::relaxed</code>, <code>memory_order::acquire</code> or <code>memory_order::seq_cst</code>.</p> <p>This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</p>

Member function	Description
<pre>bool compare_exchange_strong(     T&amp; expected, T desired,     memory_order order = default_read_modify_write_order) const</pre>	Equivalent to <code>compare_exchange_strong(expected, desired, order, order, scope)</code> .

Table 133. Additional member functions available on an object of type `atomic_ref<T>` for integral `T`

Member function	Description
<pre>T fetch_add(T operand, memory_order order =     default_read_modify_write_order,     memory_scope scope = default_scope) const</pre>	Atomically adds <code>operand</code> to the value of the object referenced by this <code>atomic_ref</code> and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code> .
<pre>T operator+=(T operand) const</pre>	Equivalent to <code>fetch_add(operand) + operand</code> .
<pre>T operator++(int) const</pre>	Equivalent to <code>fetch_add(1)</code> .
<pre>T operator++() const</pre>	Equivalent to <code>fetch_add(1) + 1</code> .
<pre>T fetch_sub(T operand, memory_order order =     default_read_modify_write_order,     memory_scope scope = default_scope) const</pre>	Atomically subtracts <code>operand</code> from the value of the object referenced by this <code>atomic_ref</code> and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code> .
<pre>T operator--(T operand) const</pre>	Equivalent to <code>fetch_sub(operand) - operand</code> .
<pre>T operator--(int) const</pre>	Equivalent to <code>fetch_sub(1)</code> .
<pre>T operator--() const</pre>	Equivalent to <code>fetch_sub(1) - 1</code> .



Member function	Description
<pre>T fetch_and(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const</pre>	Atomically performs a bitwise AND between <code>operand</code> and the value of the object referenced by this <code>atomic_ref</code> , and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code> .
<pre>T operator&amp;=(T operand) const</pre>	Equivalent to <code>fetch_and(operand) &amp; operand</code> .
<pre>T fetch_or(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const</pre>	Atomically performs a bitwise OR between <code>operand</code> and the value of the object referenced by this <code>atomic_ref</code> , and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code> .
<pre>T operator =(T operand) const</pre>	Equivalent to <code>fetch_or(operand)   operand</code> .
<pre>T fetch_xor(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const</pre>	Atomically performs a bitwise XOR between <code>operand</code> and the value of the object referenced by this <code>atomic_ref</code> , and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code> .
<pre>T operator^=(T operand) const</pre>	Equivalent to <code>fetch_xor(operand) ^ operand</code> .
<pre>T fetch_min(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const</pre>	Atomically computes the minimum of <code>operand</code> and the value of the object referenced by this <code>atomic_ref</code> , and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code> .

Member function	Description
<pre>T fetch_max(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const</pre>	Atomically computes the maximum of <code>operand</code> and the value of the object referenced by this <code>atomic_ref</code> , and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code> .

Table 134. Additional member functions available on an object of type `atomic_ref<T>` for floating-point T

Member function	Description
<pre>T fetch_add(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const</pre>	Atomically adds <code>operand</code> to the value of the object referenced by this <code>atomic_ref</code> and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code> .
<pre>T operator+=(T operand) const</pre>	Equivalent to <code>fetch_add(operand) + operand</code> .
<pre>T fetch_sub(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const</pre>	Atomically subtracts <code>operand</code> from the value of the object referenced by this <code>atomic_ref</code> and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code> .
<pre>T operator-=(T operand) const</pre>	Equivalent to <code>fetch_sub(operand) - operand</code> .
<pre>T fetch_min(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const</pre>	Atomically computes the minimum of <code>operand</code> and the value of the object referenced by this <code>atomic_ref</code> , and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code> .

Member function	Description
<pre>T fetch_max(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const</pre>	Atomically computes the maximum of <code>operand</code> and the value of the object referenced by this <code>atomic_ref</code> , and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code> .

Table 135. Additional member functions available on an object of type `atomic_ref<T*>`

Member function	Description
<pre>T* fetch_add(ptrdiff_t operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const</pre>	Atomically adds <code>operand</code> to the value of the object referenced by this <code>atomic_ref</code> and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit pointers on devices that have <code>aspect::atomic64</code> .
<pre>T* operator+=(ptrdiff_t operand) const</pre>	Equivalent to <code>fetch_add(operand) + operand</code> .
<pre>T* operator++(int) const</pre>	Equivalent to <code>fetch_add(1)</code> .
<pre>T* operator++() const</pre>	Equivalent to <code>fetch_add(1) + 1</code> .
<pre>T* fetch_sub(ptrdiff_t operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const</pre>	Atomically subtracts <code>operand</code> from the value of the object referenced by this <code>atomic_ref</code> and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit pointers on devices that have <code>aspect::atomic64</code> .
<pre>T* operator-=(ptrdiff_t operand) const</pre>	Equivalent to <code>fetch_sub(operand) - operand</code> .
<pre>T* operator--(int) const</pre>	Equivalent to <code>fetch_sub(1)</code> .
<pre>T* operator--() const</pre>	Equivalent to <code>fetch_sub(1) - 1</code> .

#### 4.15.4. Atomic types (deprecated)

The atomic types and operations on atomic types provided by SYCL 1.2.1 are deprecated in SYCL 2020,

and will be removed in a future version of SYCL.

The constructors and member functions for the `sycl::atomic` class are listed in [Table 136](#) and [Table 137](#) respectively.

```

1 namespace sycl {
2
3 /* Deprecated in SYCL 2020 */
4 template <typename T, access::address_space AddressSpace =
5             access::address_space::global_space>
6 class atomic {
7 public:
8     template <typename PointerT, access::decorated IsDecorated>
9     atomic(multi_ptr<PointerT, AddressSpace, IsDecorated> ptr);
10
11     void store(T operand, memory_order memoryOrder = memory_order::relaxed);
12
13     T load(memory_order memoryOrder = memory_order::relaxed) const;
14
15     T exchange(T operand, memory_order memoryOrder = memory_order::relaxed);
16
17     /* Available only when: T != float */
18     bool compare_exchange_strong(
19         T& expected, T desired,
20         memory_order successMemoryOrder = memory_order::relaxed,
21         memory_order failMemoryOrder = memory_order::relaxed);
22
23     /* Available only when: T != float */
24     T fetch_add(T operand, memory_order memoryOrder = memory_order::relaxed);
25
26     /* Available only when: T != float */
27     T fetch_sub(T operand, memory_order memoryOrder = memory_order::relaxed);
28
29     /* Available only when: T != float */
30     T fetch_and(T operand, memory_order memoryOrder = memory_order::relaxed);
31
32     /* Available only when: T != float */
33     T fetch_or(T operand, memory_order memoryOrder = memory_order::relaxed);
34
35     /* Available only when: T != float */
36     T fetch_xor(T operand, memory_order memoryOrder = memory_order::relaxed);
37
38     /* Available only when: T != float */
39     T fetch_min(T operand, memory_order memoryOrder = memory_order::relaxed);
40
41     /* Available only when: T != float */
42     T fetch_max(T operand, memory_order memoryOrder = memory_order::relaxed);
43 };
44
45 } // namespace sycl

```

The global functions are as follows and described in [Table 138](#).

```

1 namespace sycl {
2 /* Deprecated in SYCL 2020 */

```

```

3 template <typename T, access::address_space AddressSpace>
4 void atomic_store(atomic<T, AddressSpace> object, T operand,
5                  memory_order memoryOrder = memory_order::relaxed);
6
7 /* Deprecated in SYCL 2020 */
8 template <typename T, access::address_space AddressSpace>
9 T atomic_load(atomic<T, AddressSpace> object,
10              memory_order memoryOrder = memory_order::relaxed);
11
12 /* Deprecated in SYCL 2020 */
13 template <typename T, access::address_space AddressSpace>
14 T atomic_exchange(atomic<T, AddressSpace> object, T operand,
15                  memory_order memoryOrder = memory_order::relaxed);
16
17 /* Deprecated in SYCL 2020 */
18 template <typename T, access::address_space AddressSpace>
19 bool atomic_compare_exchange_strong(
20     atomic<T, AddressSpace> object, T& expected, T desired,
21     memory_order successMemoryOrder = memory_order::relaxed,
22     memory_order failMemoryOrder = memory_order::relaxed);
23
24 /* Deprecated in SYCL 2020 */
25 template <typename T, access::address_space AddressSpace>
26 T atomic_fetch_add(atomic<T, AddressSpace> object, T operand,
27                   memory_order memoryOrder = memory_order::relaxed);
28
29 /* Deprecated in SYCL 2020 */
30 template <typename T, access::address_space AddressSpace>
31 T atomic_fetch_sub(atomic<T, AddressSpace> object, T operand,
32                   memory_order memoryOrder = memory_order::relaxed);
33
34 /* Deprecated in SYCL 2020 */
35 template <typename T, access::address_space AddressSpace>
36 T atomic_fetch_and(atomic<T, AddressSpace> object, T operand,
37                   memory_order memoryOrder = memory_order::relaxed);
38
39 /* Deprecated in SYCL 2020 */
40 template <typename T, access::address_space AddressSpace>
41 T atomic_fetch_or(atomic<T, AddressSpace> object, T operand,
42                  memory_order memoryOrder = memory_order::relaxed);
43
44 /* Deprecated in SYCL 2020 */
45 template <typename T, access::address_space AddressSpace>
46 T atomic_fetch_xor(atomic<T, AddressSpace> object, T operand,
47                   memory_order memoryOrder = memory_order::relaxed);
48
49 /* Deprecated in SYCL 2020 */
50 template <typename T, access::address_space AddressSpace>
51 T atomic_fetch_min(atomic<T, AddressSpace> object, T operand,
52                   memory_order memoryOrder = memory_order::relaxed);
53
54 /* Deprecated in SYCL 2020 */
55 template <typename T, access::address_space AddressSpace>
56 T atomic_fetch_max(atomic<T, AddressSpace> object, T operand,
57                   memory_order memoryOrder = memory_order::relaxed);

```

```
58 } // namespace sycl
```

Table 136. Constructors of the `sycl::atomic` class template

Constructor	Description
<pre>template &lt;typename pointerT&gt; atomic(multi_ptr&lt;pointerT, AddressSpace&gt; ptr)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Permitted data types for <code>pointerT</code> are any valid scalar data type which is the same size in bytes as <code>T</code>. Constructs an instance of SYCL <code>atomic</code> which is associated with the pointer <code>ptr</code>, converted to a pointer of data type <code>T</code>.</p>

Table 137. Member functions available on an object of type `sycl::atomic<T>`

Member function	Description
<pre>void store(T operand, memory_order memoryOrder = memory_order::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Atomically stores the value <code>operand</code> at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code>. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</p>
<pre>T load(memory_order memoryOrder = memory_order::relaxed) const</pre>	<p>Deprecated in SYCL 2020.</p> <p>Atomically loads the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code>. Returns the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> before the call. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</p>

Member function	Description
<pre>T exchange(T operand, memory_order memoryOrder = memory_order::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Atomically replaces the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> with value <code>operand</code> and returns the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> before the call. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</p>
<pre>bool compare_exchange_strong(     T&amp; expected, T desired,     memory_order successMemoryOrder = memory_order:: relaxed,     memory_order failMemoryOrder = memory_order::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Available only when: <code>T != float</code>.</p> <p>Atomically compares the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> against the value of <code>expected</code>. If the values are equal, replaces value at address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> with the value of <code>desired</code>; otherwise assigns the original value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> to <code>expected</code>. Returns <code>true</code> if the comparison operation was successful. The memory order of this atomic operation must be <code>memory_order::relaxed</code> for both success and fail. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</p>

Member function	Description
<pre>T fetch_add(T operand, memory_order memoryOrder = memory_order::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Available only when: <code>T != float</code>.</p> <p>Atomically adds the value <code>operand</code> to the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> and assigns the result to the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code>. Returns the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> before the call. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</p>
<pre>T fetch_sub(T operand, memory_order memoryOrder = memory_order::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Available only when: <code>T != float</code>.</p> <p>Atomically subtracts the value <code>operand</code> to the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> and assigns the result to the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code>. Returns the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> before the call. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</p>



Member function	Description
<pre>T fetch_and(T operand, memory_order memoryOrder = memory_order::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Available only when: <code>T != float</code>.</p> <p>Atomically performs a bitwise AND between the value <code>operand</code> and the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> and assigns the result to the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code>. Returns the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> before the call. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</p>
<pre>T fetch_or(T operand, memory_order memoryOrder = memory_order::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Available only when: <code>T != float</code>.</p> <p>Atomically performs a bitwise OR between the value <code>operand</code> and the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> and assigns the result to the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code>. Returns the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> before the call. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</p>

Member function	Description
<pre>T fetch_xor(T operand, memory_order memoryOrder = memory_order::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Available only when: <code>T != float</code>.</p> <p>Atomically performs a bitwise XOR between the value <code>operand</code> and the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> and assigns the result to the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code>. Returns the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> before the call. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</p>
<pre>T fetch_min(T operand, memory_order memoryOrder = memory_order::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Available only when: <code>T != float</code>.</p> <p>Atomically computes the minimum of the value <code>operand</code> and the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> and assigns the result to the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code>. Returns the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> before the call. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</p>

Member function	Description
<pre>T fetch_max(T operand, memory_order memoryOrder = memory_order::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Available only when: <code>T != float</code>.</p> <p>Atomically computes the maximum of the value <code>operand</code> and the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> and assigns the result to the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code>. Returns the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> before the call. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</p>

Table 138. Global functions available on atomic types

Functions	Description
<pre>template &lt;typename T, access::address_space AddressSpace&gt; T atomic_load(atomic&lt;T, AddressSpace&gt; object, memory_order memoryOrder = memory_order ::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Equivalent to calling <code>object.load(memoryOrder)</code>.</p>
<pre>template &lt;typename T, access::address_space AddressSpace&gt; void atomic_store(atomic&lt;T, AddressSpace&gt; object, T operand, memory_order memoryOrder = memory_order ::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Equivalent to calling <code>object.store(operand, memoryOrder)</code>.</p>
<pre>template &lt;typename T, access::address_space AddressSpace&gt; T atomic_exchange(atomic&lt;T, AddressSpace&gt; object, T operand, memory_order memoryOrder = memory_order ::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Equivalent to calling <code>object.exchange(operand, memoryOrder)</code>.</p>
<pre>template &lt;typename T, access::address_space AddressSpace&gt; bool atomic_compare_exchange_strong( atomic&lt;T, AddressSpace&gt; object, T&amp; expected, T desired, memory_order successMemoryOrder = memory_order::relaxed memory_order failMemoryOrder = memory_order::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Equivalent to calling <code>object.compare_exchange_strong(expected, desired, successMemoryOrder, failMemoryOrders)</code>.</p>

Functions	Description
<pre>template &lt;typename T, access::address_space AddressSpace&gt; T atomic_fetch_add(atomic&lt;T, AddressSpace&gt; object, T operand, memory_order memoryOrder = memory_order::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Equivalent to calling <code>object.fetch_add(operand, memory_order)</code>.</p>
<pre>template &lt;typename T, access::address_space AddressSpace&gt; T atomic_fetch_sub(atomic&lt;T, AddressSpace&gt; object, T operand, memory_order memoryOrder = memory_order::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Equivalent to calling <code>object.fetch_sub(operand, memory_order)</code>.</p>
<pre>template &lt;typename T, access::address_space AddressSpace&gt; T atomic_fetch_and(atomic&lt;T&gt; operand, T object, memory_order memoryOrder = memory_order::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Equivalent to calling <code>object.fetch_and(operand, memory_order)</code>.</p>
<pre>template &lt;typename T, access::address_space AddressSpace&gt; T atomic_fetch_or(atomic&lt;T, AddressSpace&gt; object, T operand, memory_order memoryOrder = memory_order ::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Equivalent to calling <code>object.fetch_or(operand, memory_order)</code>.</p>
<pre>template &lt;typename T, access::address_space AddressSpace&gt; T atomic_fetch_xor(atomic&lt;T, AddressSpace&gt; object, T operand, memory_order memoryOrder = memory_order::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Equivalent to calling <code>object.fetch_xor(operand, memory_order)</code>.</p>
<pre>template &lt;typename T, access::address_space AddressSpace&gt; T atomic_fetch_min(atomic&lt;T, AddressSpace&gt; object, T operand, memory_order memoryOrder = memory_order::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Equivalent to calling <code>object.fetch_min(operand, memory_order)</code>.</p>
<pre>template &lt;typename T, access::address_space AddressSpace&gt; T atomic_fetch_max(atomic&lt;T, AddressSpace&gt; object, T operand, memory_order memoryOrder = memory_order::relaxed)</pre>	<p>Deprecated in SYCL 2020.</p> <p>Equivalent to calling <code>object.fetch_max(operand, memory_order)</code>.</p>

## 4.15.5. Interaction with host code

When a kernel runs on a device that has either `aspect::usm_atomic_host_allocations` or `aspect::usm_atomic_shared_allocations`, the device code and the host code can concurrently access the same memory. This has a ramification on the atomic operations because it is possible for device code and host code to perform atomic operations on the same object *M* in this shared memory. It also has a

ramification on the fence operations because the C++ core language defines the semantics of these fence operations in relation to atomic operations on some shared object  $M$ . The following paragraphs specify the guarantees that the SYCL implementation provides when the application performs atomic or fence operations in device code using the memory scope `memory_scope::system`.

Atomic operations in device code using `sycl::atomic_ref` on an object  $M$  are guaranteed to be atomic with respect to atomic operations in host code using `std::atomic_ref` on that same object  $M$ .

Fence operations in device code using `sycl::atomic_fence` synchronize with fence operations in host code using `std::atomic_thread_fence` if the fence operations shared the same atomic object  $M$  and follow the rules for fence synchronization defined in the C++ core language.

Fence operations in device code using `sycl::atomic_fence` synchronize with atomic operations in host code using `std::atomic_ref` if the operations share the same atomic object  $M$  and follow the rules for fence synchronization defined in the C++ core language.

Atomic operations in device code using `sycl::atomic_ref` synchronize with fence operations in host code using `std::atomic_thread_fence` if the operations share the same atomic object  $M$  and follow the rules for fence synchronization defined in the C++ core language.

## 4.16. Stream class

The SYCL `stream` class is a buffered output stream that allows outputting the values of built-in, vector and SYCL types to the console. The implementation of how values are streamed to the console is left as an implementation detail.

The way in which values are output by an instance of the SYCL `stream` class can also be altered using a range of manipulators.

There are two limits that are relevant for the `stream` class. The `totalBufferSize` limit specifies the maximum size of the overall character stream that can be output during a kernel invocation, and the `workItemBufferSize` limit specifies the maximum size of the character stream that can be output within a work-item before a flush must be performed. Both of these limits are specified in bytes. The `totalBufferSize` limit must be sufficient to contain the characters output by all stream statements during execution of a kernel invocation (the aggregate of outputs from all work-items), and the `workItemBufferSize` limit must be sufficient to contain the characters output within a work-item between stream flush operations.

If the `totalBufferSize` or `workItemBufferSize` limits are exceeded, it is implementation-defined whether the streamed characters exceeding the limit are output, or silently ignored/discarded, and if output it is implementation-defined whether those extra characters exceeding the `workItemBufferSize` limit count toward the `totalBufferSize` limit. Regardless of this implementation defined behavior of output exceeding the limits, no undefined or erroneous behavior is permitted of an implementation when the limits are exceeded. Unused characters within `workItemBufferSize` (any portion of the `workItemBufferSize` capacity that has not been used at the time of a stream flush) do not count toward the `totalBufferSize` limit, in that only characters flushed count toward the `totalBufferSize` limit.

The SYCL `stream` class provides the common reference semantics (see [Section 4.5.2](#)).

### 4.16.1. Stream class interface

The constructors and member functions of the SYCL `stream` class are listed in [Table 141](#), [Table 142](#), and [Table 143](#) respectively. The additional common special member functions and common member functions are listed in [Table 7](#) and [Table 8](#), respectively.

The operand types that are supported by the SYCL `stream` class `operator<<()` operator are listed in [Table 139](#).

The manipulators that are supported by the SYCL `stream` class `operator<<()` operator are listed in [Table 140](#).

```

1 namespace sycl {
2
3 enum class stream_manipulator : /* unspecified */ {
4     flush,
5     dec,
6     hex,
7     oct,
8     noshowbase,
9     showbase,
10    noshowpos,
11    showpos,
12    endl,
13    fixed,
14    scientific,
15    hexfloat,
16    defaultfloat
17 };
18
19 const stream_manipulator flush = stream_manipulator::flush;
20
21 const stream_manipulator dec = stream_manipulator::dec;
22
23 const stream_manipulator hex = stream_manipulator::hex;
24
25 const stream_manipulator oct = stream_manipulator::oct;
26
27 const stream_manipulator noshowbase = stream_manipulator::noshowbase;
28
29 const stream_manipulator showbase = stream_manipulator::showbase;
30
31 const stream_manipulator noshowpos = stream_manipulator::noshowpos;
32
33 const stream_manipulator showpos = stream_manipulator::showpos;
34
35 const stream_manipulator endl = stream_manipulator::endl;
36
37 const stream_manipulator fixed = stream_manipulator::fixed;
38
39 const stream_manipulator scientific = stream_manipulator::scientific;
40
41 const stream_manipulator hexfloat = stream_manipulator::hexfloat;
42
43 const stream_manipulator defaultfloat = stream_manipulator::defaultfloat;
44
45 __precision_manipulator__ setprecision(int precision);
46
47 __width_manipulator__ setw(int width);
48
49 class stream {
50 public:
51     stream(size_t totalBufferSize, size_t workItemBufferSize,
52           handler& cgh, const property_list& proplist = {});
53

```

```

54  /* -- common interface members -- */
55
56  /* -- property interface members -- */
57
58  size_t size() const noexcept;
59
60  // Deprecated
61  size_t get_size() const;
62
63  size_t get_work_item_buffer_size() const;
64
65  /* get_max_statement_size() has the same functionality as
66     get_work_item_buffer_size(), and is provided for backward compatibility.
67     get_max_statement_size() is a deprecated query. */
68  size_t get_max_statement_size() const;
69 };
70
71 template <typename T> const stream& operator<<(const stream& os, const T& rhs);
72
73 } // namespace sycl

```

Table 139. Operand types supported by the `stream` class

Stream operand type	Description
<code>bool</code> , <code>char</code> , <code>signed char</code> , <code>unsigned char</code> , <code>int</code> , <code>unsigned int</code> , <code>short</code> , <code>unsigned short</code> , <code>long int</code> , <code>unsigned long int</code> , <code>long long int</code> , <code>unsigned long long int</code>	Outputs the value as a stream of characters.
<code>float</code> , <code>double</code> , <code>half</code>	Outputs the value according to the precision of the current statement as a stream of characters.
<code>char*</code> , <code>const char*</code>	Outputs the string.
<code>T*</code> , <code>const T*</code> , <code>multi_ptr</code>	Outputs the address of the pointer as a stream of characters.
<code>vec</code>	Outputs the value of each component of the vector as a stream of characters.
<code>id</code> , <code>range</code> , <code>item</code> , <code>nd_item</code> , <code>group</code> , <code>nd_range</code> , <code>h_item</code>	Outputs the value of each component of each id or range as a stream of characters.

Table 140. Manipulators supported by the `stream` class

Stream manipulator	Description
<code>flush</code>	Triggers a flush operation, which copies the contents of the work-item stream buffer to the global stream buffer, and then empties the work-item stream buffer. After a flush, the full <code>workItemBufferSize</code> is available again for subsequent streaming within the work-item.
<code>endl</code>	Outputs a new-line character and then triggers a flush operation.
<code>dec</code>	Outputs any subsequent values in the current statement in decimal base.
<code>hex</code>	Outputs any subsequent values in the current statement in hexadecimal base.
<code>oct</code>	Outputs any subsequent values in the current statement in octal base.
<code>noshowbase</code>	Outputs any subsequent values without the base prefix.
<code>showbase</code>	Outputs any subsequent values with the base prefix.
<code>noshwpos</code>	Outputs any subsequent values without a plus sign if the value is positive.
<code>showpos</code>	Outputs any subsequent values with a plus sign if the value is positive.
<code>setw(int)</code>	Sets the field width of any subsequent values in the current statement.
<code>setprecision(int)</code>	Sets the precision of any subsequent values in the current statement.
<code>fixed</code>	Outputs any subsequent floating-point values in the current statement in fixed notation.
<code>scientific</code>	Outputs any subsequent floating-point values in the current statement in scientific notation.
<code>hexfloat</code>	Outputs any subsequent floating-point values in the current statement in hexadecimal notation.



Stream manipulator	Description
<code>defaultfloat</code>	Outputs any subsequent floating-point values in the current statement in the default notation.

Table 141. Constructors of the `stream` class

Constructor	Description
<pre>stream(size_t totalBufferSize, size_t workItemBufferSize,         handler&amp; cgh,         const property_list&amp; propList = {})</pre>	Constructs a SYCL <code>stream</code> instance associated with the command group specified by <code>cgh</code> , with a maximum buffer size in bytes per kernel invocation specified by the parameter <code>totalBufferSize</code> , and a maximum stream size that can be buffered by a work-item between stream flushes specified by the parameter <code>workItemBufferSize</code> . Zero or more properties can be provided to the constructed SYCL <code>stream</code> via an instance of <code>property_list</code> .

Table 142. Member functions of the `stream` class

Member function	Description
<code>size_t size() const noexcept</code>	Returns the total buffer size, in bytes.
<code>size_t get_size() const</code>	Returns the same value as <code>size()</code> . Deprecated.
<code>size_t get_work_item_buffer_size() const</code>	Returns the buffer size per work-item, in bytes.
<code>size_t get_max_statement_size() const</code>	Deprecated query with same functionality as <code>get_work_item_buffer_size()</code> .

Table 143. Global functions of the `stream` class

Global function	Description
<pre>template &lt;typename T&gt; const stream&amp; operator&lt;&lt;(const stream&amp; os, const T&amp; rhs)</pre>	Outputs any valid values (see <a href="#">Table 139</a> ) as a stream of characters and applies any valid manipulator (see <a href="#">Table 140</a> ) to the current stream.

## 4.16.2. Output

An instance of the SYCL `stream` class is required to output everything that is streamed to it via the `operator<<()` operator before a flush operation (that doesn't exceed the `workItemBufferSize` or `totalBufferSize` limits) within a SYCL kernel function by the time that the event associated with a command group submission enters the completed state. The point at which the flush operation is performed is implementation-defined.

The SYCL `stream` class is required to output the content of each stream, between flushes (up to `workItem-BufferSize`), without mixing with content from the same stream in other work-items. There are no other output order guarantees between work-items or between streams. The stream flush operation therefore delimits the unit of output that is guaranteed to be displayed without mixing with other work-items, with respect to a single stream.

### 4.16.3. Implicit flush

There is guaranteed to be an implicit flush of each stream used by a kernel, at the end of kernel execution, from the perspective of each work-item. There is also an implicit flush when the `endl` stream manipulator is executed. No other implicit flushes are permitted in an implementation.

### 4.16.4. Performance note

The usage of the `stream` class is designed for debugging purposes and is therefore not recommended for performance critical applications.

## 4.17. SYCL built-in functions for SYCL host and device

SYCL kernels may execute on any SYCL device, which requires the functions used in the kernels to be compiled and linked for both device and host. In the SYCL programming model, the built-ins are available for the entire SYCL application within the `sycl` namespace, although their semantics may be different. This section follows the OpenCL 1.2 specification document [ch. 6.12](#) - except that for SYCL, all functions are located within the `sycl` namespace - and describes the behavior of these functions for SYCL host and device. The expected precision and any other semantic requirements are defined in the back-end specification.

The SYCL built-in functions are available throughout the SYCL application, and depending on where they execute, they are either implemented using their host implementation or the device implementation. The SYCL system guarantees that all of the built-in functions fulfill the same requirements for both host and device.

### 4.17.1. Function objects

SYCL provides a number of function objects in the `sycl` namespace on host and device. All function objects obey C++ conversion and promotion rules. Each function object is additionally specialized for `void` as a *transparent* function object that deduces its parameter types and return type.

```

1 namespace sycl {
2
3 template <typename T = void> struct plus {
4     T operator()(const T& x, const T& y) const;
5 };
6
7 template <typename T = void> struct multiplies {
8     T operator()(const T& x, const T& y) const;
9 };
10
11 template <typename T = void> struct bit_and {
12     T operator()(const T& x, const T& y) const;
13 };
14
15 template <typename T = void> struct bit_or {
16     T operator()(const T& x, const T& y) const;
17 };

```

```

18
19 template <typename T = void> struct bit_xor {
20     T operator()(const T& x, const T& y) const;
21 };
22
23 template <typename T = void> struct logical_and {
24     T operator()(const T& x, const T& y) const;
25 };
26
27 template <typename T = void> struct logical_or {
28     T operator()(const T& x, const T& y) const;
29 };
30
31 template <typename T = void> struct minimum {
32     T operator()(const T& x, const T& y) const;
33 };
34
35 template <typename T = void> struct maximum {
36     T operator()(const T& x, const T& y) const;
37 };
38
39 } // namespace sycl

```

Table 144. Member functions for the **plus** function object

Member function	Description
<code>T operator()(const T&amp; x, const T&amp; y) const</code>	Returns the sum of its arguments, equivalent to <code>x + y</code> .

Table 145. Member functions for the **multiplies** function object

Member function	Description
<code>T operator()(const T&amp; x, const T&amp; y) const</code>	Returns the product of its arguments, equivalent to <code>x * y</code> .

Table 146. Member functions for the **bit\_and** function object

Member function	Description
<code>T operator()(const T&amp; x, const T&amp; y) const</code>	Returns the bitwise AND of its arguments, equivalent to <code>x &amp; y</code> .

Table 147. Member functions for the **bit\_or** function object

Member function	Description
<code>T operator()(const T&amp; x, const T&amp; y) const</code>	Returns the bitwise OR of its arguments, equivalent to <code>x   y</code> .

Table 148. Member functions for the **bit\_xor** function object

Member function	Description
<code>T operator()(const T&amp; x, const T&amp; y) const</code>	Returns the bitwise XOR of its arguments, equivalent to <code>x ^ y</code> .

Table 149. Member functions for the **logical\_and** function object

Member function	Description
<code>T operator()(const T&amp; x, const T&amp; y) const</code>	Returns the logical AND of its arguments, equivalent to <code>x &amp;&amp; y</code> .

Table 150. Member functions for the `logical_or` function object

Member function	Description
<code>T operator()(const T&amp; x, const T&amp; y) const</code>	Returns the logical OR of its arguments, equivalent to <code>x    y</code> .

Table 151. Member functions for the `minimum` function object

Member function	Description
<code>T operator()(const T&amp; x, const T&amp; y) const</code>	Returns the smaller value. Returns the first argument when the arguments are equivalent.

Table 152. Member functions for the `maximum` function object

Member function	Description
<code>T operator()(const T&amp; x, const T&amp; y) const</code>	Returns the larger value. Returns the first argument when the arguments are equivalent.

## 4.17.2. Group functions

SYCL provides a number of functions that expose functionality tied to groups of work-items (such as [group barriers](#) and collective operations). These group functions act as *synchronization points* and must be encountered in converged [control flow](#) by all work-items in the group.

The behavior of every group function is as follows:

- Each work-item in the group arrives at the synchronization point associated with the group function, then blocks until any operation(s) specified by the group function have completed.
- Once all work-items in the group have arrived, an unspecified subset of those work-items cooperate to execute any operation(s) specified by the group function.
- When the set of cooperating work-items have completed execution of all operation(s) specified by the group function, all work-items blocked on the synchronization point associated with the group function are unblocked.

The completion of the operation(s) specified by the group function happens before the returns from all calls that were unblocked.



The behavior of group functions is analogous to the behavior of the C++20 `std::barrier::arrive_and_wait` function, for an implementation-defined barrier object with an expected count equal to the number of work-items in the group. Any operation(s) performed by the group function behave as if they were defined in the barrier's completion function and were invoked as part of the barrier's phase completion step.

If one work-item in a group calls a group function, then all work-items in that group must call exactly the same function under the same set of conditions --- calling the same function under different conditions (e.g. in different iterations of a loop, or different branches of a conditional statement) results in undefined behavior. Additionally, restrictions may be placed on the arguments passed to each function in order to ensure that all work-items in the group agree on the operation that is being performed. Any

such restrictions on the arguments passed to a function are defined within the descriptions of those functions. Violating these restrictions results in undefined behavior.

All group functions are supported for the fundamental scalar types supported by SYCL (see [Table 153](#)) and instances of the SYCL `vec` and `marray` classes.

Using a group function inside of a kernel may introduce additional limits on the resources available to user code inside the same kernel. The behavior of these limits is implementation-defined, but must be reflected by calls to kernel querying functions (such as `kernel::get_info`) as described in [Section 4.11.13.1](#).

It is undefined behavior for any group function to be invoked within a `parallel_for_work_group` or `parallel_for_work_item` context.

#### 4.17.2.1. Group type trait

```
1 namespace sycl {
2   template <class T> struct is_group;
3
4   template <class T> inline constexpr bool is_group_v = is_group<T>::value;
5 } // namespace sycl
```

The `is_group` type trait is used to determine which types of groups are supported by group functions, and to control when group functions participate in overload resolution.

`is_group<T>` inherits from `std::true_type` if `T` is the type of a standard SYCL group (`group` or `sub_group`) and it inherits from `std::false_type` otherwise. A SYCL implementation may introduce additional specializations of `is_group<T>` for implementation-defined group types, if the interface of those types supports all member functions and static members common to the `group` and `sub_group` classes.

#### 4.17.2.2. `group_broadcast`

The `group_broadcast` function communicates a value held by one work-item to all other work-items in the group.

```
1 template <typename Group, typename T> T group_broadcast(Group g, T x); // (1)
2
3 template <typename Group, typename T>
4 T group_broadcast(Group g, T x, Group::linear_id_type local_linear_id); // (2)
5
6 template <typename Group, typename T>
7 T group_broadcast(Group g, T x, Group::id_type local_id); // (3)
```

1. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true and `T` is a trivially copyable type.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the broadcast operation.

*Synchronization:* The call to this function in each work-item happens before the broadcast operation begins execution. The completion of the broadcast operation happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* The value of `x` from the work-item with the smallest linear id within group `g`.

2. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true and `T` is a trivially copy-

able type.

*Preconditions:* `local_linear_id` must be the same for all work-items in the group and must be in the range `[0, get_local_linear_range())`.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the broadcast operation.

*Synchronization:* The call to this function in each work-item happens before the broadcast operation begins execution. The completion of the broadcast operation happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* The value of `x` from the work-item with the specified linear id within group `g`.

3. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true and `T` is a trivially copyable type.

*Preconditions:* `local_id` must be the same for all work-items in the group, and its dimensionality must match the dimensionality of the group. The value of `local_id` in each dimension must be greater than or equal to 0 and less than the value of `get_local_range()` in the same dimension.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the broadcast operation.

*Synchronization:* The call to this function in each work-item happens before the broadcast operation begins execution. The completion of the broadcast operation happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* The value of `x` from the work-item with the specified id within group `g`.

#### 4.17.2.3. `group_barrier`

The `group_barrier` function is a coordination mechanism for all work-items in a group.

```
1 template <typename Group>
2 void group_barrier(Group g,
3                   memory_scope scope = Group::fence_scope); // (1)
```

1. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point.

*Synchronization:* The call to `group_barrier` in each work-item happens before any work-item blocking on the same synchronization point is unblocked. Synchronization operations used in an implementation of `group_barrier` must respect the memory scope specified by the `scope` parameter, which defaults to the narrowest scope including all work-items in group `g` (as reported by `Group::fence_scope`).

### 4.17.3. Group algorithms library

SYCL provides an algorithms library based on the functions described in Section 28 of the C++17 specification. The first argument to each function is a `group`, and data ranges can be described using pointers or instances of the `multi_ptr` class. The functions defined in this section are free functions available in the `sycl` namespace.

Any restrictions from the standard algorithms library apply. Some of the functions in the SYCL algorithms library introduce additional restrictions in order to maximize portability across different devices and to minimize the chances of encountering unexpected behavior.

All algorithms are supported for the fundamental scalar types supported by SYCL (see [Table 153](#)) and instances of the SYCL `vec` and `marray` classes.

The `group` argument to a SYCL algorithm denotes that it should be performed collaboratively by the work-items in the specified group. All algorithms act as group functions (as defined in [Section 4.17.2](#)), inheriting all restrictions of group functions. Unless the description of a function says otherwise, how the elements of a range are processed by the work-items in a group is undefined.

SYCL provides separate functions for algorithms which use the work-items in a group to execute an operation over a range (specified by a start pointer and an end pointer) and algorithms which are applied to data held directly by the work-items in a group. An example of the usage of these functions is given below:

```

1  buffer<int> inputBuf{1024};
2  buffer<int> outputBuf{2};
3  {
4    // Initialize buffer on the host with 0, 1, 2, 3, ..., 1023
5    host_accessor a{inputBuf};
6    std::iota(a.begin(), a.end(), 0);
7  }
8
9  myQueue.submit([&](handler& cgh) {
10   accessor inputValues{inputBuf, cgh, read_only};
11   accessor outputValues{outputBuf, cgh, write_only, no_init};
12
13   cgh.parallel_for(nd_range<1>(range<1>(16)), range<1>(16)), [=](nd_item<1> it) {
14     // Apply a group algorithm to any number of values, described by an iterator
15     // range. The work-group reduces all inputValues and each work-item works on
16     // part of the range.
17     int* first = inputValues.get_pointer();
18     int* last = first + 1024;
19     int sum = joint_reduce(it.get_group(), first, last, plus<>());
20     outputValues[0] = sum;
21
22     // Apply a group algorithm to a set of values held directly by work-items.
23     // The work-group reduces a number of values equal to the size of the group
24     // and each work-item provides one value.
25     int partial_sum = reduce_over_group(
26       it.get_group(), inputValues[it.get_global_linear_id()], plus<>());
27     outputValues[1] = partial_sum;
28   });
29 });
30
31 host_accessor a{outputBuf};
32 assert(a[0] == 523776 && a[1] == 120);

```

#### 4.17.3.1. `any_of`, `all_of` and `none_of`

The `any_of`, `all_of` and `none_of` functions from standard C++ test whether Boolean conditions hold for any of, all of or none of the values in a range, respectively.

SYCL provides two sets of similar algorithms:

1. `joint_any_of`, `joint_all_of` and `joint_none_of` use the work-items in a group to execute the corresponding algorithm in parallel.



2. `any_of_group`, `all_of_group` and `none_of_group` test Boolean conditions applied to data held directly by the work-items in a group.

```

1 template <typename Group, typename Ptr, typename Predicate>
2 bool joint_any_of(Group g, Ptr first, Ptr last, Predicate pred); // (1)
3
4 template <typename Group, typename T, typename Predicate>
5 bool any_of_group(Group g, T x, Predicate pred); // (2)
6
7 template <typename Group> bool any_of_group(Group g, bool pred); // (3)

```

1. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true and `Ptr` is a pointer.

*Preconditions:* `first` and `last` must be the same for all work-items in group `g`, and `pred` must be an immutable callable with the same type and state for all work-items in group `g`.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* true if `pred` returns true when applied to the result of dereferencing any pointer in the range `[first, last)`.

2. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true.

*Preconditions:* `pred` must be an immutable callable with the same type and state for all work-items in group `g`.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* true if `pred(x)` returns true for any work-item in group `g`.

3. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* true if `pred` is true for any work-item in group `g`.

```

1 template <typename Group, typename Ptr, typename Predicate>
2 bool joint_all_of(Group g, Ptr first, Ptr last, Predicate pred); // (1)
3
4 template <typename Group, typename T, typename Predicate>
5 bool all_of_group(Group g, T x, Predicate pred); // (2)
6

```



```
7 template <typename Group> bool all_of_group(Group g, bool pred); // (3)
```

1. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true and `Ptr` is a pointer.

*Preconditions:* `first` and `last` must be the same for all work-items in group `g`, and `pred` must be an immutable callable with the same type and state for all work-items in group `g`.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* true if `pred` returns true when applied to the result of dereferencing all pointers in the range `[first, last)`.

2. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true.

*Preconditions:* `pred` must be an immutable callable with the same type and state for all work-items in group `g`.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* true if `pred(x)` returns true for all work-items in group `g`.

3. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* true if `pred` is true for all work-items in group `g`.

```
1 template <typename Group, typename Ptr, typename Predicate>
2 bool joint_none_of(Group g, Ptr first, Ptr last, Predicate pred); // (1)
3
4 template <typename Group, typename T, typename Predicate>
5 bool none_of_group(Group g, T x, Predicate pred); // (2)
6
7 template <typename Group> bool none_of_group(Group g, bool pred); // (3)
```

1. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true and `Ptr` is a pointer.

*Preconditions:* `first` and `last` must be the same for all work-items in group `g`, and `pred` must be an immutable callable with the same type and state for all work-items in group `g`.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* true if `pred` returns false when applied to the result of dereferencing all pointers in the range `[first, last)`.

2. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true.

*Preconditions:* `pred` must be an immutable callable with the same type and state for all work-items in group `g`.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* true if `pred(x)` returns false for all work-items in group `g`.

3. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* true if `pred` is false for all work-items in group `g`.

#### 4.17.3.2. `shift_left` and `shift_right`

The `shift_left` and `shift_right` functions from standard C++ move values in a range down (to the left) or up (to the right) respectively.

SYCL provides similar algorithms compatible with the `sub_group` class:

1. `shift_group_left` and `shift_group_right` move values held by the work-items in a group directly to another work-item in group `g`, by shifting values a fixed number of work-items to the left or right.

```
1 template <typename Group, typename T>
2 T shift_group_left(Group g, T x, Group::linear_id_type delta = 1); // (1)
3
4 template <typename Group, typename T>
5 T shift_group_right(Group g, T x, Group::linear_id_type delta = 1); // (2)
```

1. *Constraints:* Available only if `std::is_same_v<std::decay_t<Group>, sub_group>` is true and `T` is a trivially copyable type.

*Preconditions:* `delta` must be the same for all work-items in the group.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* the value of `x` from the work-item whose group local id (`id`) is `delta` larger than that of the calling work-item. `id + delta` may be greater than or equal to the group's linear size, but the value returned in this case is unspecified.

1. *Constraints:* Available only if `std::is_same_v<std::decay_t<Group>, sub_group>` is true and `T` is a trivially copyable type.

*Preconditions:* `delta` must be the same for all work-items in the group.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* the value of `x` from the work-item whose group local id (`id`) is `delta` smaller than that of the calling work-item. `id - delta` may be less than 0, but the value returned in this case is unspecified.

#### 4.17.3.3. `permute`

SYCL provides an algorithm to permute the values held by work-items in a sub-group:

1. `permute_group_by_xor` permutes values by exchanging values held by pairs of work-items identified by computing the bitwise exclusive OR of the work-item id and some fixed mask.

```
1 template <typename Group, typename T>
2 T permute_group_by_xor(Group g, T x, Group::linear_id_type mask); // (1)
```

1. *Constraints:* Available only if `std::is_same_v<std::decay_t<Group>, sub_group>` is true and `T` is a trivially copyable type.

*Preconditions:* `mask` must be the same for all work-items in the group.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* the value of `x` from the work-item whose group local id is equal to the bitwise exclusive OR of the calling work-item's group local id and `mask`. The result of the exclusive OR may be greater than or equal to the group's linear size, but the value returned in this case is unspecified.

#### 4.17.3.4. `select`

SYCL provides an algorithm to directly exchange the values held by work-items in a sub-group:

1. `select_from_group` allows work-items to obtain a copy of a value held by any other work-item in group `g`.

```
1 template <typename Group, typename T>
2 T select_from_group(Group g, T x, Group::id_type remote_local_id); // (1)
```

1. *Constraints:* Available only if `std::is_same_v<std::decay_t<Group>, sub_group>` is true and `T` is a trivially copyable type.

*Effects:* Blocks until all work-items in group **g** have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* the value of **x** from the work-item with the group local id specified by **remote\_local\_id**. The value of **remote\_local\_id** may be outside of the group, but the value returned in this case is unspecified.

#### 4.17.3.5. **reduce**

The **reduce** function from standard C++ combines the values in a range in an unspecified order using a binary operator.

SYCL provides two similar algorithms that compute the same generalized sum as defined by standard C++:

1. **joint\_reduce** uses the work-items in a group to execute a **reduce** operation in parallel.
2. **reduce\_over\_group** combines values held directly by the work-items in a group.

The result of a call to these functions is non-deterministic if the binary operator is not commutative and associative. Only the binary operators defined in [Section 4.17.1](#) are supported by the **reduce** functions in SYCL 2020, but the standard C++ syntax is used for forward compatibility with future SYCL versions.

```

1 template <typename Group, typename Ptr, typename BinaryOperation>
2 std::iterator_traits<Ptr>::value_type
3 joint_reduce(Group g, Ptr first, Ptr last, BinaryOperation binary_op); // (1)
4
5 template <typename Group, typename Ptr, typename T, typename BinaryOperation>
6 T joint_reduce(Group g, Ptr first, Ptr last, T init,
7               BinaryOperation binary_op); // (2)
8
9 template <typename Group, typename T, typename BinaryOperation>
10 T reduce_over_group(Group g, T x, BinaryOperation binary_op); // (3)
11
12 template <typename Group, typename V, typename T, typename BinaryOperation>
13 T reduce_over_group(Group g, V x, T init, BinaryOperation binary_op); // (4)

```

1. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true, **Ptr** is a pointer to a fundamental type, and **BinaryOperation** is a SYCL function object type.

*Mandates:* `binary_op(*first, *first)` must return a value of type `std::iterator_traits<Ptr>::value_type`.

*Preconditions:* **first**, **last** and the type of **binary\_op** must be the same for all work-items in group **g**. **binary\_op** must be an instance of a SYCL function object.

*Effects:* Blocks until all work-items in group **g** have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* The result of combining the values resulting from dereferencing all pointers in the range

`[first, last)` using the operator `binary_op`, where the values are combined according to the generalized sum defined in standard C++.

2. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true, `Ptr` is a pointer to a fundamental type, `T` is a fundamental type, and `BinaryOperation` is a SYCL function object type.

*Mandates:* `binary_op(init, *first)` must return a value of type `T`.

*Preconditions:* `first`, `last`, `init` and the type of `binary_op` must be the same for all work-items in group `g`. `binary_op` must be an instance of a SYCL function object.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* The result of combining the values resulting from dereferencing all pointers in the range `[first, last)` and the initial value `init` using the operator `binary_op`, where the values are combined according to the generalized sum defined in standard C++.

3. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true, `T` is a fundamental type and `BinaryOperation` is a SYCL function object type.

*Mandates:* `binary_op(x, x)` must return a value of type `T`.

*Preconditions:* `binary_op` must be an instance of a SYCL function object.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* The result of combining all the values of `x` specified by each work-item in group `g` using the operator `binary_op`, where the values are combined according to the generalized sum defined in standard C++.

4. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true, `V` and `T` are fundamental types, and `BinaryOperation` is a SYCL function object type.

*Mandates:* `binary_op(init, x)` must return a value of type `T`.

*Preconditions:* `binary_op` must be an instance of a SYCL function object.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* The result of combining all the values of `x` specified by each work-item in group `g` and the initial value `init` using the operator `binary_op`, where the values are combined according to the generalized sum defined in standard C++.

#### 4.17.3.6. `exclusive_scan` and `inclusive_scan`

The `exclusive_scan` and `inclusive_scan` functions in standard C++ compute a prefix sum using a binary operator. For a scan of elements  $[x_0, \dots, x_n]$ , the  $i$ th result in an exclusive scan is the generalized noncommutative sum of all elements preceding  $x_i$  (excluding  $x_i$  itself), whereas the  $i$ th result in an inclusive scan is the generalized noncommutative sum of all elements preceding  $x_i$  (including  $x_i$  itself).

SYCL provides two similar sets of algorithms that compute the same prefix sums using the generalized noncommutative sum as defined by standard C++:

1. `joint_exclusive_scan` and `joint_inclusive_scan` use the work-items in a group to execute the corresponding algorithm in parallel, and intermediate partial prefix sums are written to memory as in standard C++.
2. `exclusive_scan_over_group` and `inclusive_scan_over_group` perform a scan over values held directly by the work-items in a group, and the result returned to each work-item represents a partial prefix sum.

The result of a call to a scan is non-deterministic if the binary operator is not associative. Only the binary operators defined in [Section 4.17.1](#) are supported by the scan functions in SYCL 2020, but the standard C++ syntax is used for forward compatibility with future SYCL versions.

```

1 template <typename Group, typename InPtr, typename OutPtr,
2           typename BinaryOperation>
3 OutPtr joint_exclusive_scan(Group g, InPtr first, InPtr last, OutPtr result,
4                             BinaryOperation binary_op); // (1)
5
6 template <typename Group, typename InPtr, typename OutPtr, typename T,
7           typename BinaryOperation>
8 OutPtr joint_exclusive_scan(Group g, InPtr first, InPtr last, OutPtr result,
9                             T init, BinaryOperation binary_op); // (2)
10
11 template <typename Group, typename T, typename BinaryOperation>
12 T exclusive_scan_over_group(Group g, T x, BinaryOperation binary_op); // (3)
13
14 template <typename Group, typename V, typename T, typename BinaryOperation>
15 T exclusive_scan_over_group(Group g, V x, T init,
16                             BinaryOperation binary_op); // (4)

```

1. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true, `InPtr` and `OutPtr` are pointers to fundamental types, and `BinaryOperation` is a SYCL function object type.

*Mandates:* `binary_op(*first, *first)` must return a value of type `std::iterator_traits<OutPtr>::value_type`.

*Preconditions:* `first`, `last`, `result` and the type of `binary_op` must be the same for all work-items in group `g`. `binary_op` must be an instance of a SYCL function object.



Note that `first` may be equal to `result`.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

The value written to `result + i` is the exclusive scan of the values resulting from dereferencing the first  $i$  values in the range `[first, last)` and the identity value of `binary_op` (as identified by `sycl::known_identity`), using the operator `binary_op`. The scan is computed using a generalized non-commutative sum as defined in standard C++.



*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* A pointer to the end of the output range.

2. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true, `InPtr` and `OutPtr` are pointers to fundamental types, `T` is a fundamental type, and `BinaryOperation` is a SYCL function object type.

*Mandates:* `binary_op(init, *first)` must return a value of type `T`.

*Preconditions:* `first`, `last`, `result`, `init` and the type of `binary_op` must be the same for all work-items in group `g`. `binary_op` must be an instance of a SYCL function object.



Note that `first` may be equal to `result`.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

The value written to `result + i` is the exclusive scan of the values resulting from dereferencing the first `i` values in the range `[first, last)` and an initial value specified by `init`, using the operator `binary_op`. The scan is computed using a generalized noncommutative sum as defined in standard C++.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* A pointer to the end of the output range.

3. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true, `T` is a fundamental type, and `BinaryOperation` is a SYCL function object type.

*Mandates:* `binary_op(x, x)` must return a value of type `T`.

*Preconditions:* `binary_op` must be an instance of a SYCL function object.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* The value returned on work-item `i` is the exclusive scan of the first `i` values in group `g` and the identity value of `binary_op` (as identified by `sycl::known_identity`), using the operator `binary_op`. The scan is computed using a generalized noncommutative sum as defined in standard C++. For multi-dimensional groups, the order of work-items in group `g` is determined by their linear id.

4. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true, `V` and `T` are fundamental types, and `BinaryOperation` is a SYCL function object type.

*Mandates:* `binary_op(init, x)` must return a value of type `T`.

*Preconditions:* `binary_op` must be an instance of a SYCL function object.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins exe-

cution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* The value returned on work-item  $i$  is the exclusive scan of the first  $i$  values in group  $g$  and an initial value specified by `init`, using the operator `binary_op`. The scan is computed using a generalized noncommutative sum as defined in standard C++. For multi-dimensional groups, the order of work-items in group  $g$  is determined by their linear id.

```

1 template <typename Group, typename InPtr, typename OutPtr,
2           typename BinaryOperation>
3 OutPtr joint_inclusive_scan(Group g, InPtr first, InPtr last, OutPtr result,
4                             BinaryOperation binary_op); // (1)
5
6 template <typename Group, typename InPtr, typename OutPtr, typename T,
7           typename BinaryOperation>
8 OutPtr joint_inclusive_scan(Group g, InPtr first, InPtr last, OutPtr result,
9                             BinaryOperation binary_op, T init); // (2)
10
11 template <typename Group, typename T, typename BinaryOperation>
12 T inclusive_scan_over_group(Group g, T x, BinaryOperation binary_op); // (3)
13
14 template <typename Group, typename V, typename T, typename BinaryOperation>
15 T inclusive_scan_over_group(Group g, V x, BinaryOperation binary_op,
16                             T init); // (4)

```

1. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true, `InPtr` and `OutPtr` are pointers to fundamental types, and `BinaryOperation` is a SYCL function object type.

*Mandates:* `binary_op(*first, *first)` must return a value of type `std::iterator_traits<OutPtr>::value_type`.

*Preconditions:* `first`, `last`, `result` and the type of `binary_op` must be the same for all work-items in group  $g$ . `binary_op` must be an instance of a SYCL function object.



Note that `first` may be equal to `result`.

*Effects:* Blocks until all work-items in group  $g$  have reached this synchronization point, then executes the algorithm.

The value written to `result + i` is the inclusive scan of the values resulting from dereferencing the first  $i$  values in the range `[first, last)`, using the operator `binary_op`. The scan is computed using a generalized noncommutative sum as defined in standard C++.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* A pointer to the end of the output range.

2. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true, `InPtr` and `OutPtr` are pointers to fundamental types, `BinaryOperation` is a SYCL function object type, and `T` is a fundamental type.

*Mandates:* `binary_op(init, *first)` must return a value of type `T`.

*Preconditions:* `first`, `last`, `result`, `init` and the type of `binary_op` must be the same for all work-items in group  $g$ . `binary_op` must be an instance of a SYCL function object.





Note that `first` may be equal to `result`.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

The value written to `result + i` is the inclusive scan of the values resulting from dereferencing the first `i` values in the range `[first, last)` and an initial value specified by `init`, using the operator `binary_op`. The scan is computed using a generalized noncommutative sum as defined in standard C++.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* A pointer to the end of the output range.

3. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true, `T` is a fundamental type, and `BinaryOperation` is a SYCL function object type.

*Mandates:* `binary_op(x, x)` must return a value of type `T`.

*Preconditions:* `binary_op` must be an instance of a SYCL function object.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* The value returned on work-item `i` is the inclusive scan of the first `i` values in group `g`, using the operator `binary_op`. The scan is computed using a generalized noncommutative sum as defined in standard C++. For multi-dimensional groups, the order of work-items in group `g` is determined by their linear id.

4. *Constraints:* Available only if `sycl::is_group_v<std::decay_t<Group>>` is true, `V` is a fundamental type, `BinaryOperation` is a SYCL function object type, and `T` is a fundamental type.

*Mandates:* `binary_op(init, x)` must return a value of type `T`.

*Preconditions:* `binary_op` must be an instance of a SYCL function object.

*Effects:* Blocks until all work-items in group `g` have reached this synchronization point, then executes the algorithm.

*Synchronization:* The call to this function in each work-item happens before the algorithm begins execution. The completion of the algorithm happens before any work-item blocking on the same synchronization point is unblocked.

*Returns:* The value returned on work-item `i` is the inclusive scan of the first `i` values in group `g` and an initial value specified by `init`, using the operator `binary_op`. The scan is computed using a generalized noncommutative sum as defined in standard C++. For multi-dimensional groups, the order of work-items in group `g` is determined by their linear id.

## 4.17.4. Math functions

This section describes the math functions that are available in the `sycl` namespace in both host and device code.

The function descriptions in this section use the term *writable address space* to represent the following

address spaces:

- `access::address_space::global_space`
- `access::address_space::local_space`
- `access::address_space::private_space`
- `access::address_space::generic_space`

The descriptions in this section use the type name `__swizzle__` to refer to the classes defined in [Section 4.14.2.4](#). This type can be any instantiation of the class templates named `__writeable_swizzle__` or `__const_swizzle__` in that section, so long as the instantiation satisfies the constraints listed in the function's description.

## acos

```
float acos(float x)           (1)
double acos(double x)        (2)
half acos(half x)            (3)

template<typename NonScalar>  (4)
/*return-type*/ acos(NonScalar x)
```

### Overloads (1) - (3):

*Returns:* The inverse cosine of `x`.

### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the inverse cosine of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

## acosh

```
float acosh(float x)          (1)
double acosh(double x)        (2)
half acosh(half x)            (3)

template<typename NonScalar>  (4)
/*return-type*/ acosh(NonScalar x)
```

### Overloads (1) - (3):

*Returns:* The inverse hyperbolic cosine of `x`.

### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the inverse hyperbolic cosine of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### `acospi`

```
float acospi(float x)           (1)
double acospi(double x)        (2)
half acospi(half x)            (3)

template<typename NonScalar>    (4)
/*return-type*/ acospi(NonScalar x)
```

#### Overloads (1) - (3):

*Returns:* The value  $\text{acos}(x) / \pi$ .

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the value  $\text{acos}(x[i]) / \pi$ .

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### `asin`

```
float asin(float x)           (1)
double asin(double x)        (2)
half asin(half x)            (3)

template<typename NonScalar>    (4)
/*return-type*/ asin(NonScalar x)
```

#### Overloads (1) - (3):

*Returns:* The inverse sine of `x`.

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the inverse sine of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

### asinh

```
float asinh(float x)           (1)
double asinh(double x)        (2)
half asinh(half x)             (3)

template<typename NonScalar>   (4)
/*return-type*/ asinh(NonScalar x)
```

#### Overloads (1) - (3):

*Returns:* The inverse hyperbolic sine of `x`.

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the inverse hyperbolic sine of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

### asinpi

```
float asinpi(float x)          (1)
double asinpi(double x)        (2)
half asinpi(half x)             (3)

template<typename NonScalar>   (4)
/*return-type*/ asinpi(NonScalar x)
```

#### Overloads (1) - (3):

*Returns:* The value `asin(x) / π`.

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the value `asin(x[i]) / π`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**atan**

```

float atan(float y_over_x)           (1)
double atan(double y_over_x)        (2)
half atan(half y_over_x)            (3)

template<typename NonScalar>         (4)
/*return-type*/ atan(NonScalar y_over_x)

```

**Overloads (1) - (3):**

*Returns:* The inverse tangent of the input.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of the input, the inverse tangent of the element.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**atan2**

```

float atan2(float y, float x)        (1)
double atan2(double y, double x)     (2)
half atan2(half y, half x)           (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ atan2(NonScalar1 y, NonScalar2 x)

```

**Overloads (1) - (3):**

*Returns:* The arc tangent of  $y / x$ .

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for `NonScalar1` and `NonScalar2`:
  - Both `NonScalar1` and `NonScalar2` are `marray`; or
  - `NonScalar1` and `NonScalar2` are any combination of `vec` and the `__swizzle__` type;
- `NonScalar1` and `NonScalar2` have the same number of elements;
- `NonScalar1` and `NonScalar2` have the same element type; and
- The element type of `NonScalar1` and `NonScalar2` is `float`, `double`, or `half`.

*Returns:* For each element of `x` and `y`, the arc tangent of  $y[i] / x[i]$ .

The return type is `NonScalar1` unless `NonScalar1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**atanh**

```

float atanh(float x)           (1)
double atanh(double x)        (2)
half atanh(half x)            (3)

template<typename NonScalar>   (4)
/*return-type*/ atanh(NonScalar x)

```

**Overloads (1) - (3):**

*Returns:* The hyperbolic inverse tangent of **x**.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**, **double**, or **half**.

*Returns:* For each element of **x**, the hyperbolic inverse tangent of **x[i]**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**atanpi**

```

float atanpi(float x)          (1)
double atanpi(double x)        (2)
half atanpi(half x)            (3)

template<typename NonScalar>   (4)
/*return-type*/ atanpi(NonScalar x)

```

**Overloads (1) - (3):**

*Returns:* The value **atan(x) /  $\pi$** .

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**, **double**, or **half**.

*Returns:* For each element of **x**, the value **atan(x[i]) /  $\pi$** .

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**atan2pi**

```

float atan2pi(float y, float x) (1)
double atan2pi(double y, double x) (2)
half atan2pi(half y, half x) (3)

```

```
template<typename NonScalar1, typename NonScalar2>    (4)
/*return-type*/ atan2pi(NonScalar1 y, NonScalar2 x)
```

**Overloads (1) - (3):**

*Returns:* The value  $\text{atan2}(y, x) / \pi$ .

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for `NonScalar1` and `NonScalar2`:
  - Both `NonScalar1` and `NonScalar2` are `marray`; or
  - `NonScalar1` and `NonScalar2` are any combination of `vec` and the `__swizzle__` type;
- `NonScalar1` and `NonScalar2` have the same number of elements;
- `NonScalar1` and `NonScalar2` have the same element type; and
- The element type of `NonScalar1` and `NonScalar2` is `float`, `double`, or `half`.

*Returns:* For each element of `x` and `y`, the value  $\text{atan2}(y[i], x[i]) / \pi$ .

The return type is `NonScalar1` unless `NonScalar1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**cbrt**

```
float cbrt(float x)                (1)
double cbrt(double x)             (2)
half cbrt(half x)                 (3)

template<typename NonScalar>      (4)
/*return-type*/ cbrt(NonScalar x)
```

**Overloads (1) - (3):**

*Returns:* The cube-root of `x`.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the cube-root of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**ceil**

```
float ceil(float x)                (1)
double ceil(double x)             (2)
```

```
half ceil(half x) (3)

template<typename NonScalar> (4)
/*return-type*/ ceil(NonScalar x)
```

**Overloads (1) - (3):**

*Returns:* The value `x` rounded to an integral value using the round to positive infinity rounding mode.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the value `x[i]` rounded to an integral value using the round to positive infinity rounding mode.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**copysign**

```
float copysign(float x, float y) (1)
double copysign(double x, double y) (2)
half copysign(half x, half y) (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ copysign(NonScalar1 x, NonScalar2 y)
```

**Overloads (1) - (3):**

*Returns:* The value of `x` with its sign changed to match the sign of `y`.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for `NonScalar1` and `NonScalar2`:
  - Both `NonScalar1` and `NonScalar2` are `marray`; or
  - `NonScalar1` and `NonScalar2` are any combination of `vec` and the `__swizzle__` type;
- `NonScalar1` and `NonScalar2` have the same number of elements;
- `NonScalar1` and `NonScalar2` have the same element type; and
- The element type of `NonScalar1` and `NonScalar2` is `float`, `double`, or `half`.

*Returns:* For each element of `x` and `y`, the value of `x[i]` with its sign changed to match the sign of `y[i]`.

The return type is `NonScalar1` unless `NonScalar1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.



**cos**

```

float cos(float x)           (1)
double cos(double x)        (2)
half cos(half x)            (3)

template<typename NonScalar> (4)
/*return-type*/ cos(NonScalar x)

```

**Overloads (1) - (3):**

*Returns:* The cosine of **x**.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**, **double**, or **half**.

*Returns:* For each element of **x**, the cosine of **x[i]**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**cosh**

```

float cosh(float x)         (1)
double cosh(double x)      (2)
half cosh(half x)          (3)

template<typename NonScalar> (4)
/*return-type*/ cosh(NonScalar x)

```

**Overloads (1) - (3):**

*Returns:* The hyperbolic cosine of **x**.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**, **double**, or **half**.

*Returns:* For each element of **x**, the hyperbolic cosine of **x[i]**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**cospi**

```

float cospi(float x)       (1)
double cospi(double x)    (2)
half cospi(half x)        (3)

```

```
template<typename NonScalar>           (4)
/*return-type*/ cospi(NonScalar x)
```

**Overloads (1) - (3):**

*Returns:* The value  $\cos(\pi * x)$ .

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the value  $\cos(\pi * x[i])$ .

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**erfc**

```
float erfc(float x)           (1)
double erfc(double x)        (2)
half erfc(half x)            (3)

template<typename NonScalar> (4)
/*return-type*/ erfc(NonScalar x)
```

**Overloads (1) - (3):**

*Returns:* The complementary error function of `x`.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the complementary error function of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**erf**

```
float erf(float x)           (1)
double erf(double x)        (2)
half erf(half x)            (3)

template<typename NonScalar> (4)
/*return-type*/ erf(NonScalar x)
```

**Overloads (1) - (3):**

*Returns:* The error function of `x` (encountered in integrating the normal distribution).

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the error function of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**exp**

```
float exp(float x)           (1)
double exp(double x)        (2)
half exp(half x)            (3)

template<typename NonScalar> (4)
/*return-type*/ exp(NonScalar x)
```

**Overloads (1) - (3):**

*Returns:* The base-*e* exponential of `x`.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the base-*e* exponential of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**exp2**

```
float exp2(float x)          (1)
double exp2(double x)        (2)
half exp2(half x)            (3)

template<typename NonScalar> (4)
/*return-type*/ exp2(NonScalar x)
```

**Overloads (1) - (3):**

*Returns:* The base-2 exponential of `x`.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the base-2 exponential of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### `exp10`

```
float exp10(float x)           (1)
double exp10(double x)        (2)
half exp10(half x)            (3)

template<typename NonScalar>   (4)
/*return-type*/ exp10(NonScalar x)
```

#### Overloads (1) - (3):

*Returns:* The base-10 exponential of `x`.

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the base-10 exponential of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### `expm1`

```
float expm1(float x)           (1)
double expm1(double x)        (2)
half expm1(half x)            (3)

template<typename NonScalar>   (4)
/*return-type*/ expm1(NonScalar x)
```

#### Overloads (1) - (3):

*Returns:* The value  $e^x - 1.0$ .

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of **x**, the value  $e^{x[i]} - 1.0$ .

The return type is **NonScalar** unless **NonScalar** is the `__swizzle__` type, in which case the return type is the corresponding **vec**.

#### **fabs**

```
float fabs(float x)           (1)
double fabs(double x)        (2)
half fabs(half x)            (3)

template<typename NonScalar> (4)
/*return-type*/ fabs(NonScalar x)
```

#### **Overloads (1) - (3):**

*Returns:* The absolute value of **x**.

#### **Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the `__swizzle__` type; and
- The element type is **float**, **double**, or **half**.

*Returns:* For each element of **x**, the absolute value of **x[i]**.

The return type is **NonScalar** unless **NonScalar** is the `__swizzle__` type, in which case the return type is the corresponding **vec**.

#### **fdim**

```
float fdim(float x, float y)           (1)
double fdim(double x, double y)        (2)
half fdim(half x, half y)             (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ fdim(NonScalar1 x, NonScalar2 y)
```

#### **Overloads (1) - (3):**

*Returns:* The value **x - y** if **x > y**, otherwise +0.

#### **Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for **NonScalar1** and **NonScalar2**:
  - Both **NonScalar1** and **NonScalar2** are **marray**; or
  - **NonScalar1** and **NonScalar2** are any combination of **vec** and the `__swizzle__` type;
- **NonScalar1** and **NonScalar2** have the same number of elements;
- **NonScalar1** and **NonScalar2** have the same element type; and
- The element type of **NonScalar1** and **NonScalar2** is **float**, **double**, or **half**.

*Returns:* For each element of **x** and **y**, the value **x[i] - y[i]** if **x[i] > y[i]**, otherwise +0.

The return type is **NonScalar1** unless **NonScalar1** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

#### floor

```
float floor(float x) (1)
double floor(double x) (2)
half floor(half x) (3)

template<typename NonScalar> (4)
/*return-type*/ floor(NonScalar x)
```

#### Overloads (1) - (3):

*Returns:* The value **x** rounded to an integral value using the round to negative infinity rounding mode.

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**, **double**, or **half**.

*Returns:* For each element of **x**, the value **x[i]** rounded to an integral value using the round to negative infinity rounding mode.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

#### fma

```
float fma(float a, float b, float c) (1)
double fma(double a, double b, double c) (2)
half fma(half a, half b, half c) (3)

template<typename NonScalar1, typename NonScalar2, typename NonScalar3> (4)
/*return-type*/ fma(NonScalar1 a, NonScalar2 b, NonScalar3 c)
```

#### Overloads (1) - (3):

*Returns:* The correctly rounded floating-point representation of the sum of **c** with the infinitely precise product of **a** and **b**. Rounding of intermediate products shall not occur. Edge case behavior is per the IEEE 754-2008 standard.

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for **NonScalar1**, **NonScalar2**, and **NonScalar3**:
  - **NonScalar1**, **NonScalar2**, and **NonScalar3** are each **marray**; or
  - **NonScalar1**, **NonScalar2**, and **NonScalar3** are any combination of **vec** and the **\_\_swizzle\_\_** type;
- **NonScalar1**, **NonScalar2**, and **NonScalar3** have the same number of elements;

- `NonScalar1`, `NonScalar2`, and `NonScalar3` have the same element type; and
- The element type of `NonScalar1`, `NonScalar2`, and `NonScalar3` is `float`, `double`, or `half`.

*Returns:* For each element of `a`, `b`, and `c`; the correctly rounded floating-point representation of the sum of `c[i]` with the infinitely precise product of `a[i]` and `b[i]`. Rounding of intermediate products shall not occur. Edge case behavior is per the IEEE 754-2008 standard.

The return type is `NonScalar1` unless `NonScalar1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### fmax

```
float fmax(float x, float y)           (1)
double fmax(double x, double y)       (2)
half fmax(half x, half y)             (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ fmax(NonScalar1 x, NonScalar2 y)

template<typename NonScalar> (5)
/*return-type*/ fmax(NonScalar x, NonScalar::value_type y)
```

#### Overloads (1) - (3):

*Returns:* `y` if `x < y`, otherwise `x`. If one argument is a NaN, returns the other argument. If both arguments are NaNs, returns a NaN.

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for `NonScalar1` and `NonScalar2`:
  - Both `NonScalar1` and `NonScalar2` are `marray`; or
  - `NonScalar1` and `NonScalar2` are any combination of `vec` and the `__swizzle__` type;
- `NonScalar1` and `NonScalar2` have the same number of elements;
- `NonScalar1` and `NonScalar2` have the same element type; and
- The element type of `NonScalar1` and `NonScalar2` is `float`, `double`, or `half`.

*Returns:* For each element of `x` and `y`, the value `y[i]` if `x[i] < y[i]`, otherwise `x[i]`. If one element is a NaN, the result is the other element. If both elements are NaNs, the result is NaN.

The return type is `NonScalar1` unless `NonScalar1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### Overload (5):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the value `y` if `x[i] < y`, otherwise `x[i]`. If one value is a NaN, the result is the other value. If both value are NaNs, the result is a NaN.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the

corresponding **vec**.

### fmin

```
float fmin(float x, float y) (1)
double fmin(double x, double y) (2)
half fmin(half x, half y) (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ fmin(NonScalar1 x, NonScalar2 y)

template<typename NonScalar> (5)
/*return-type*/ fmin(NonScalar x, NonScalar::value_type y)
```

#### Overloads (1) - (3):

Returns: **y** if **y < x**, otherwise **x**. If one argument is a NaN, returns the other argument. If both arguments are NaNs, returns a NaN.

#### Overload (4):

Constraints: Available only if all of the following conditions are met:

- One of the following conditions must hold for **NonScalar1** and **NonScalar2**:
  - Both **NonScalar1** and **NonScalar2** are **marray**; or
  - **NonScalar1** and **NonScalar2** are any combination of **vec** and the **\_\_swizzle\_\_** type;
- **NonScalar1** and **NonScalar2** have the same number of elements;
- **NonScalar1** and **NonScalar2** have the same element type; and
- The element type of **NonScalar1** and **NonScalar2** is **float**, **double**, or **half**.

Returns: For each element of **x** and **y**, the value **y[i]** if **y[i] < x[i]**, otherwise **x[i]**. If one element is a NaN, the result is the other element. If both elements are NaNs, the result is NaN.

The return type is **NonScalar1** unless **NonScalar1** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

#### Overload (5):

Constraints: Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**, **double**, or **half**.

Returns: For each element of **x**, the value **y** if **y < x[i]**, otherwise **x[i]**. If one value is a NaN, the result is the other value. If both value are NaNs, the result is a NaN.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

### fmod

```
float fmod(float x, float y) (1)
double fmod(double x, double y) (2)
```



```
half fmod(half x, half y) (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ fmod(NonScalar1 x, NonScalar2 y)
```

**Overloads (1) - (3):**

*Returns:* The value  $x - y * \text{trunc}(x/y)$ .

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for `NonScalar1` and `NonScalar2`:
  - Both `NonScalar1` and `NonScalar2` are `marray`; or
  - `NonScalar1` and `NonScalar2` are any combination of `vec` and the `__swizzle__` type;
- `NonScalar1` and `NonScalar2` have the same number of elements;
- `NonScalar1` and `NonScalar2` have the same element type; and
- The element type of `NonScalar1` and `NonScalar2` is `float`, `double`, or `half`.

*Returns:* For each element of `x` and `y`, the value  $x[i] - y[i] * \text{trunc}(x[i]/y[i])$ .

The return type is `NonScalar1` unless `NonScalar1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**fract**

```
template<typename Ptr> (1)
float fract(float x, Ptr iptr)

template<typename Ptr> (2)
double fract(double x, Ptr iptr)

template<typename Ptr> (3)
half fract(half x, Ptr iptr)

template<typename NonScalar, typename Ptr> (4)
/*return-type*/ fract(NonScalar x, Ptr iptr)
```

**Overloads (1) - (3):**

*Constraints:* Available only if `Ptr` is one of the following:

- A C++ cv-unqualified pointer to the same type as `x`; or
- A `multi_ptr` with `ElementType` equal to the same type as `x` and with `Space` equal to one of the *writable address spaces* as defined above.

*Effects:* Writes the value  $\text{floor}(x)$  to `iptr`.

*Returns:* The value  $\text{fmin}(x - \text{floor}(x), \text{nextafter}(T\{1.0\}, T\{0.0\}))$ , where `T` is the type of `x`.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type with element type `float`, `double`, or `half`;
- `Ptr` is one of the following:
  - A C++ cv-unqualified pointer to `NonScalar`, unless `NonScalar` is the `__swizzle__` type, in which case it is a cv-unqualified pointer to the corresponding `vec`; or
  - A `multi_ptr` where:
    - The `ElementType` is equal to `NonScalar`, unless `NonScalar` is the `__swizzle__` type, in which case the `ElementType` is the corresponding `vec`; and
    - The `Space` is equal to one of the *writable address spaces* as defined above.

**Effects:** Writes the value `floor(x)` to `iptr`.

**Returns:** For each element of `x`, the value `fmin(x[i] - floor(x[i]), nextafter(T{1.0}, T{0.0}))`, where `T` is the element type of `x`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

### frexp

```

template<typename Ptr>                                (1)
float frexp(float x, Ptr exp)

template<typename Ptr>                                (2)
double frexp(double x, Ptr exp)

template<typename Ptr>                                (3)
half frexp(half x, Ptr exp)

template<typename NonScalar, typename Ptr>            (4)
/*return-type*/ frexp(NonScalar x, Ptr exp)

```

#### Overloads (1) - (3):

**Constraints:** Available only if `Ptr` is one of the following:

- A C++ cv-unqualified pointer to `int`; or
- A `multi_ptr` with `ElementType` of `int` and with `Space` equal to one of the *writable address spaces* as defined above.

**Effects:** Extracts the mantissa and exponent from `x`. The mantissa is a floating point number whose magnitude is in the interval  $[0.5, 1)$  or 0. The extracted mantissa and exponent are such that  $\text{mantissa} * 2^{\text{exp}}$  equals `x`. The exponent is written to `exp`.

**Returns:** The mantissa of `x`.

#### Overload (4):

**Constraints:** Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type with element type `float`, `double`, or `half`;
- `Ptr` is one of the following:
  - (If `NonScalar` is `marray`): A C++ cv-unqualified pointer to `marray` of `int` with the same number of elements as `NonScalar`; or

- (If `NonScalar` is `vec` or the `__swizzle__` type): A C++ cv-unqualified pointer to `vec` of `int32_t` with the same number of elements as `NonScalar`; or
- (If `NonScalar` is `marray`): A `multi_ptr` whose `Space` is equal to one of the *writable address spaces* as defined above and whose `ElementType` is `marray` of `int` with the same number of elements as `NonScalar`; or
- (If `NonScalar` is `vec` or the `__swizzle__` type): A `multi_ptr` whose `Space` is equal to one of the *writable address spaces* as defined above and whose `ElementType` is `vec` of `int32_t` with the same number of elements as `NonScalar`.

**Effects:** Extracts the mantissa and exponent from each element of `x`. Each mantissa is a floating point number whose magnitude is in the interval  $[0.5, 1)$  or 0. Each extracted mantissa and exponent are such that  $\text{mantissa} * 2^{\text{exp}}$  equals `x[i]`. The exponent of each element of `x` is written to `exp`.

**Returns:** For each element of `x`, the mantissa of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

### hypot

```
float hypot(float x, float y)           (1)
double hypot(double x, double y)       (2)
half hypot(half x, half y)             (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ hypot(NonScalar1 x, NonScalar2 y)
```

#### Overloads (1) - (3):

**Returns:** The value of the square root of  $x^2 + y^2$  without undue overflow or underflow.

#### Overload (4):

**Constraints:** Available only if all of the following conditions are met:

- One of the following conditions must hold for `NonScalar1` and `NonScalar2`:
  - Both `NonScalar1` and `NonScalar2` are `marray`; or
  - `NonScalar1` and `NonScalar2` are any combination of `vec` and the `__swizzle__` type;
- `NonScalar1` and `NonScalar2` have the same number of elements;
- `NonScalar1` and `NonScalar2` have the same element type; and
- The element type of `NonScalar1` and `NonScalar2` is `float`, `double`, or `half`.

**Returns:** For each element of `x` and `y`, the value of the square root of  $x[i]^2 + y[i]^2$  without undue overflow or underflow.

The return type is `NonScalar1` unless `NonScalar1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

### ilogb

```
int ilogb(float x)           (1)
int ilogb(double x)         (2)
```

```
int ilogb(half x) (3)

template<typename NonScalar> (4)
/*return-type*/ ilogb(NonScalar x)
```

**Overloads (1) - (3):**

*Returns:* Compute the integral part of  $\log_r|x|$  and return the result as an integer, where  $r$  is the value returned by `std::numeric_limits<decltype(x)>::radix`.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, compute the integral part of  $\log_r|x[i]|$  and return the result as an integer, where  $r$  is the value returned by `std::numeric_limits<NonScalar::value_type>::radix`.

The return type depends on `NonScalar`. If `NonScalar` is `marray`, the return type is `marray` of `int` with the same number of element as `NonScalar`. If `NonScalar` is `vec` or the `__swizzle__` type, the return type is `vec` of `int32_t` with the same number of elements as `NonScalar`.

**ldexp**

```
float ldexp(float x, int k) (1)
double ldexp(double x, int k) (2)
half ldexp(half x, int k) (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ ldexp(NonScalar1 x, NonScalar2 k)

template<typename NonScalar> (5)
/*return-type*/ ldexp(NonScalar x, int k)
```

**Overloads (1) - (3):**

*Returns:* The value `x` multiplied by  $2^k$ .

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar1` is `marray`, `vec`, or the `__swizzle__` type;
- The element type of `NonScalar1` is `float`, `double`, or `half`;
- If `NonScalar1` is `marray`, `NonScalar2` is `marray` of `int` with the same number of elements as `NonScalar1`; and
- If `NonScalar1` is `vec` or the `__swizzle__` type, `NonScalar2` is `vec` or the `__swizzle__` type of `int32_t` with the same number of elements as `NonScalar1`.

*Returns:* For each element of `x` and `k`, the value `x[i]` multiplied by  $2^{k[i]}$ .

The return type is `NonScalar1` unless `NonScalar1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**Overload (5):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type of `NonScalar` is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the value `x[i]` multiplied by  $2^k$ .

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**lgamma**

```
float lgamma(float x)           (1)
double lgamma(double x)        (2)
half lgamma(half x)            (3)

template<typename NonScalar>    (4)
/*return-type*/ lgamma(NonScalar x)
```

**Overloads (1) - (3):**

*Returns:* The natural logarithm of the absolute value of the gamma function of `x`.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the natural logarithm of the absolute value of the gamma function of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**lgamma\_r**

```
template<typename Ptr>          (1)
float lgamma_r(float x, Ptr signp)

template<typename Ptr>          (2)
double lgamma_r(double x, Ptr signp)

template<typename Ptr>          (3)
half lgamma_r(half x, Ptr signp)

template<typename NonScalar, typename Ptr> (4)
/*return-type*/ lgamma_r(NonScalar x, Ptr signp)
```

**Overloads (1) - (3):**

**Constraints:** Available only if `Ptr` is one of the following:

- A C++ cv-unqualified pointer to `int`; or
- A `multi_ptr` with `ElementType` of `int` and with `Space` equal to one of the *writable address spaces* as defined above.

**Effects:** Writes the sign of the gamma function of `x` to `signp`.

**Returns:** The natural logarithm of the absolute value of the gamma function of `x`.

#### Overload (4):

**Constraints:** Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type with element type `float`, `double`, or `half`;
- `Ptr` is one of the following:
  - (If `NonScalar` is `marray`): A C++ cv-unqualified pointer to `marray` of `int` with the same number of elements as `NonScalar`; or
  - (If `NonScalar` is `vec` or the `__swizzle__` type): A C++ cv-unqualified pointer to `vec` of `int32_t` with the same number of elements as `NonScalar`; or
  - (If `NonScalar` is `marray`): A `multi_ptr` whose `Space` is equal to one of the *writable address spaces* as defined above and whose `ElementType` is `marray` of `int` with the same number of elements as `NonScalar`; or
  - (If `NonScalar` is `vec` or the `__swizzle__` type): A `multi_ptr` whose `Space` is equal to one of the *writable address spaces* as defined above and whose `ElementType` is `vec` of `int32_t` with the same number of elements as `NonScalar`.

**Effects:** Computes the gamma function for each element of `x` and writes the sign for each of these values to `signp`.

**Returns:** For each element of `x`, the natural logarithm of the absolute value of the gamma function of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### log

```
float log(float x)           (1)
double log(double x)        (2)
half log(half x)            (3)

template<typename NonScalar> (4)
/*return-type*/ log(NonScalar x)
```

#### Overloads (1) - (3):

**Returns:** The natural logarithm of `x`.

#### Overload (4):

**Constraints:** Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of **x**, the natural logarithm of **x[i]**.

The return type is **NonScalar** unless **NonScalar** is the `__swizzle__` type, in which case the return type is the corresponding **vec**.

## log2

```
float log2(float x)           (1)
double log2(double x)        (2)
half log2(half x)            (3)

template<typename NonScalar> (4)
/*return-type*/ log2(NonScalar x)
```

### Overloads (1) - (3):

*Returns:* The base 2 logarithm of **x**.

### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the `__swizzle__` type; and
- The element type is **float**, **double**, or **half**.

*Returns:* For each element of **x**, the base 2 logarithm of **x[i]**.

The return type is **NonScalar** unless **NonScalar** is the `__swizzle__` type, in which case the return type is the corresponding **vec**.

## log10

```
float log10(float x)          (1)
double log10(double x)        (2)
half log10(half x)            (3)

template<typename NonScalar> (4)
/*return-type*/ log10(NonScalar x)
```

### Overloads (1) - (3):

*Returns:* The base 10 logarithm of **x**.

### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the `__swizzle__` type; and
- The element type is **float**, **double**, or **half**.

*Returns:* For each element of **x**, the base 10 logarithm of **x[i]**.

The return type is **NonScalar** unless **NonScalar** is the `__swizzle__` type, in which case the return type is the corresponding **vec**.

**log1p**

```

float log1p(float x)           (1)
double log1p(double x)        (2)
half log1p(half x)            (3)

template<typename NonScalar>   (4)
/*return-type*/ log1p(NonScalar x)

```

**Overloads (1) - (3):**

*Returns:* The value  $\log(1.0 + x)$ .

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the value  $\log(1.0 + x[i])$ .

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**logb**

```

float logb(float x)           (1)
double logb(double x)        (2)
half logb(half x)            (3)

template<typename NonScalar>   (4)
/*return-type*/ logb(NonScalar x)

```

**Overloads (1) - (3):**

*Returns:* The integral part of  $\log_r|x|$ , where `r` is the value returned by `std::numeric_limits<decltype(x)>::radix`.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the integral part of  $\log_r|x[i]|$ , where `r` is the value returned by `std::numeric_limits<NonScalar::value_type>::radix`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.



**mad**

```

float mad(float a, float b, float c) (1)
double mad(double a, double b, double c) (2)
half mad(half a, half b, half c) (3)

template<typename NonScalar1, typename NonScalar2, typename NonScalar3> (4)
/*return-type*/ mad(NonScalar1 a, NonScalar2 b, NonScalar3 c)

```

**Overloads (1) - (3):**

*Effects:* Computes the approximate value of  $a * b + c$ . Whether or how the product of  $a * b$  is rounded and how supernormal or subnormal intermediate products are handled is not defined. The **mad** function is intended to be used where speed is preferred over accuracy.

*Returns:* The approximate value of  $a * b + c$ .

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for **NonScalar1**, **NonScalar2**, and **NonScalar3**:
  - **NonScalar1**, **NonScalar2**, and **NonScalar3** are each **marray**; or
  - **NonScalar1**, **NonScalar2**, and **NonScalar3** are any combination of **vec** and the **\_\_swizzle\_\_** type;
- **NonScalar1**, **NonScalar2**, and **NonScalar3** have the same number of elements;
- **NonScalar1**, **NonScalar2**, and **NonScalar3** have the same element type; and
- The element type of **NonScalar1**, **NonScalar2**, and **NonScalar3** is **float**, **double**, or **half**.

*Returns:* For each element of **a**, **b**, and **c**; the The approximate value of  $a[i] * b[i] + c[i]$ .

The return type is **NonScalar1** unless **NonScalar1** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**maxmag**

```

float maxmag(float x, float y) (1)
double maxmag(double x, double y) (2)
half maxmag(half x, half y) (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ maxmag(NonScalar1 x, NonScalar2 y)

```

**Overloads (1) - (3):**

*Returns:* The value **x** if  $|x| > |y|$ , **y** if  $|y| > |x|$ , otherwise **fmax(x, y)**.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for **NonScalar1** and **NonScalar2**:
  - Both **NonScalar1** and **NonScalar2** are **marray**; or
  - **NonScalar1** and **NonScalar2** are any combination of **vec** and the **\_\_swizzle\_\_** type;

- `NonScalar1` and `NonScalar2` have the same number of elements;
- `NonScalar1` and `NonScalar2` have the same element type; and
- The element type of `NonScalar1` and `NonScalar2` is `float`, `double`, or `half`.

*Returns:* For each element of `x` and `y`, the value `x[i]` if  $|x[i]| > |y[i]|$ , `y[i]` if  $|y[i]| > |x[i]|$ , otherwise `fmax(x[i], y[i])`.

The return type is `NonScalar1` unless `NonScalar1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### minmag

```
float minmag(float x, float y)           (1)
double minmag(double x, double y)       (2)
half minmag(half x, half y)             (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ minmag(NonScalar1 x, NonScalar2 y)
```

#### Overloads (1) - (3):

*Returns:* The value `x` if  $|x| < |y|$ , `y` if  $|y| < |x|$ , otherwise `fmin(x, y)`.

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for `NonScalar1` and `NonScalar2`:
  - Both `NonScalar1` and `NonScalar2` are `marray`; or
  - `NonScalar1` and `NonScalar2` are any combination of `vec` and the `__swizzle__` type;
- `NonScalar1` and `NonScalar2` have the same number of elements;
- `NonScalar1` and `NonScalar2` have the same element type; and
- The element type of `NonScalar1` and `NonScalar2` is `float`, `double`, or `half`.

*Returns:* For each element of `x` and `y`, the value `x[i]` if  $|x[i]| < |y[i]|$ , `y[i]` if  $|y[i]| < |x[i]|$ , otherwise `fmin(x[i], y[i])`.

The return type is `NonScalar1` unless `NonScalar1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### modf

```
template<typename Ptr> (1)
float modf(float x, Ptr iptr)

template<typename Ptr> (2)
double modf(double x, Ptr iptr)

template<typename Ptr> (3)
half modf(half x, Ptr iptr)

template<typename NonScalar, typename Ptr> (4)
```

```
/*return-type*/ modf(NonScalar x, Ptr iptr)
```

### Overloads (1) - (3):

**Constraints:** Available only if **Ptr** is one of the following:

- A C++ cv-unqualified pointer to the same type as **x**; or
- A **multi\_ptr** with **ElementType** equal to the same type as **x** and with **Space** equal to one of the *writable address spaces* as defined above.

**Effects:** The **modf** function breaks the argument **x** into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part to the object pointed to by **iptr**.

**Returns:** The fractional part of the argument **x**.

### Overload (4):

**Constraints:** Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type with element type **float**, **double**, or **half**;
- **Ptr** is one of the following:
  - A C++ cv-unqualified pointer to **NonScalar**, unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case it is a cv-unqualified pointer to the corresponding **vec**; or
  - A **multi\_ptr** where:
    - The **ElementType** is equal to **NonScalar**, unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the **ElementType** is the corresponding **vec**; and
    - The **Space** is equal to one of the *writable address spaces* as defined above.

**Effects:** The **modf** function breaks each element of the argument **x** into integral and fractional parts, each of which has the same sign as the element. It stores the integral parts of each element to the object pointed to by **iptr**.

**Returns:** The fractional parts of each element of the argument **x**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

## nan

```
float nan(uint32_t nancode)           (1)
double nan(uint64_t nancode)         (2)
half nan(uint16_t nancode)           (3)

template<typename NonScalar>         (4)
/*return-type*/ nan(NonScalar nancode)
```

### Overloads (1) - (3):

**Returns:** A quiet NaN. The **nancode** may be placed in the significand of the resulting NaN.

### Overload (4):

**Constraints:** Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and

- The element type is `uint32_t`, `uint64_t`, or `uint16_t`.

*Returns:* A quiet NaN for each element of `nancode`. Each `nancode[i]` may be placed in the significand of the resulting NaN.

The return type depends on `NonScalar`:

<b>NonScalar</b>	<b>Return Type</b>
<code>marray&lt;uint32_t, N&gt;</code>	<code>marray&lt;float, N&gt;</code>
<code>marray&lt;uint64_t, N&gt;</code>	<code>marray&lt;double, N&gt;</code>
<code>marray&lt;uint16_t, N&gt;</code>	<code>marray&lt;half, N&gt;</code>
<code>vec&lt;uint32_t, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;uint32_t, N&gt;</code>	<code>vec&lt;float, N&gt;</code>
<code>vec&lt;uint64_t, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;uint64_t, N&gt;</code>	<code>vec&lt;double, N&gt;</code>
<code>vec&lt;uint16_t, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;uint16_t, N&gt;</code>	<code>vec&lt;half, N&gt;</code>

#### `nextafter`

```

float nextafter(float x, float y)           (1)
double nextafter(double x, double y)       (2)
half nextafter(half x, half y)             (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ nextafter(NonScalar1 x, NonScalar2 y)

```

#### Overloads (1) - (3):

*Returns:* The next representable floating-point value following `x` in the direction of `y`. Thus, if `y` is less than `x`, `nextafter` returns the largest representable floating-point number less than `x`.

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for `NonScalar1` and `NonScalar2`:
  - Both `NonScalar1` and `NonScalar2` are `marray`; or
  - `NonScalar1` and `NonScalar2` are any combination of `vec` and the `__swizzle__` type;
- `NonScalar1` and `NonScalar2` have the same number of elements;
- `NonScalar1` and `NonScalar2` have the same element type; and
- The element type of `NonScalar1` and `NonScalar2` is `float`, `double`, or `half`.

*Returns:* For each element of `x` and `y`, the next representable floating-point value following `x[i]` in the direction of `y[i]`.

The return type is `NonScalar1` unless `NonScalar1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**pow**

```

float pow(float x, float y)           (1)
double pow(double x, double y)       (2)
half pow(half x, half y)             (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ pow(NonScalar1 x, NonScalar2 y)

```

**Overloads (1) - (3):**

*Returns:* The value of `x` raised to the power `y`.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for `NonScalar1` and `NonScalar2`:
  - Both `NonScalar1` and `NonScalar2` are `marray`; or
  - `NonScalar1` and `NonScalar2` are any combination of `vec` and the `__swizzle__` type;
- `NonScalar1` and `NonScalar2` have the same number of elements;
- `NonScalar1` and `NonScalar2` have the same element type; and
- The element type of `NonScalar1` and `NonScalar2` is `float`, `double`, or `half`.

*Returns:* For each element of `x` and `y`, the value of `x[i]` raised to the power `y[i]`.

The return type is `NonScalar1` unless `NonScalar1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**pown**

```

float pown(float x, int y)           (1)
double pown(double x, int y)       (2)
half pown(half x, int y)           (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ pown(NonScalar1 x, NonScalar2 y)

```

**Overloads (1) - (3):**

*Returns:* The value of `x` raised to the power `y`.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar1` is `marray`, `vec`, or the `__swizzle__` type;
- The element type of `NonScalar1` is `float`, `double`, or `half`;
- If `NonScalar1` is `marray`, `NonScalar2` is `marray` of `int` with the same number of elements as `NonScalar1`; and
- If `NonScalar1` is `vec` or the `__swizzle__` type, `NonScalar2` is `vec` or the `__swizzle__` type of `int32_t` with the same number of elements as `NonScalar1`.

*Returns:* For each element of **x** and **y**, the value of **x[i]** raised to the power **y[i]**.

The return type is **NonScalar1** unless **NonScalar1** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

#### powr

```
float powr(float x, float y)           (1)
double powr(double x, double y)       (2)
half powr(half x, half y)             (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ powr(NonScalar1 x, NonScalar2 y)
```

#### Overloads (1) - (3):

*Preconditions:* The value of **x** must be greater than or equal to zero.

*Returns:* The value of **x** raised to the power **y**.

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for **NonScalar1** and **NonScalar2**:
  - Both **NonScalar1** and **NonScalar2** are **marray**; or
  - **NonScalar1** and **NonScalar2** are any combination of **vec** and the **\_\_swizzle\_\_** type;
- **NonScalar1** and **NonScalar2** have the same number of elements;
- **NonScalar1** and **NonScalar2** have the same element type; and
- The element type of **NonScalar1** and **NonScalar2** is **float**, **double**, or **half**.

*Preconditions:* Each element of **x** must be greater than or equal to zero.

*Returns:* For each element of **x** and **y**, the value of **x[i]** raised to the power **y[i]**.

The return type is **NonScalar1** unless **NonScalar1** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

#### remainder

```
float remainder(float x, float y)      (1)
double remainder(double x, double y)   (2)
half remainder(half x, half y)         (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ remainder(NonScalar1 x, NonScalar2 y)
```

#### Overloads (1) - (3):

*Returns:* The value **r** such that  $r = x - n*y$ , where **n** is the integer nearest the exact value of **x/y**. If there are two integers closest to **x/y**, **n** shall be the even one. If **r** is zero, it is given the same sign as **x**.

#### Overload (4):

**Constraints:** Available only if all of the following conditions are met:

- One of the following conditions must hold for `NonScalar1` and `NonScalar2`:
  - Both `NonScalar1` and `NonScalar2` are `marray`; or
  - `NonScalar1` and `NonScalar2` are any combination of `vec` and the `__swizzle__` type;
- `NonScalar1` and `NonScalar2` have the same number of elements;
- `NonScalar1` and `NonScalar2` have the same element type; and
- The element type of `NonScalar1` and `NonScalar2` is `float`, `double`, or `half`.

**Returns:** For each element of `x` and `y`, the value `r` such that  $r = x[i] - n*y[i]$ , where `n` is the integer nearest the exact value of  $x[i]/y[i]$ . If there are two integers closest to  $x[i]/y[i]$ , `n` shall be the even one. If `r` is zero, it is given the same sign as `x[i]`.

The return type is `NonScalar1` unless `NonScalar1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### remquo

```
template<typename Ptr> (1)
float remquo(float x, float y, Ptr quo)

template<typename Ptr> (2)
double remquo(double x, double y, Ptr quo)

template<typename Ptr> (3)
half remquo(half x, half y, Ptr quo)

template<typename NonScalar1, typename NonScalar2, typename Ptr> (4)
/*return-type*/ remquo(NonScalar1 x, NonScalar2 y, Ptr quo)
```

#### Overloads (1) - (3):

**Constraints:** Available only if `Ptr` is one of the following:

- A C++ cv-unqualified pointer to `int`; or
- A `multi_ptr` with `ElementType` of `int` and with `Space` equal to one of the *writable address spaces* as defined above.

**Effects:** Computes the value `r` such that  $r = x - k*y$ , where `k` is the integer nearest the exact value of  $x/y$ . If there are two integers closest to  $x/y$ , `k` shall be the even one. If `r` is zero, it is given the same sign as `x`. This is the same value that is returned by the `remainder` function. The `remquo` function also calculates the lower seven bits of the integral quotient  $x/y$  and gives that value the same sign as  $x/y$ . It stores this signed value to the object pointed to by `quo`.

**Returns:** The value `r` defined above.

#### Overload (4):

**Constraints:** Available only if all of the following conditions are met:

- One of the following conditions must hold for `NonScalar1` and `NonScalar2`:
  - Both `NonScalar1` and `NonScalar2` are `marray`; or
  - `NonScalar1` and `NonScalar2` are any combination of `vec` and the `__swizzle__` type;

- **Ptr** is one of the following:
  - (If **NonScalar1** is **marray**): A C++ cv-unqualified pointer to **marray** of **int** with the same number of elements as **NonScalar1**; or
  - (If **NonScalar1** is **vec** or the **\_\_swizzle\_\_** type): A C++ cv-unqualified pointer to **vec** of **int32\_t** with the same number of elements as **NonScalar1**; or
  - (If **NonScalar1** is **marray**): A **multi\_ptr** whose **Space** is equal to one of the *writable address spaces* as defined above and whose **ElementType** is **marray** of **int** with the same number of elements as **NonScalar1**; or
  - (If **NonScalar1** is **vec** or the **\_\_swizzle\_\_** type): A **multi\_ptr** whose **Space** is equal to one of the *writable address spaces* as defined above and whose **ElementType** is **vec** of **int32\_t** with the same number of elements as **NonScalar1**.

**Effects:** Computes the value **r** for each element of **x** and **y** such that  $r = x[i] - k \cdot y[i]$ , where **k** is the integer nearest the exact value of  $x[i]/y[i]$ . If there are two integers closest to  $x[i]/y[i]$ , **k** shall be the even one. If **r** is zero, it is given the same sign as **x[i]**. This is the same value that is returned by the **remainder** function. The **remquo** function also calculates the lower seven bits of the integral quotient  $x[i]/y[i]$  and gives that value the same sign as  $x[i]/y[i]$ . It stores these signed values to the object pointed to by **quo**.

**Returns:** The values of **r** defined above.

The return type is **NonScalar1** unless **NonScalar1** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

## rint

```
float rint(float x)           (1)
double rint(double x)        (2)
half rint(half x)            (3)

template<typename NonScalar> (4)
/*return-type*/ rint(NonScalar x)
```

### Overloads (1) - (3):

**Returns:** The value **x** rounded to an integral value (using round to nearest even rounding mode) in floating-point format. Refer to [section 7.1 of the OpenCL 1.2 specification document](#) for a description of the rounding modes.

### Overload (4):

**Constraints:** Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**, **double**, or **half**.

**Returns:** For each element of **x**, the value **x[i]** rounded to an integral value (using round to nearest even rounding mode) in floating-point format.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.



**rootn**

```

float rootn(float x, int y)           (1)
double rootn(double x, int y)        (2)
half rootn(half x, int y)            (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ rootn(NonScalar1 x, NonScalar2 y)

```

**Overloads (1) - (3):**

*Returns:* The value of **x** raised to the power  $1/y$ .

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar1** is **marray**, **vec**, or the **\_\_swizzle\_\_** type;
- The element type of **NonScalar1** is **float**, **double**, or **half**;
- If **NonScalar1** is **marray**, **NonScalar2** is **marray** of **int** with the same number of elements as **NonScalar1**; and
- If **NonScalar1** is **vec** or the **\_\_swizzle\_\_** type, **NonScalar2** is **vec** or the **\_\_swizzle\_\_** type of **int32\_t** with the same number of elements as **NonScalar1**.

*Returns:* For each element of **x** and **y**, the value of **x[i]** raised to the power  $1/y[i]$ .

The return type is **NonScalar1** unless **NonScalar1** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**round**

```

float round(float x)                 (1)
double round(double x)               (2)
half round(half x)                   (3)

template<typename NonScalar>         (4)
/*return-type*/ round(NonScalar x)

```

**Overloads (1) - (3):**

*Returns:* The integral value nearest to **x** rounding halfway cases away from zero, regardless of the current rounding direction.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**, **double**, or **half**.

*Returns:* For each element of **x**, the integral value nearest to **x[i]** rounding halfway cases away from zero, regardless of the current rounding direction.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**rsqrt**

```

float rsqrt(float x)           (1)
double rsqrt(double x)        (2)
half rsqrt(half x)            (3)

template<typename NonScalar>   (4)
/*return-type*/ rsqrt(NonScalar x)

```

**Overloads (1) - (3):**

*Returns:* The reciprocal square root of **x**.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**, **double**, or **half**.

*Returns:* For each element of **x**, the reciprocal square root of **x[i]**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**sin**

```

float sin(float x)           (1)
double sin(double x)        (2)
half sin(half x)            (3)

template<typename NonScalar>   (4)
/*return-type*/ sin(NonScalar x)

```

**Overloads (1) - (3):**

*Returns:* The sine of **x**.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**, **double**, or **half**.

*Returns:* For each element of **x**, the sine of **x[i]**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**sincos**

```

template<typename Ptr>           (1)

```

```

float sincos(float x, Ptr cosval)

template<typename Ptr>                                (2)
double sincos(double x, Ptr cosval)

template<typename Ptr>                                (3)
half sincos(half x, Ptr cosval)

template<typename NonScalar, typename Ptr>            (4)
/*return-type*/ sincos(NonScalar x, Ptr cosval)

```

**Overloads (1) - (3):**

*Constraints:* Available only if **Ptr** is one of the following:

- A C++ cv-unqualified pointer to the same type as **x**; or
- A **multi\_ptr** with **ElementType** equal to the same type as **x** and with **Space** equal to one of the *writable address spaces* as defined above.

*Effects:* Compute the sine and cosine of **x**. The computed cosine is written to **cosval**.

*Returns:* The sine of **x**.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type with element type **float**, **double**, or **half**;
- **Ptr** is one of the following:
  - A C++ cv-unqualified pointer to **NonScalar**, unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case it is a cv-unqualified pointer to the corresponding **vec**; or
  - A **multi\_ptr** where:
    - The **ElementType** is equal to **NonScalar**, unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the **ElementType** is the corresponding **vec**; and
    - The **Space** is equal to one of the *writable address spaces* as defined above.

*Effects:* Compute the sine and cosine of each element of **x**. The computed cosine values are written to **cosval**.

*Returns:* The sine of each element of **x**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**sinh**

```

float sinh(float x)                                (1)
double sinh(double x)                              (2)
half sinh(half x)                                  (3)

template<typename NonScalar>                        (4)
/*return-type*/ sinh(NonScalar x)

```

**Overloads (1) - (3):**

*Returns:* The hyperbolic sine of  $x$ .

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of  $x$ , the hyperbolic sine of  $x[i]$ .

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### `sinpi`

```
float sinpi(float x)           (1)
double sinpi(double x)        (2)
half sinpi(half x)            (3)

template<typename NonScalar>   (4)
/*return-type*/ sinpi(NonScalar x)
```

#### Overloads (1) - (3):

*Returns:* The value  $\sin(\pi * x)$ .

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of  $x$ , the value  $\sin(\pi * x[i])$ .

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### `sqrt`

```
float sqrt(float x)           (1)
double sqrt(double x)        (2)
half sqrt(half x)            (3)

template<typename NonScalar>   (4)
/*return-type*/ sqrt(NonScalar x)
```

#### Overloads (1) - (3):

*Returns:* The square root of  $x$ .

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the square root of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

## `tan`

```
float tan(float x)           (1)
double tan(double x)        (2)
half tan(half x)            (3)

template<typename NonScalar> (4)
/*return-type*/ tan(NonScalar x)
```

### Overloads (1) - (3):

*Returns:* The tangent of `x`.

### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the tangent of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

## `tanh`

```
float tanh(float x)          (1)
double tanh(double x)        (2)
half tanh(half x)            (3)

template<typename NonScalar> (4)
/*return-type*/ tanh(NonScalar x)
```

### Overloads (1) - (3):

*Returns:* The hyperbolic tangent of `x`.

### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the hyperbolic tangent of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

### `tanpi`

```
float tanpi(float x)           (1)
double tanpi(double x)        (2)
half tanpi(half x)            (3)

template<typename NonScalar>   (4)
/*return-type*/ tanpi(NonScalar x)
```

#### Overloads (1) - (3):

*Returns:* The value  $\tan(\pi * x)$ .

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the value  $\tan(\pi * x[i])$ .

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

### `tgamma`

```
float tgamma(float x)          (1)
double tgamma(double x)        (2)
half tgamma(half x)            (3)

template<typename NonScalar>   (4)
/*return-type*/ tgamma(NonScalar x)
```

#### Overloads (1) - (3):

*Returns:* The gamma function of `x`.

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the gamma function of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**trunc**

```

float trunc(float x)           (1)
double trunc(double x)        (2)
half trunc(half x)            (3)

template<typename NonScalar>   (4)
/*return-type*/ trunc(NonScalar x)

```

**Overloads (1) - (3):**

*Returns:* The value `x` rounded to an integral value using the round to zero rounding mode.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* For each element of `x`, the value `x[i]` rounded to an integral value using the round to zero rounding mode.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

## 4.17.5. Native precision math functions

This section describes the native precision math functions that are available in the `sycl::native` namespace in both host and device code.

The precision requirements and the set of legal input values for these functions are defined in the back-end specification. The intent is that these functions might make use of native device functionality which has better performance than their counterparts in [Section 4.17.4](#), but they may sacrifice accuracy or limit the set of legal input values.

The descriptions in this section use the type name `__swizzle__` to refer to the classes defined in [Section 4.14.2.4](#). This type can be any instantiation of the class templates named `__writeable_swizzle__` or `__const_swizzle__` in that section, so long as the instantiation satisfies the constraints listed in the function's description.

**native::cos**

```

float cos(float x)           (1)

template<typename NonScalar>   (2)
/*return-type*/ cos(NonScalar x)

```

**Overload (1):**

*Returns:* The cosine of `x`.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`.

*Returns:* For each element of `x`, the cosine of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### `native::divide`

```
float divide(float x, float y) (1)
```

```
template<typename NonScalar1, typename NonScalar2> (2)
/*return-type*/ divide(NonScalar1 x, NonScalar2 y)
```

#### **Overload (1):**

*Returns:* The value `x / y`.

#### **Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`.

*Returns:* For each element of `x` and `y`, the value `x[i] / y[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### `native::exp`

```
float exp(float x) (1)
```

```
template<typename NonScalar> (2)
/*return-type*/ exp(NonScalar x)
```

#### **Overload (1):**

*Returns:* The base-*e* exponential of `x`.

#### **Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`.

*Returns:* For each element of `x`, the base-*e* exponential of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.



**native::exp2**

```
float exp2(float x) (1)

template<typename NonScalar> (2)
/*return-type*/ exp2(NonScalar x)
```

**Overload (1):**

*Returns:* The base-2 exponential of **x**.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**.

*Returns:* For each element of **x**, the base-2 exponential of **x[i]**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**native::exp10**

```
float exp10(float x) (1)

template<typename NonScalar> (2)
/*return-type*/ exp10(NonScalar x)
```

**Overload (1):**

*Returns:* The base-10 exponential of **x**.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**.

*Returns:* For each element of **x**, the base-10 exponential of **x[i]**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**native::log**

```
float log(float x) (1)

template<typename NonScalar> (2)
/*return-type*/ log(NonScalar x)
```

**Overload (1):**

*Returns:* The natural logarithm of `x`.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`.

*Returns:* For each element of `x`, the natural logarithm of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**native::log2**

```
float log2(float x) (1)
```

```
template<typename NonScalar> (2)
/*return-type*/ log2(NonScalar x)
```

**Overload (1):**

*Returns:* The base 2 logarithm of `x`.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`.

*Returns:* For each element of `x`, the base 2 logarithm of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**native::log10**

```
float log10(float x) (1)
```

```
template<typename NonScalar> (2)
/*return-type*/ log10(NonScalar x)
```

**Overload (1):**

*Returns:* The base 10 logarithm of `x`.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and

- The element type is `float`.

*Returns:* For each element of `x`, the base 10 logarithm of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### `native::powr`

```
float powr(float x, float y) (1)

template<typename NonScalar1, typename NonScalar2> (2)
/*return-type*/ powr(NonScalar1 x, NonScalar2 y)
```

##### **Overload (1):**

*Preconditions:* The value of `x` must be greater than or equal to zero.

*Returns:* The value of `x` raised to the power `y`.

##### **Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`.

*Preconditions:* Each element of `x` must be greater than or equal to zero.

*Returns:* For each element of `x` and `y`, the value of `x[i]` raised to the power `y[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### `native::recip`

```
float recip(float x) (1)

template<typename NonScalar> (2)
/*return-type*/ recip(NonScalar x)
```

##### **Overload (1):**

*Returns:* The reciprocal of `x`.

##### **Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`.

*Returns:* For each element of `x`, the reciprocal of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**native::rsqrt**

```
float rsqrt(float x) (1)
```

```
template<typename NonScalar> (2)
/*return-type*/ rsqrt(NonScalar x)
```

**Overload (1):**

*Returns:* The reciprocal square root of **x**.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**.

*Returns:* For each element of **x**, the reciprocal square root of **x[i]**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**native::sin**

```
float sin(float x) (1)
```

```
template<typename NonScalar> (2)
/*return-type*/ sin(NonScalar x)
```

**Overload (1):**

*Returns:* The sine of **x**.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**.

*Returns:* For each element of **x**, the sine of **x[i]**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**native::sqrt**

```
float sqrt(float x) (1)
```

```
template<typename NonScalar> (2)
/*return-type*/ sqrt(NonScalar x)
```

**Overload (1):**

*Returns:* The square root of `x`.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`.

*Returns:* For each element of `x`, the square root of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**native::tan**

```
float tan(float x) (1)
```

```
template<typename NonScalar> (2)
/*return-type*/ tan(NonScalar x)
```

**Overload (1):**

*Returns:* The tangent of `x`.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`.

*Returns:* For each element of `x`, the tangent of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

## 4.17.6. Half precision math functions

This section describes the half precision math functions that are available in the `sycl::half_precision` namespace in both host and device code.

The precision requirements for these functions are defined in the backend specification. The intent is that these functions have higher performance than their counterparts in [Section 4.17.4](#), but they have lower accuracy.

The descriptions in this section use the type name `__swizzle__` to refer to the classes defined in [Section 4.14.2.4](#). This type can be any instantiation of the class templates named `__writeable_swizzle__` or `__const_swizzle__` in that section, so long as the instantiation satisfies the constraints listed in the function's description.

**half\_precision::cos**

```
float cos(float x) (1)

template<typename NonScalar> (2)
/*return-type*/ cos(NonScalar x)
```

**Overload (1):**

*Preconditions:* The value of **x** must be in the range  $[-2^{16}, +2^{16}]$ .

*Returns:* The cosine of **x**.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**.

*Preconditions:* The value of each element of **x** must be in the range  $[-2^{16}, +2^{16}]$ .

*Returns:* For each element of **x**, the cosine of **x[i]**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**half\_precision::divide**

```
float divide(float x, float y) (1)

template<typename NonScalar1, typename NonScalar2> (2)
/*return-type*/ divide(NonScalar1 x, NonScalar2 y)
```

**Overload (1):**

*Returns:* The value **x / y**.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**.

*Returns:* For each element of **x** and **y**, the value **x[i] / y[i]**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**half\_precision::exp**

```
float exp(float x) (1)

template<typename NonScalar> (2)
```

```
/*return-type*/ exp(NonScalar x)
```

**Overload (1):**

*Returns:* The base-*e* exponential of *x*.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- *NonScalar* is *marray*, *vec*, or the *\_\_swizzle\_\_* type; and
- The element type is *float*.

*Returns:* For each element of *x*, the base-*e* exponential of *x[i]*.

The return type is *NonScalar* unless *NonScalar* is the *\_\_swizzle\_\_* type, in which case the return type is the corresponding *vec*.

**half\_precision::exp2**

```
float exp2(float x) (1)
```

```
template<typename NonScalar> (2)
/*return-type*/ exp2(NonScalar x)
```

**Overload (1):**

*Returns:* The base-2 exponential of *x*.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- *NonScalar* is *marray*, *vec*, or the *\_\_swizzle\_\_* type; and
- The element type is *float*.

*Returns:* For each element of *x*, the base-2 exponential of *x[i]*.

The return type is *NonScalar* unless *NonScalar* is the *\_\_swizzle\_\_* type, in which case the return type is the corresponding *vec*.

**half\_precision::exp10**

```
float exp10(float x) (1)
```

```
template<typename NonScalar> (2)
/*return-type*/ exp10(NonScalar x)
```

**Overload (1):**

*Returns:* The base-10 exponential of *x*.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`.

*Returns:* For each element of `x`, the base-10 exponential of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### `half_precision::log`

```
float log(float x) (1)
```

```
template<typename NonScalar> (2)
/*return-type*/ log(NonScalar x)
```

##### **Overload (1):**

*Returns:* The natural logarithm of `x`.

##### **Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`.

*Returns:* For each element of `x`, the natural logarithm of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### `half_precision::log2`

```
float log2(float x) (1)
```

```
template<typename NonScalar> (2)
/*return-type*/ log2(NonScalar x)
```

##### **Overload (1):**

*Returns:* The base 2 logarithm of `x`.

##### **Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`.

*Returns:* For each element of `x`, the base 2 logarithm of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.



**half\_precision::log10**

```
float log10(float x) (1)

template<typename NonScalar> (2)
/*return-type*/ log10(NonScalar x)
```

**Overload (1):**

*Returns:* The base 10 logarithm of **x**.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**.

*Returns:* For each element of **x**, the base 10 logarithm of **x[i]**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**half\_precision::powr**

```
float powr(float x, float y) (1)

template<typename NonScalar1, typename NonScalar2> (2)
/*return-type*/ powr(NonScalar1 x, NonScalar2 y)
```

**Overload (1):**

*Preconditions:* The value of **x** must be greater than or equal to zero.

*Returns:* The value of **x** raised to the power **y**.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**.

*Preconditions:* Each element of **x** must be greater than or equal to zero.

*Returns:* For each element of **x** and **y**, the value of **x[i]** raised to the power **y[i]**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**half\_precision::recip**

```
float recip(float x) (1)

template<typename NonScalar> (2)
```

```
/*return-type*/ recip(NonScalar x)
```

**Overload (1):**

*Returns:* The reciprocal of **x**.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**.

*Returns:* For each element of **x**, the reciprocal of **x[i]**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**half\_precision::rsqrt**

```
float rsqrt(float x) (1)
```

```
template<typename NonScalar> (2)
/*return-type*/ rsqrt(NonScalar x)
```

**Overload (1):**

*Returns:* The reciprocal square root of **x**.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type; and
- The element type is **float**.

*Returns:* For each element of **x**, the reciprocal square root of **x[i]**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**half\_precision::sin**

```
float sin(float x) (1)
```

```
template<typename NonScalar> (2)
/*return-type*/ sin(NonScalar x)
```

**Overload (1):**

*Preconditions:* The value of **x** must be in the range  $[-2^{16}, +2^{16}]$ .

*Returns:* The sine of **x**.

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`.

*Preconditions:* The value of each element of `x` must be in the range  $[-2^{16}, +2^{16}]$ .

*Returns:* For each element of `x`, the sine of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### `half_precision::sqrt`

```
float sqrt(float x) (1)
```

```
template<typename NonScalar> (2)
/*return-type*/ sqrt(NonScalar x)
```

##### **Overload (1):**

*Returns:* The square root of `x`.

##### **Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`.

*Returns:* For each element of `x`, the square root of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### `half_precision::tan`

```
float tan(float x) (1)
```

```
template<typename NonScalar> (2)
/*return-type*/ tan(NonScalar x)
```

##### **Overload (1):**

*Preconditions:* The value of `x` must be in the range  $[-2^{16}, +2^{16}]$ .

*Returns:* The tangent of `x`.

##### **Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`.

*Preconditions:* The value of each element of `x` must be in the range  $[-2^{16}, +2^{16}]$ .

*Returns:* For each element of `x`, the tangent of `x[i]`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

## 4.17.7. Integer functions

This section describes the integer math functions that are available in the `sycl` namespace in both host and device code.

The descriptions in this section use the type name `__swizzle__` to refer to the classes defined in [Section 4.14.2.4](#). This type can be any instantiation of the class templates named `__writeable_swizzle__` or `__const_swizzle__` in that section, so long as the instantiation satisfies the constraints listed in the function's description.

The function descriptions in this section also use the term *generic integer type* to represent the following types:

- `char`
- `signed char`
- `short`
- `int`
- `long`
- `long long`
- `unsigned char`
- `unsigned short`
- `unsigned int`
- `unsigned long`
- `unsigned long long`
- `marray<char, N>`
- `marray<signed char, N>`
- `marray<short, N>`
- `marray<int, N>`
- `marray<long, N>`
- `marray<long long, N>`
- `marray<unsigned char, N>`
- `marray<unsigned short, N>`
- `marray<unsigned int, N>`
- `marray<unsigned long, N>`
- `marray<unsigned long long, N>`
- `vec<int8_t, N>`
- `vec<int16_t, N>`
- `vec<int32_t, N>`
- `vec<int64_t, N>`

- `vec<uint8_t, N>`
- `vec<uint16_t, N>`
- `vec<uint32_t, N>`
- `vec<uint64_t, N>`
- `__swizzle__` that is convertible to `vec<int8_t, N>`
- `__swizzle__` that is convertible to `vec<int16_t, N>`
- `__swizzle__` that is convertible to `vec<int32_t, N>`
- `__swizzle__` that is convertible to `vec<int64_t, N>`
- `__swizzle__` that is convertible to `vec<uint8_t, N>`
- `__swizzle__` that is convertible to `vec<uint16_t, N>`
- `__swizzle__` that is convertible to `vec<uint32_t, N>`
- `__swizzle__` that is convertible to `vec<uint64_t, N>`

**abs**

```
template<typename GenInt>
/*return-type*/ abs(GenInt x)
```

*Constraints:* Available only if `GenInt` is a *generic integer type* as defined above.

*Returns:* When the input is a scalar, returns  $|x|$ . Otherwise, returns  $|x[i]|$  for each element of `x`. The behavior is undefined if the result cannot be represented by the return type.

The return type is `GenInt` unless `GenInt` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**abs\_diff**

```
template<typename GenInt1, typename GenInt2>
/*return-type*/ abs_diff(GenInt1 x, GenInt2 y)
```

*Constraints:* Available only if all of the following conditions are met:

- `GenInt1` is a *generic integer type* as defined above;
- If `GenInt1` is not `vec` or the `__swizzle__` type, then `GenInt2` must be the same as `GenInt1`; and
- If `GenInt1` is `vec` or the `__swizzle__` type, then `GenInt2` must also be `vec` or the `__swizzle__` type, and both must have the same element type and the same number of elements.

*Returns:* When the inputs are scalars, returns  $|x - y|$ . Otherwise, returns  $|x[i] - y[i]|$  for each element of `x` and `y`. The subtraction is done without modulo overflow. The behavior is undefined if the result cannot be represented by the return type.

The return type is `GenInt1` unless `GenInt1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**add\_sat**

```
template<typename GenInt1, typename GenInt2>
```

```
/*return-type*/ add_sat(GenInt1 x, GenInt2 y)
```

*Constraints:* Available only if all of the following conditions are met:

- **GenInt1** is a *generic integer type* as defined above;
- If **GenInt1** is not **vec** or the **\_\_swizzle\_\_** type, then **GenInt2** must be the same as **GenInt1**; and
- If **GenInt1** is **vec** or the **\_\_swizzle\_\_** type, then **GenInt2** must also be **vec** or the **\_\_swizzle\_\_** type, and both must have the same element type and the same number of elements.

*Returns:* When the inputs are scalars, returns  $x + y$ . Otherwise, returns  $x[i] + y[i]$  for each element of **x** and **y**. The addition operation saturates the result.

The return type is **GenInt1** unless **GenInt1** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

#### hadd

```
template<typename GenInt1, typename GenInt2>
/*return-type*/ hadd(GenInt1 x, GenInt2 y)
```

*Constraints:* Available only if all of the following conditions are met:

- **GenInt1** is a *generic integer type* as defined above;
- If **GenInt1** is not **vec** or the **\_\_swizzle\_\_** type, then **GenInt2** must be the same as **GenInt1**; and
- If **GenInt1** is **vec** or the **\_\_swizzle\_\_** type, then **GenInt2** must also be **vec** or the **\_\_swizzle\_\_** type, and both must have the same element type and the same number of elements.

*Returns:* When the inputs are scalars, returns  $(x + y) \gg 1$ . Otherwise, returns  $(x[i] + y[i]) \gg 1$  for each element of **x** and **y**. The intermediate sum does not modulo overflow.

The return type is **GenInt1** unless **GenInt1** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

#### rhadd

```
template<typename GenInt1, typename GenInt2>
/*return-type*/ rhadd(GenInt1 x, GenInt2 y)
```

*Constraints:* Available only if all of the following conditions are met:

- **GenInt1** is a *generic integer type* as defined above;
- If **GenInt1** is not **vec** or the **\_\_swizzle\_\_** type, then **GenInt2** must be the same as **GenInt1**; and
- If **GenInt1** is **vec** or the **\_\_swizzle\_\_** type, then **GenInt2** must also be **vec** or the **\_\_swizzle\_\_** type, and both must have the same element type and the same number of elements.

*Returns:* When the inputs are scalars, returns  $(x + y + 1) \gg 1$ . Otherwise, returns  $(x[i] + y[i] + 1) \gg 1$  for each element of **x** and **y**. The intermediate sum does not modulo overflow.

The return type is **GenInt1** unless **GenInt1** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**clamp**

```
template<typename GenInt1, typename GenInt2, typename GenInt3>      (1)
/*return-type*/ clamp(GenInt1 x, GenInt2 minval, GenInt3 maxval)

template<typename NonScalar>                                       (2)
/*return-type*/ clamp(NonScalar x, NonScalar::value_type minval,
                      NonScalar::value_type maxval)
```

**Overload (1):**

*Constraints:* Available only if all of the following conditions are met:

- **GenInt1** is a *generic integer type* as defined above;
- If **GenInt1** is not **vec** or the **\_\_swizzle\_\_** type, then **GenInt2** and **GenInt3** must be the same as **GenInt1**; and
- If **GenInt1** is **vec** or the **\_\_swizzle\_\_** type, then **GenInt2** and **GenInt3** must also be **vec** or the **\_\_swizzle\_\_** type, and all three must have the same element type and the same number of elements.

*Preconditions:* If the inputs are scalars, the value of **minval** must be less than or equal to the value of **maxval**. If the inputs are not scalars, each **minval** must be less than or equal to the corresponding **maxval** value.

*Returns:* When the inputs are scalars, returns  $\min(\max(x, \text{minval}), \text{maxval})$ . Otherwise, returns  $\min(\max(x[i], \text{minval}[i]), \text{maxval}[i])$  for each element of **x**, **minval**, and **maxval**.

The return type is **GenInt1** unless **GenInt1** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**Overload (2):**

*Constraints:* Available only if **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type and is a *generic integer type* as defined above.

*Preconditions:* The value of **minval** must be less than or equal to the value of **maxval**.

*Returns:*  $\min(\max(x[i], \text{minval}), \text{maxval})$  for each element of **x**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**clz**

```
template<typename GenInt>
/*return-type*/ clz(GenInt x)
```

*Constraints:* Available only if **GenInt** is a *generic integer type* as defined above.

*Returns:* When the input is a scalar, returns the number of leading 0-bits in **x**, starting at the most significant bit position. Otherwise, returns the number of leading 0-bits in each element of **x**. When a value is 0, the computed count is the size in bits of that value.

The return type is **GenInt** unless **GenInt** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**ctz**

```
template<typename GenInt>
/*return-type*/ ctz(GenInt x)
```

*Constraints:* Available only if **GenInt** is a *generic integer type* as defined above.

*Returns:* When the input is a scalar, returns the number of trailing 0-bits in **x**. Otherwise, returns the number of trailing 0-bits in each element of **x**. When a value is 0, the computed count is the size in bits of that value.

The return type is **GenInt** unless **GenInt** is the `__swizzle__` type, in which case the return type is the corresponding **vec**.

**mad\_hi**

```
template<typename GenInt1, typename GenInt2, typename GenInt3>
/*return-type*/ mad_hi(GenInt1 a, GenInt2 b, GenInt3 c)
```

*Constraints:* Available only if all of the following conditions are met:

- **GenInt1** is a *generic integer type* as defined above;
- If **GenInt1** is not **vec** or the `__swizzle__` type, then **GenInt2** and **GenInt3** must be the same as **GenInt1**; and
- If **GenInt1** is **vec** or the `__swizzle__` type, then **GenInt2** and **GenInt3** must also be **vec** or the `__swizzle__` type, and all three must have the same element type and the same number of elements.

*Returns:* When the inputs are scalars, returns `mul_hi(a, b)+c`. Otherwise, returns `mul_hi(a[i], b[i])+c[i]` for each element of **a**, **b**, and **c**.

The return type is **GenInt1** unless **GenInt1** is the `__swizzle__` type, in which case the return type is the corresponding **vec**.

**mad\_sat**

```
template<typename GenInt1, typename GenInt2, typename GenInt3>
/*return-type*/ mad_sat(GenInt1 a, GenInt2 b, GenInt3 c)
```

*Constraints:* Available only if all of the following conditions are met:

- **GenInt1** is a *generic integer type* as defined above;
- If **GenInt1** is not **vec** or the `__swizzle__` type, then **GenInt2** and **GenInt3** must be the same as **GenInt1**; and
- If **GenInt1** is **vec** or the `__swizzle__` type, then **GenInt2** and **GenInt3** must also be **vec** or the `__swizzle__` type, and all three must have the same element type and the same number of elements.

*Returns:* When the inputs are scalars, returns `a * b + c`. Otherwise, returns `a[i] * b[i] + c[i]` for each element of **a**, **b**, and **c**. The operation saturates the result.

The return type is **GenInt1** unless **GenInt1** is the `__swizzle__` type, in which case the return type is the corresponding **vec**.



**max**

```
template<typename GenInt1, typename GenInt2>                (1)
/*return-type*/ max(GenInt1 x, GenInt2 y)

template<typename NonScalar>                                (2)
/*return-type*/ max(NonScalar x, NonScalar::value_type y)
```

**Overload (1):**

*Constraints:* Available only if all of the following conditions are met:

- **GenInt1** is a *generic integer type* as defined above;
- If **GenInt1** is not **vec** or the **\_\_swizzle\_\_** type, then **GenInt2** must be the same as **GenInt1**; and
- If **GenInt1** is **vec** or the **\_\_swizzle\_\_** type, then **GenInt2** must also be **vec** or the **\_\_swizzle\_\_** type, and both must have the same element type and the same number of elements.

*Returns:* When the inputs are scalars, returns **y** if **x < y** otherwise **x**. When the inputs are not scalars, returns **y[i]** if **x[i] < y[i]** otherwise **x[i]** for each element of **x** and **y**.

The return type is **GenInt1** unless **GenInt1** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**Overload (2):**

*Constraints:* Available only if **NonScalar** is **marray**, **vec**, or the **\_\_swizzle\_\_** type and is a *generic integer type* as defined above.

*Returns:* **y** if **x[i] < y** otherwise **x[i]** for each element of **x**.

The return type is **NonScalar** unless **NonScalar** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**min**

```
template<typename GenInt1, typename GenInt2>                (1)
/*return-type*/ min(GenInt1 x, GenInt2 y)

template<typename NonScalar>                                (2)
/*return-type*/ min(NonScalar x, NonScalar::value_type y)
```

**Overload (1):**

*Constraints:* Available only if all of the following conditions are met:

- **GenInt1** is a *generic integer type* as defined above;
- If **GenInt1** is not **vec** or the **\_\_swizzle\_\_** type, then **GenInt2** must be the same as **GenInt1**; and
- If **GenInt1** is **vec** or the **\_\_swizzle\_\_** type, then **GenInt2** must also be **vec** or the **\_\_swizzle\_\_** type, and both must have the same element type and the same number of elements.

*Returns:* When the inputs are scalars, returns **y** if **y < x** otherwise **x**. When the inputs are not scalars, returns **y[i]** if **y[i] < x[i]** otherwise **x[i]** for each element of **x** and **y**.

The return type is **GenInt1** unless **GenInt1** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**Overload (2):**

**Constraints:** Available only if `NonScalar` is `marray`, `vec`, or the `__swizzle__` type and is a *generic integer type* as defined above.

**Returns:** `y` if `y < x[i]` otherwise `x[i]` for each element of `x`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**mul\_hi**

```
template<typename GenInt1, typename GenInt2>
/*return-type*/ mul_hi(GenInt1 x, GenInt2 y)
```

**Constraints:** Available only if all of the following conditions are met:

- `GenInt1` is a *generic integer type* as defined above;
- If `GenInt1` is not `vec` or the `__swizzle__` type, then `GenInt2` must be the same as `GenInt1`; and
- If `GenInt1` is `vec` or the `__swizzle__` type, then `GenInt2` must also be `vec` or the `__swizzle__` type, and both must have the same element type and the same number of elements.

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

**Returns:** When the inputs are scalars, returns the high half of the product of `x * y`. Otherwise, returns the high half of the product of `x[i] * y[i]` for each element of `x` and `y`.

The return type is `GenInt1` unless `GenInt1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**rotate**

```
template<typename GenInt1, typename GenInt2>
/*return-type*/ rotate(GenInt1 v, GenInt2 count)
```

**Constraints:** Available only if all of the following conditions are met:

- `GenInt1` is a *generic integer type* as defined above;
- If `GenInt1` is not `vec` or the `__swizzle__` type, then `GenInt2` must be the same as `GenInt1`; and
- If `GenInt1` is `vec` or the `__swizzle__` type, then `GenInt2` must also be `vec` or the `__swizzle__` type, and both must have the same element type and the same number of elements.

**Effects:** For each element in `v`, the bits are shifted left by the number of bits given by the corresponding element in `count` (subject to usual shift modulo rules described in the OpenCL 1.2 specification [section 6.3](#)). Bits shifted off the left side of the element are shifted back in from the right.

**Returns:** When the inputs are scalars, the result of rotating `v` by `count` as described above. Otherwise, the result of rotating `v[i]` by `count[i]` for each element of `v` and `count`.

The return type is `GenInt1` unless `GenInt1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**sub\_sat**

```
template<typename GenInt1, typename GenInt2>
/*return-type*/ sub_sat(GenInt1 x, GenInt2 y)
```

*Constraints:* Available only if all of the following conditions are met:

- **GenInt1** is a *generic integer type* as defined above;
- If **GenInt1** is not **vec** or the **\_\_swizzle\_\_** type, then **GenInt2** must be the same as **GenInt1**; and
- If **GenInt1** is **vec** or the **\_\_swizzle\_\_** type, then **GenInt2** must also be **vec** or the **\_\_swizzle\_\_** type, and both must have the same element type and the same number of elements.

*Returns:* When the inputs are scalars, returns  $x - y$ . Otherwise, returns  $x[i] - y[i]$  for each element of  $x$  and  $y$ . The subtraction operation saturates the result.

The return type is **GenInt1** unless **GenInt1** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

**upsample**

```
template<typename UInt8Bit1, typename UInt8Bit2>
/*return-type*/ upsample(UInt8Bit1 hi, UInt8Bit2 lo)
```

*Constraints:* Available only if one of the following conditions is met:

- **UInt8Bit1** and **UInt8Bit2** are both **uint8\_t**;
- **UInt8Bit1** and **UInt8Bit2** are both **marray** with element type **uint8\_t** and the same number of elements; or
- **UInt8Bit1** and **UInt8Bit2** are any combination of **vec** or the **\_\_swizzle\_\_** type with element type **uint8\_t** and the same number of elements.

*Returns:* When the inputs are scalars, returns  $((\text{uint16\_t})hi \ll 8) \mid lo$ . Otherwise, returns  $((\text{uint16\_t})hi[i] \ll 8) \mid lo[i]$  for each element of  $hi$  and  $lo$ .

The return type is **uint16\_t** when the inputs are scalar. When the inputs are **marray**, the return type is **marray** with element type **uint16\_t** and the same number of elements as the inputs. Otherwise, the return type is **vec** with element type **uint16\_t** and the same number of elements as the inputs.

**upsample**

```
template<typename Int8Bit, typename UInt8Bit>
/*return-type*/ upsample(Int8Bit hi, UInt8Bit lo)
```

*Constraints:* Available only if one of the following conditions is met:

- **Int8Bit** is **int8\_t** and **UInt8Bit** is **uint8\_t**;
- **Int8Bit** is **marray** with element type **int8\_t** and **UInt8Bit** is **marray** with element type **uint8\_t** and both have the same number of elements; or
- **Int8Bit** is **vec** or the **\_\_swizzle\_\_** type with element type **int8\_t** and **UInt8Bit** is **vec** or the **\_\_swizzle\_\_** type with element type **uint8\_t** and both have the same number of elements.

*Returns:* When the inputs are scalars, returns  $((\text{int16\_t})hi \ll 8) \mid lo$ . Otherwise, returns

`((int16_t)hi[i] << 8) | lo[i]` for each element of `hi` and `lo`.

The return type is `int16_t` when the inputs are scalar. When the inputs are `marray`, the return type is `marray` with element type `int16_t` and the same number of elements as the inputs. Otherwise, the return type is `vec` with element type `int16_t` and the same number of elements as the inputs.

#### upsample

```
template<typename UInt16Bit1, typename UInt16Bit2>
/*return-type*/ upsample(UInt16Bit1 hi, UInt16Bit2 lo)
```

*Constraints:* Available only if one of the following conditions is met:

- `UInt16Bit1` and `UInt16Bit2` are both `uint16_t`;
- `UInt16Bit1` and `UInt16Bit2` are both `marray` with element type `uint16_t` and the same number of elements; or
- `UInt16Bit1` and `UInt16Bit2` are any combination of `vec` or the `__swizzle__` type with element type `uint16_t` and the same number of elements.

*Returns:* When the inputs are scalars, returns `((uint32_t)hi << 16) | lo`. Otherwise, returns `((uint32_t)hi[i] << 16) | lo[i]` for each element of `hi` and `lo`.

The return type is `uint32_t` when the inputs are scalar. When the inputs are `marray`, the return type is `marray` with element type `uint32_t` and the same number of elements as the inputs. Otherwise, the return type is `vec` with element type `uint32_t` and the same number of elements as the inputs.

#### upsample

```
template<typename Int16Bit, typename UInt16Bit>
/*return-type*/ upsample(Int16Bit hi, UInt16Bit lo)
```

*Constraints:* Available only if one of the following conditions is met:

- `Int16Bit` is `int16_t` and `UInt16Bit` is `uint16_t`;
- `Int16Bit` is `marray` with element type `int16_t` and `UInt16Bit` is `marray` with element type `uint16_t` and both have the same number of elements; or
- `Int16Bit` is `vec` or the `__swizzle__` type with element type `int16_t` and `UInt16Bit` is `vec` or the `__swizzle__` type with element type `uint16_t` and both have the same number of elements.

*Returns:* When the inputs are scalars, returns `((int32_t)hi << 16) | lo`. Otherwise, returns `((int32_t)hi[i] << 16) | lo[i]` for each element of `hi` and `lo`.

The return type is `int32_t` when the inputs are scalar. When the inputs are `marray`, the return type is `marray` with element type `int32_t` and the same number of elements as the inputs. Otherwise, the return type is `vec` with element type `int32_t` and the same number of elements as the inputs.

#### upsample

```
template<typename UInt32Bit1, typename UInt32Bit2>
/*return-type*/ upsample(UInt32Bit1 hi, UInt32Bit2 lo)
```

*Constraints:* Available only if one of the following conditions is met:

- `UInt32Bit1` and `UInt32Bit2` are both `uint32_t`;
- `UInt32Bit1` and `UInt32Bit2` are both `marray` with element type `uint32_t` and the same number of elements; or
- `UInt32Bit1` and `UInt32Bit2` are any combination of `vec` or the `__swizzle__` type with element type `uint32_t` and the same number of elements.

*Returns:* When the inputs are scalars, returns `((uint64_t)hi << 32) | lo`. Otherwise, returns `((uint64_t)hi[i] << 32) | lo[i]` for each element of `hi` and `lo`.

The return type is `uint64_t` when the inputs are scalar. When the inputs are `marray`, the return type is `marray` with element type `uint64_t` and the same number of elements as the inputs. Otherwise, the return type is `vec` with element type `uint64_t` and the same number of elements as the inputs.

#### upsample

```
template<typename Int32Bit, typename UInt32Bit>
/*return-type*/ upsample(Int32Bit hi, UInt32Bit lo)
```

*Constraints:* Available only if one of the following conditions is met:

- `Int32Bit` is `int32_t` and `UInt32Bit` is `uint32_t`;
- `Int32Bit` is `marray` with element type `int32_t` and `UInt32Bit` is `marray` with element type `uint32_t` and both have the same number of elements; or
- `Int32Bit` is `vec` or the `__swizzle__` type with element type `int32_t` and `UInt32Bit` is `vec` or the `__swizzle__` type with element type `uint32_t` and both have the same number of elements.

*Returns:* When the inputs are scalars, returns `((int64_t)hi << 32) | lo`. Otherwise, returns `((int64_t)hi[i] << 32) | lo[i]` for each element of `hi` and `lo`.

The return type is `int64_t` when the inputs are scalar. When the inputs are `marray`, the return type is `marray` with element type `int64_t` and the same number of elements as the inputs. Otherwise, the return type is `vec` with element type `int64_t` and the same number of elements as the inputs.

#### popcount

```
template<typename GenInt>
/*return-type*/ popcount(GenInt x)
```

*Constraints:* Available only if `GenInt` is a *generic integer type* as defined above.

*Returns:* When the input is a scalar, returns the number of non-zero bits in `x`. Otherwise, returns the number of non-zero bits in `x[i]` for each element of `x`.

The return type is `GenInt` unless `GenInt` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### mad24

```
template<typename Int32Bit1, typename Int32Bit2, typename Int32Bit3>
```

```
/*return-type*/ mad24(Int32Bit1 x, Int32Bit2 y, Int32Bit3 z)
```

*Constraints:* Available only if all of the following conditions are met:

- **Int32Bit1** is one of the following types:
  - `int32_t`
  - `uint32_t`
  - `marray<int32_t, N>`
  - `marray<uint32_t, N>`
  - `vec<int32_t, N>`
  - `vec<uint32_t, N>`
  - `__swizzle__` that is convertible to `vec<int32_t, N>`
  - `__swizzle__` that is convertible to `vec<uint32_t, N>`
- If **Int32Bit1** is not `vec` or the `__swizzle__` type, then **Int32Bit2** and **Int32Bit3** must be the same as **Int32Bit1**; and
- If **Int32Bit1** is `vec` or the `__swizzle__` type, then **Int32Bit2** and **Int32Bit3** must also be `vec` or the `__swizzle__` type, and all three must have the same element type and the same number of elements.

*Preconditions:* If the inputs are signed scalars, the values of **x** and **y** must be in the range  $[-2^{23}, 2^{23}-1]$ . If the inputs are unsigned scalars, the values of **x** and **y** must be in the range  $[0, 2^{24}-1]$ . If the inputs are not scalars, each element of **x** and **y** must be in these ranges.

*Returns:* When the inputs are scalars, returns  $x * y + z$ . Otherwise, returns  $x[i] * y[i] + z[i]$  for each element of **x**, **y**, and **z**.

The return type is **Int32Bit1** unless **Int32Bit1** is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

## mul24

```
template<typename Int32Bit1, typename Int32Bit2>
/*return-type*/ mul24(Int32Bit1 x, Int32Bit2 y)
```

*Constraints:* Available only if all of the following conditions are met:

- **Int32Bit1** is one of the following types:
  - `int32_t`
  - `uint32_t`
  - `marray<int32_t, N>`
  - `marray<uint32_t, N>`
  - `vec<int32_t, N>`
  - `vec<uint32_t, N>`
  - `__swizzle__` that is convertible to `vec<int32_t, N>`
  - `__swizzle__` that is convertible to `vec<uint32_t, N>`
- If **Int32Bit1** is not `vec` or the `__swizzle__` type, then **Int32Bit2** must be the same as **Int32Bit1**; and
- If **Int32Bit1** is `vec` or the `__swizzle__` type, then **Int32Bit2** must also be `vec` or the `__swizzle__` type, and both must have the same element type and the same number of elements.

**Preconditions:** If the inputs are signed scalars, the values of `x` and `y` must be in the range  $[-2^{23}, 2^{23}-1]$ . If the inputs are unsigned scalars, the values of `x` and `y` must be in the range  $[0, 2^{24}-1]$ . If the inputs are not scalars, each element of `x` and `y` must be in these ranges.

**Returns:** When the inputs are scalars, returns `x * y`. Otherwise, returns `x[i] * y[i]` for each element of `x` and `y`.

The return type is `Int32Bit1` unless `Int32Bit1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

## 4.17.8. Common functions

This section describes the common functions that are available in the `sycl` namespace in both host and device code.

The descriptions in this section use the type name `__swizzle__` to refer to the classes defined in [Section 4.14.2.4](#). This type can be any instantiation of the class templates named `__writeable_swizzle__` or `__const_swizzle__` in that section, so long as the instantiation satisfies the constraints listed in the function's description.

The function descriptions in this section also use the term *generic floating point type* to represent the following types:

- `float`
- `double`
- `half`
- `marray<float, N>`
- `marray<double, N>`
- `marray<half, N>`
- `vec<float, N>`
- `vec<double, N>`
- `vec<half, N>`
- `__swizzle__` that is convertible to `vec<float, N>`
- `__swizzle__` that is convertible to `vec<double, N>`
- `__swizzle__` that is convertible to `vec<half, N>`

### `clamp`

```
template<typename GenFloat1, typename GenFloat2, typename GenFloat3>      (1)
/*return-type*/ clamp(GenFloat1 x, GenFloat2 minval, GenFloat3 maxval)

template<typename NonScalar>                                              (2)
/*return-type*/ clamp(NonScalar x, NonScalar::value_type minval,
                      NonScalar::value_type maxval)
```

#### Overload (1):

**Constraints:** Available only if all of the following conditions are met:

- `GenFloat1` is a *generic floating point type* as defined above;

- If `GenFloat1` is not `vec` or the `__swizzle__` type, then `GenFloat2` and `GenFloat3` must be the same as `GenFloat1`; and
- If `GenFloat1` is `vec` or the `__swizzle__` type, then `GenFloat2` and `GenFloat3` must also be `vec` or the `__swizzle__` type, and all three must have the same element type and the same number of elements.

**Preconditions:** If the inputs are scalars, the value of `minval` must be less than or equal to the value of `maxval`. If the inputs are not scalars, each element of `minval` must be less than or equal to the corresponding element of `maxval`.

**Returns:** When the inputs are scalars, returns `fmin(fmax(x, minval), maxval)`. Otherwise, returns `fmin(fmax(x[i], minval[i]), maxval[i])` for each element of `x`, `minval`, and `maxval`.

The return type is `GenFloat1` unless `GenFloat1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### Overload (2):

**Constraints:** Available only if `NonScalar` is `marray`, `vec`, or the `__swizzle__` type and is a *generic floating point type* as defined above.

**Preconditions:** The value of `minval` must be less than or equal to the value of `maxval`.

**Returns:** `fmin(fmax(x[i], minval), maxval)` for each element of `x`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### degrees

```
template<typename GenFloat>
/*return-type*/ degrees(GenFloat radians)
```

**Constraints:** Available only if `GenFloat` is a *generic floating point type* as defined above.

**Effects:** Converts radians to degrees.

**Returns:** When the inputs are scalars, returns  $(180 / \pi) * \text{radians}$ . Otherwise, returns  $(180 / \pi) * \text{radians}[i]$  for each element of `radians`.

The return type is `GenFloat` unless `GenFloat` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### max

```
template<typename GenFloat1, typename GenFloat2> (1)
/*return-type*/ max(GenFloat1 x, GenFloat2 y)

template<typename NonScalar> (2)
/*return-type*/ max(NonScalar x, NonScalar::value_type y)
```

#### Overload (1):

**Constraints:** Available only if all of the following conditions are met:

- `GenFloat1` is a *generic floating point type* as defined above;



- If `GenFloat1` is not `vec` or the `__swizzle__` type, then `GenFloat2` must be the same as `GenFloat1`; and
- If `GenFloat1` is `vec` or the `__swizzle__` type, then `GenFloat2` must also be `vec` or the `__swizzle__` type, and both must have the same element type and the same number of elements.

**Preconditions:** When the inputs are scalars, `x` and `y` must not be infinite or NaN. When the inputs are not scalars, no element of `x` or `y` may be infinite or NaN.

**Returns:** When the inputs are scalars, returns `y` if `x < y` otherwise `x`. When the inputs are not scalars, returns `y[i]` if `x[i] < y[i]` otherwise `x[i]` for each element of `x` and `y`.

The return type is `GenFloat1` unless `GenFloat1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### Overload (2):

**Constraints:** Available only if `NonScalar` is `marray`, `vec`, or the `__swizzle__` type and is a *generic floating point type* as defined above.

**Preconditions:** No element of `x` may be infinite or NaN. The value of `y` must not be infinite or NaN.

**Returns:** `y` if `x[i] < y` otherwise `x[i]` for each element of `x`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

**min**

```
template<typename GenFloat1, typename GenFloat2>           (1)
/*return-type*/ min(GenFloat1 x, GenFloat2 y)

template<typename NonScalar>                               (2)
/*return-type*/ min(NonScalar x, NonScalar::value_type y)
```

#### Overload (1):

**Constraints:** Available only if all of the following conditions are met:

- `GenFloat1` is a *generic floating point type* as defined above;
- If `GenFloat1` is not `vec` or the `__swizzle__` type, then `GenFloat2` must be the same as `GenFloat1`; and
- If `GenFloat1` is `vec` or the `__swizzle__` type, then `GenFloat2` must also be `vec` or the `__swizzle__` type, and both must have the same element type and the same number of elements.

**Preconditions:** When the inputs are scalars, `x` and `y` must not be infinite or NaN. When the inputs are not scalars, no element of `x` or `y` may be infinite or NaN.

**Returns:** When the inputs are scalars, returns `y` if `y < x` otherwise `x`. When the inputs are not scalars, returns `y[i]` if `y[i] < x[i]` otherwise `x[i]` for each element of `x` and `y`.

The return type is `GenFloat1` unless `GenFloat1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### Overload (2):

**Constraints:** Available only if `NonScalar` is `marray`, `vec`, or the `__swizzle__` type and is a *generic floating point type* as defined above.

**Preconditions:** No element of `x` may be infinite or NaN. The value of `y` must not be infinite or NaN.

Returns:  $y$  if  $y < x[i]$  otherwise  $x[i]$  for each element of  $x$ .

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### `mix`

```
template<typename GenFloat1, typename GenFloat2, typename GenFloat3>      (1)
/*return-type*/ mix(GenFloat1 x, GenFloat2 y, GenFloat3 a)

template<typename NonScalar1, typename NonScalar2>                      (2)
/*return-type*/ mix(NonScalar1 x, NonScalar2 y, NonScalar1::value_type a)
```

#### Overload (1):

*Constraints:* Available only if all of the following conditions are met:

- `GenFloat1` is a *generic floating point type* as defined above;
- If `GenFloat1` is not `vec` or the `__swizzle__` type, then `GenFloat2` and `GenFloat3` must be the same as `GenFloat1`; and
- If `GenFloat1` is `vec` or the `__swizzle__` type, then `GenFloat2` and `GenFloat3` must also be `vec` or the `__swizzle__` type, and all three must have the same element type and the same number of elements.

*Preconditions:* If the inputs are scalars, the value of  $a$  must be in the range  $[0.0, 1.0]$ . If the inputs are not scalars, each element of  $a$  must be in the range  $[0.0, 1.0]$ .

*Returns:* The linear blend of  $x$  and  $y$ . When the inputs are scalars, returns  $x + (y - x) * a$ . Otherwise, returns  $x[i] + (y[i] - x[i]) * a[i]$  for each element of  $x$ ,  $y$ , and  $a$ .

The return type is `GenFloat1` unless `GenFloat1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### Overload (2):

*Constraints:* Available only if `NonScalar` is `marray`, `vec`, or the `__swizzle__` type and is a *generic floating point type* as defined above.

*Preconditions:* The value of  $a$  must be in the range  $[0.0, 1.0]$ .

*Returns:* The linear blend of  $x$  and  $y$ , computed as  $x[i] + (y[i] - x[i]) * a$  for each element of  $x$  and  $y$ .

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### `radians`

```
template<typename GenFloat>
/*return-type*/ radians(GenFloat degrees)
```

*Constraints:* Available only if `GenFloat` is a *generic floating point type* as defined above.

*Effects:* Converts degrees to radians.

*Returns:* When the inputs are scalars, returns  $(\pi / 180) * degrees$ . Otherwise, returns  $(\pi / 180) * degrees[i]$  for each element of  $degrees$ .

The return type is `GenFloat` unless `GenFloat` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

### step

```
template<typename GenFloat1, typename GenFloat2> (1)
/*return-type*/ step(GenFloat1 edge, GenFloat2 x)

template<typename NonScalar> (2)
/*return-type*/ step(NonScalar::value_type edge, NonScalar x)
```

#### Overload (1):

*Constraints:* Available only if all of the following conditions are met:

- `GenFloat1` is a *generic floating point type* as defined above;
- If `GenFloat1` is not `vec` or the `__swizzle__` type, then `GenFloat2` must be the same as `GenFloat1`; and
- If `GenFloat1` is `vec` or the `__swizzle__` type, then `GenFloat2` must also be `vec` or the `__swizzle__` type, and both must have the same element type and the same number of elements.

*Returns:* When the inputs are scalars, returns the value  $(x < \text{edge}) ? 0.0 : 1.0$ . When the inputs are not scalars, returns the value  $(x[i] < \text{edge}[i]) ? 0.0 : 1.0$  for each element of `x` and `edge`.

The return type is `GenFloat1` unless `GenFloat1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### Overload (2):

*Constraints:* Available only if `NonScalar` is `marray`, `vec`, or the `__swizzle__` type and is a *generic floating point type* as defined above.

*Returns:* The value  $(x[i] < \text{edge}) ? 0.0 : 1.0$  for each element of `x`.

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

### smoothstep

```
template<typename GenFloat1, typename GenFloat2, typename GenFloat3> (1)
/*return-type*/ smoothstep(GenFloat1 edge0, GenFloat2 edge1, GenFloat3 x)

template<typename NonScalar> (2)
/*return-type*/ smoothstep(NonScalar::value_type edge0, NonScalar::value_type edge1,
                          NonScalar x)
```

#### Overload (1):

*Constraints:* Available only if all of the following conditions are met:

- `GenFloat1` is a *generic floating point type* as defined above;
- If `GenFloat1` is not `vec` or the `__swizzle__` type, then `GenFloat2` and `GenFloat3` must be the same as `GenFloat1`; and
- If `GenFloat1` is `vec` or the `__swizzle__` type, then `GenFloat2` and `GenFloat3` must also be `vec` or the

`__swizzle__` type, and all three must have the same element type and the same number of elements.

**Preconditions:** If the inputs are scalar, `edge0` must be less than `edge1` and none of `edge0`, `edge1`, or `x` may be NaN. If the inputs are not scalar, each element of `edge0` must be less than the corresponding element of `edge1` and no element of `edge0`, `edge1`, or `x` may be NaN.

**Returns:** When the inputs are scalars, returns 0.0 if `x <= edge0` and 1.0 if `x >= edge1` and performs smooth Hermite interpolation between 0 and 1 when `edge0 < x < edge1`. This is useful in cases where you would want a threshold function with a smooth transition. This is equivalent to:

```
GenFloat1 t;
t = clamp((x - edge0) / (edge1 - edge0), 0, 1);
return t * t * (3 - 2 * t);
```

When the inputs are not scalars, returns the following value for each element of `edge0`, `edge1`, and `x`:

```
GenFloat1::value_type t;
t = clamp((x[i] - edge0[i]) / (edge1[i] - edge0[i]), 0, 1);
return t * t * (3 - 2 * t);
```

The return type is `GenFloat1` unless `GenFloat1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

### Overload (2):

**Constraints:** Available only if `NonScalar` is `marray`, `vec`, or the `__swizzle__` type and is a *generic floating point type* as defined above.

**Preconditions:** The value of `edge0` must be less than `edge1` and neither `edge0` nor `edge1` may be NaN. No element of `x` may be NaN.

**Returns:** The following value for each element of `x`:

```
NonScalar::value_type t;
t = clamp((x[i] - edge0) / (edge1 - edge0), 0, 1);
return t * t * (3 - 2 * t);
```

The return type is `NonScalar` unless `NonScalar` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

### sign

```
template<typename GenFloat>
/*return-type*/ sign(GenFloat x)
```

**Constraints:** Available only if `GenFloat` is a *generic floating point type* as defined above.

**Returns:** When the input is scalar, returns 1.0 if `x > 0`, -0.0 if `x == -0.0`, +0.0 if `x == +0.0`, -1.0 if `x < 0`, or 0.0 if `x` is a NaN. When the input is not scalar, returns these values for each element of `x`.

The return type is `GenFloat` unless `GenFloat` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

## 4.17.9. Geometric functions

This section describes the geometric functions that are available in the `sycl` namespace in both host and device code.

The descriptions in this section use the type name `__swizzle__` to refer to the classes defined in [Section 4.14.2.4](#). This type can be any instantiation of the class templates named `__writeable_swizzle__` or `__const_swizzle__` in that section, so long as the instantiation satisfies the constraints listed in the function's description.

The function descriptions in this section also use two terms that refer to a specific list of types. The term *generic geometric type* represents the following types:

- `float`
- `double`
- `half`
- `marray<float, N>`, where `N` is 2, 3, or 4
- `marray<double, N>`, where `N` is 2, 3, or 4
- `marray<half, N>`, where `N` is 2, 3, or 4
- `vec<float, N>`, where `N` is 2, 3, or 4
- `vec<double, N>`, where `N` is 2, 3, or 4
- `vec<half, N>`, where `N` is 2, 3, or 4
- `__swizzle__` that is convertible to `vec<float, N>`, where `N` is 2, 3, or 4
- `__swizzle__` that is convertible to `vec<double, N>`, where `N` is 2, 3, or 4
- `__swizzle__` that is convertible to `vec<half, N>`, where `N` is 2, 3, or 4

The term *float geometric type* represents these types:

- `float`
- `marray<float, N>`, where `N` is 2, 3, or 4
- `vec<float, N>`, where `N` is 2, 3, or 4
- `__swizzle__` that is convertible to `vec<float, N>`, where `N` is 2, 3, or 4

### cross

```
template<typename Geo3or4Float1, typename Geo3or4Float2>
/*return-type*/ cross(Geo3or4Float1 p0, Geo3or4Float2 p1)
```

**Constraints:** Available only if all of the following conditions are met:

- `Geo3or4Float1` is one of the following types:
  - `marray<float, 3>`
  - `marray<double, 3>`
  - `marray<half, 3>`
  - `marray<float, 4>`
  - `marray<double, 4>`
  - `marray<half, 4>`
  - `vec<float, 3>`

- `vec<double, 3>`
- `vec<half, 3>`
- `vec<float, 4>`
- `vec<double, 4>`
- `vec<half, 4>`
- `__swizzle__` that is convertible to `vec<float, 3>`
- `__swizzle__` that is convertible to `vec<double, 3>`
- `__swizzle__` that is convertible to `vec<half, 3>`
- `__swizzle__` that is convertible to `vec<float, 4>`
- `__swizzle__` that is convertible to `vec<double, 4>`
- `__swizzle__` that is convertible to `vec<half, 4>`
- If `Geo3or4Float1` is `marray`, then `Geo3or4Float2` must be the same as `Geo3or4Float1`; and
- If `Geo3or4Float1` is `vec` or the `__swizzle__` type, then `Geo3or4Float2` must also be `vec` or the `__swizzle__` type, and both must have the same element type and the same number of elements.

**Returns:** The cross product of first 3 components of `p0` and `p1`. When the inputs have 4 components, the 4th component of the result is 0.0.

The return type is `Geo3or4Float1` unless `Geo3or4Float1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

#### dot

```
template<typename GeoFloat1, typename GeoFloat2>
/*return-type*/ dot(GeoFloat1 p0, GeoFloat2 p1)
```

**Constraints:** Available only if all of the following conditions are met:

- `GeoFloat1` is a *generic geometric type* as defined above;
- If `GeoFloat1` is not `vec` or the `__swizzle__` type, then `GeoFloat2` must be the same as `GeoFloat1`; and
- If `GeoFloat1` is `vec` or the `__swizzle__` type, then `GeoFloat2` must also be `vec` or the `__swizzle__` type, and both must have the same element type and the same number of elements.

**Returns:** The dot product of `p0` and `p1`.

The return type is `GeoFloat1` if the input types are scalar. Otherwise, the return type is `GeoFloat1::value_type`.

#### distance

```
template<typename GeoFloat1, typename GeoFloat2>
/*return-type*/ distance(GeoFloat1 p0, GeoFloat2 p1)
```

**Constraints:** Available only if all of the following conditions are met:

- `GeoFloat1` is a *generic geometric type* as defined above;
- If `GeoFloat1` is not `vec` or the `__swizzle__` type, then `GeoFloat2` must be the same as `GeoFloat1`; and
- If `GeoFloat1` is `vec` or the `__swizzle__` type, then `GeoFloat2` must also be `vec` or the `__swizzle__` type, and

both must have the same element type and the same number of elements.

*Returns:* The distance between `p0` and `p1`. This is calculated as `length(p0 - p1)`.

The return type is `GeoFloat1` if the input types are scalar. Otherwise, the return type is `GeoFloat1::value_type`.

### length

```
template<typename GeoFloat>
/*return-type*/ length(GeoFloat p)
```

*Constraints:* Available only if `GeoFloat` is a *generic geometric type* as defined above.

*Returns:* The length of vector `p`, i.e., `sqrt(pow(p[0],2) + pow(p[1],2) + ...)`.

The return type is `GeoFloat` if the input type is scalar. Otherwise, the return type is `GeoFloat::value_type`.

### normalize

```
template<typename GeoFloat>
/*return-type*/ normalize(GeoFloat p)
```

*Constraints:* Available only if `GeoFloat` is a *generic geometric type* as defined above.

*Returns:* A vector in the same direction as `p` but with a length of 1.

The return type is `GeoFloat` unless `GeoFloat` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

### fast\_distance

```
template<typename GeoFloat1, typename GeoFloat2>
/*return-type*/ fast_distance(GeoFloat1 p0, GeoFloat2 p1)
```

*Constraints:* Available only if all of the following conditions are met:

- `GeoFloat1` is a *float geometric type* as defined above;
- If `GeoFloat1` is not `vec` or the `__swizzle__` type, then `GeoFloat2` must be the same as `GeoFloat1`; and
- If `GeoFloat1` is `vec` or the `__swizzle__` type, then `GeoFloat2` must also be `vec` or the `__swizzle__` type, and both must have the same number of elements.

*Returns:* The value `fast_length(p0 - p1)`.

The return type is `GeoFloat1` if the input type is scalar. Otherwise, the return type is `GeoFloat1::value_type`.

### fast\_length

```
template<typename GeoFloat>
```

```
/*return-type*/ fast_length(GeoFloat p)
```

*Constraints:* Available only if `GeoFloat` is a *float geometric type* as defined above.

*Returns:* The length of vector `p` computed as: `half_precision::sqrt(pow(p[0],2) + pow(p[1],2) + ...)`.

The return type is `GeoFloat` if the input type is scalar. Otherwise, the return type is `GeoFloat::value_type`.

#### `fast_normalize`

```
template<typename GeoFloat>
/*return-type*/ fast_normalize(GeoFloat p)
```

*Constraints:* Available only if `GeoFloat` is a *float geometric type* as defined above.

*Returns:* A vector in the same direction as `p` but with a length of 1 computed as `p * half_precision::rsqrt(pow(p[0],2) + pow(p[1],2) + ...)`.

The result shall be within 8192 ulps error from the infinitely precise result of

```
if (all(p == 0.0f))
    result = p;
else
    result = p / sqrt(pow(p[0], 2) + pow(p[1], 2) + ...);
```

with the following exceptions:

1. If the sum of squares is greater than `FLT_MAX` then the value of the floating-point values in the result vector are undefined.
2. If the sum of squares is less than `FLT_MIN` then the implementation may return back `p`.

The return type is `GeoFloat` unless `GeoFloat` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

## 4.17.10. Relational functions

This section describes the relational functions that are available in the `sycl` namespace in both host and device code. These functions perform various relational comparisons on `vec`, `marray`, and scalar types.

The comparisons performed by `isequal`, `isgreater`, `isgreaterequal`, `isless`, `islessequal`, and `islessgreater` are false when one or both operands are NaN. The comparison performed by `isnotequal` is true when one or both operands are NaN.

The descriptions in this section use the type name `__swizzle__` to refer to the classes defined in [Section 4.14.2.4](#). This type can be any instantiation of the class templates named `__writeable_swizzle__` or `__const_swizzle__` in that section, so long as the instantiation satisfies the constraints listed in the function's description.

The function descriptions in this section also use two terms that refer to a specific list of types. The term *generic scalar type* represents the following types:

- `char`
- `signed char`



- `short`
- `int`
- `long`
- `long long`
- `unsigned char`
- `unsigned short`
- `unsigned int`
- `unsigned long`
- `unsigned long long`
- `float`
- `double`
- `half`

The term *vector element type* represents these types:

- `int8_t`
- `int16_t`
- `int32_t`
- `int64_t`
- `uint8_t`
- `uint16_t`
- `uint32_t`
- `uint64_t`
- `float`
- `double`
- `half`

## isequal

```
bool isequal(float x, float y)           (1)
bool isequal(double x, double y)        (2)
bool isequal(half x, half y)            (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ isequal(NonScalar1 x, NonScalar2 y)
```

### Overloads (1) - (3):

*Returns:* The value `(x == y)`.

### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for `NonScalar1` and `NonScalar2`:
  - Both `NonScalar1` and `NonScalar2` are `marray`; or
  - `NonScalar1` and `NonScalar2` are any combination of `vec` and the `__swizzle__` type;

- `NonScalar1` and `NonScalar2` have the same number of elements and the same element type; and
- The element type is `float`, `double`, or `half`.

*Returns:* If `NonScalar1` is `marray`, the value `(x[i] == y[i])` for each element of `x` and `y`. If `NonScalar1` is `vec` or the `__swizzle__` type, returns the value `((x[i] == y[i]) ? -1 : 0)` for each element of `x` and `y`.

The return type depends on `NonScalar1`:

NonScalar1	Return Type
<code>marray&lt;float, N&gt;</code> <code>marray&lt;double, N&gt;</code> <code>marray&lt;half, N&gt;</code>	<code>marray&lt;bool, N&gt;</code>
<code>vec&lt;float, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;float, N&gt;</code>	<code>vec&lt;int32_t, N&gt;</code>
<code>vec&lt;double, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;double, N&gt;</code>	<code>vec&lt;int64_t, N&gt;</code>
<code>vec&lt;half, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;half, N&gt;</code>	<code>vec&lt;int16_t, N&gt;</code>

### isnotequal

```

bool isnotequal(float x, float y)           (1)
bool isnotequal(double x, double y)         (2)
bool isnotequal(half x, half y)             (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ isnotequal(NonScalar1 x, NonScalar2 y)

```

### Overloads (1) - (3):

*Returns:* The value `(x != y)`.

### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for `NonScalar1` and `NonScalar2`:
  - Both `NonScalar1` and `NonScalar2` are `marray`; or
  - `NonScalar1` and `NonScalar2` are any combination of `vec` and the `__swizzle__` type;
- `NonScalar1` and `NonScalar2` have the same number of elements and the same element type; and
- The element type is `float`, `double`, or `half`.

*Returns:* If `NonScalar1` is `marray`, the value `(x[i] != y[i])` for each element of `x` and `y`. If `NonScalar1` is `vec` or the `__swizzle__` type, returns the value `((x[i] != y[i]) ? -1 : 0)` for each element of `x` and `y`.

The return type depends on `NonScalar1`:

NonScalar1	Return Type
<code>marray&lt;float, N&gt;</code> <code>marray&lt;double, N&gt;</code> <code>marray&lt;half, N&gt;</code>	<code>marray&lt;bool, N&gt;</code>

NonScalar1	Return Type
<code>vec&lt;float, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;float, N&gt;</code>	<code>vec&lt;int32_t, N&gt;</code>
<code>vec&lt;double, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;double, N&gt;</code>	<code>vec&lt;int64_t, N&gt;</code>
<code>vec&lt;half, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;half, N&gt;</code>	<code>vec&lt;int16_t, N&gt;</code>

**isgreater**

```

bool isgreater(float x, float y)           (1)
bool isgreater(double x, double y)        (2)
bool isgreater(half x, half y)            (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ isgreater(NonScalar1 x, NonScalar2 y)

```

**Overloads (1) - (3):**

*Returns:* The value `(x > y)`.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for `NonScalar1` and `NonScalar2`:
  - Both `NonScalar1` and `NonScalar2` are `marray`; or
  - `NonScalar1` and `NonScalar2` are any combination of `vec` and the `__swizzle__` type;
- `NonScalar1` and `NonScalar2` have the same number of elements and the same element type; and
- The element type is `float`, `double`, or `half`.

*Returns:* If `NonScalar1` is `marray`, the value `(x[i] > y[i])` for each element of `x` and `y`. If `NonScalar1` is `vec` or the `__swizzle__` type, returns the value `((x[i] > y[i]) ? -1 : 0)` for each element of `x` and `y`.

The return type depends on `NonScalar1`:

NonScalar1	Return Type
<code>marray&lt;float, N&gt;</code> <code>marray&lt;double, N&gt;</code> <code>marray&lt;half, N&gt;</code>	<code>marray&lt;bool, N&gt;</code>
<code>vec&lt;float, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;float, N&gt;</code>	<code>vec&lt;int32_t, N&gt;</code>
<code>vec&lt;double, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;double, N&gt;</code>	<code>vec&lt;int64_t, N&gt;</code>
<code>vec&lt;half, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;half, N&gt;</code>	<code>vec&lt;int16_t, N&gt;</code>

**isgreaterqual**

```

bool isgreaterqual(float x, float y)           (1)
bool isgreaterqual(double x, double y)        (2)
bool isgreaterqual(half x, half y)            (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ isgreaterqual(NonScalar1 x, NonScalar2 y)

```

**Overloads (1) - (3):**

*Returns:* The value  $(x \geq y)$ .

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for `NonScalar1` and `NonScalar2`:
  - Both `NonScalar1` and `NonScalar2` are `marray`; or
  - `NonScalar1` and `NonScalar2` are any combination of `vec` and the `__swizzle__` type;
- `NonScalar1` and `NonScalar2` have the same number of elements and the same element type; and
- The element type is `float`, `double`, or `half`.

*Returns:* If `NonScalar1` is `marray`, the value  $(x[i] \geq y[i])$  for each element of `x` and `y`. If `NonScalar1` is `vec` or the `__swizzle__` type, returns the value  $((x[i] \geq y[i]) ? -1 : 0)$  for each element of `x` and `y`.

The return type depends on `NonScalar1`:

NonScalar1	Return Type
<code>marray&lt;float, N&gt;</code> <code>marray&lt;double, N&gt;</code> <code>marray&lt;half, N&gt;</code>	<code>marray&lt;bool, N&gt;</code>
<code>vec&lt;float, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;float, N&gt;</code>	<code>vec&lt;int32_t, N&gt;</code>
<code>vec&lt;double, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;double, N&gt;</code>	<code>vec&lt;int64_t, N&gt;</code>
<code>vec&lt;half, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;half, N&gt;</code>	<code>vec&lt;int16_t, N&gt;</code>

**isless**

```

bool isless(float x, float y)                 (1)
bool isless(double x, double y)              (2)
bool isless(half x, half y)                  (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ isless(NonScalar1 x, NonScalar2 y)

```

**Overloads (1) - (3):**

*Returns:* The value  $(x < y)$ .

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for `NonScalar1` and `NonScalar2`:
  - Both `NonScalar1` and `NonScalar2` are `marray`; or
  - `NonScalar1` and `NonScalar2` are any combination of `vec` and the `__swizzle__` type;
- `NonScalar1` and `NonScalar2` have the same number of elements and the same element type; and
- The element type is `float`, `double`, or `half`.

*Returns:* If `NonScalar1` is `marray`, the value  $(x[i] < y[i])$  for each element of `x` and `y`. If `NonScalar1` is `vec` or the `__swizzle__` type, returns the value  $((x[i] < y[i]) ? -1 : 0)$  for each element of `x` and `y`.

The return type depends on `NonScalar1`:

NonScalar1	Return Type
<code>marray&lt;float, N&gt;</code> <code>marray&lt;double, N&gt;</code> <code>marray&lt;half, N&gt;</code>	<code>marray&lt;bool, N&gt;</code>
<code>vec&lt;float, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;float, N&gt;</code>	<code>vec&lt;int32_t, N&gt;</code>
<code>vec&lt;double, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;double, N&gt;</code>	<code>vec&lt;int64_t, N&gt;</code>
<code>vec&lt;half, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;half, N&gt;</code>	<code>vec&lt;int16_t, N&gt;</code>

**islessequal**

```

bool islessequal(float x, float y)           (1)
bool islessequal(double x, double y)        (2)
bool islessequal(half x, half y)            (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ islessequal(NonScalar1 x, NonScalar2 y)

```

**Overloads (1) - (3):**

*Returns:* The value  $(x \leq y)$ .

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for `NonScalar1` and `NonScalar2`:
  - Both `NonScalar1` and `NonScalar2` are `marray`; or
  - `NonScalar1` and `NonScalar2` are any combination of `vec` and the `__swizzle__` type;
- `NonScalar1` and `NonScalar2` have the same number of elements and the same element type; and
- The element type is `float`, `double`, or `half`.

*Returns:* If `NonScalar1` is `marray`, the value  $(x[i] \leq y[i])$  for each element of `x` and `y`. If `NonScalar1` is `vec` or the `__swizzle__` type, returns the value  $((x[i] \leq y[i]) ? -1 : 0)$  for each element of `x` and `y`.

The return type depends on **NonScalar1**:

NonScalar1	Return Type
<code>marray&lt;float, N&gt;</code> <code>marray&lt;double, N&gt;</code> <code>marray&lt;half, N&gt;</code>	<code>marray&lt;bool, N&gt;</code>
<code>vec&lt;float, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;float, N&gt;</code>	<code>vec&lt;int32_t, N&gt;</code>
<code>vec&lt;double, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;double, N&gt;</code>	<code>vec&lt;int64_t, N&gt;</code>
<code>vec&lt;half, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;half, N&gt;</code>	<code>vec&lt;int16_t, N&gt;</code>

### islessgreater

```

bool islessgreater(float x, float y)           (1)
bool islessgreater(double x, double y)         (2)
bool islessgreater(half x, half y)             (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ islessgreater(NonScalar1 x, NonScalar2 y)

```

#### Overloads (1) - (3):

Returns: The value `(x < y) || (x > y)`.

#### Overload (4):

Constraints: Available only if all of the following conditions are met:

- One of the following conditions must hold for **NonScalar1** and **NonScalar2**:
  - Both **NonScalar1** and **NonScalar2** are `marray`; or
  - **NonScalar1** and **NonScalar2** are any combination of `vec` and the `__swizzle__` type;
- **NonScalar1** and **NonScalar2** have the same number of elements and the same element type; and
- The element type is `float`, `double`, or `half`.

Returns: If **NonScalar1** is `marray`, the value `(x[i] < y[i] || x[i] > y[i])` for each element of `x` and `y`. If **NonScalar1** is `vec` or the `__swizzle__` type, returns the value `((x[i] < y[i] || x[i] > y[i]) ? -1 : 0)` for each element of `x` and `y`.

The return type depends on **NonScalar1**:

NonScalar1	Return Type
<code>marray&lt;float, N&gt;</code> <code>marray&lt;double, N&gt;</code> <code>marray&lt;half, N&gt;</code>	<code>marray&lt;bool, N&gt;</code>
<code>vec&lt;float, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;float, N&gt;</code>	<code>vec&lt;int32_t, N&gt;</code>
<code>vec&lt;double, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;double, N&gt;</code>	<code>vec&lt;int64_t, N&gt;</code>

NonScalar1	Return Type
<code>vec&lt;half, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;half, N&gt;</code>	<code>vec&lt;int16_t, N&gt;</code>

**isfinite**

```

bool isfinite(float x)           (1)
bool isfinite(double x)         (2)
bool isfinite(half x)           (3)

template<typename NonScalar>    (4)
/*return-type*/ isfinite(NonScalar x)

```

**Overloads (1) - (3):**

*Returns:* The value `true` only if `x` has finite value.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* If `NonScalar` is `marray`, returns true for each element of `x` only if `x[i]` is a finite value. If `NonScalar` is `vec` or the `__swizzle__` type, returns -1 for each element of `x` if `x[i]` is a finite value and returns 0 otherwise.

The return type depends on `NonScalar`:

NonScalar	Return Type
<code>marray&lt;float, N&gt;</code> <code>marray&lt;double, N&gt;</code> <code>marray&lt;half, N&gt;</code>	<code>marray&lt;bool, N&gt;</code>
<code>vec&lt;float, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;float, N&gt;</code>	<code>vec&lt;int32_t, N&gt;</code>
<code>vec&lt;double, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;double, N&gt;</code>	<code>vec&lt;int64_t, N&gt;</code>
<code>vec&lt;half, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;half, N&gt;</code>	<code>vec&lt;int16_t, N&gt;</code>

**isinf**

```

bool isinf(float x)           (1)
bool isinf(double x)         (2)
bool isinf(half x)           (3)

template<typename NonScalar>    (4)
/*return-type*/ isinf(NonScalar x)

```

**Overloads (1) - (3):**

*Returns:* The value `true` only if `x` has an infinity value (either positive or negative).

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* If `NonScalar` is `marray`, returns true for each element of `x` only if `x[i]` has an infinity value. If `NonScalar` is `vec` or the `__swizzle__` type, returns -1 for each element of `x` if `x[i]` has an infinity value and returns 0 otherwise.

The return type depends on `NonScalar`:

NonScalar	Return Type
<code>marray&lt;float, N&gt;</code> <code>marray&lt;double, N&gt;</code> <code>marray&lt;half, N&gt;</code>	<code>marray&lt;bool, N&gt;</code>
<code>vec&lt;float, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;float, N&gt;</code>	<code>vec&lt;int32_t, N&gt;</code>
<code>vec&lt;double, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;double, N&gt;</code>	<code>vec&lt;int64_t, N&gt;</code>
<code>vec&lt;half, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;half, N&gt;</code>	<code>vec&lt;int16_t, N&gt;</code>

#### isnan

```

bool isnan(float x)           (1)
bool isnan(double x)         (2)
bool isnan(half x)           (3)

template<typename NonScalar> (4)
/*return-type*/ isnan(NonScalar x)

```

#### Overloads (1) - (3):

*Returns:* The value `true` only if `x` has a NaN value.

#### Overload (4):

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* If `NonScalar` is `marray`, returns true for each element of `x` only if `x[i]` has a NaN value. If `NonScalar` is `vec` or the `__swizzle__` type, returns -1 for each element of `x` if `x[i]` has a NaN value and returns 0 otherwise.

The return type depends on `NonScalar`:



NonScalar	Return Type
<code>marray&lt;float, N&gt;</code> <code>marray&lt;double, N&gt;</code> <code>marray&lt;half, N&gt;</code>	<code>marray&lt;bool, N&gt;</code>
<code>vec&lt;float, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;float, N&gt;</code>	<code>vec&lt;int32_t, N&gt;</code>
<code>vec&lt;double, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;double, N&gt;</code>	<code>vec&lt;int64_t, N&gt;</code>
<code>vec&lt;half, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;half, N&gt;</code>	<code>vec&lt;int16_t, N&gt;</code>

**isnormal**

```

bool isnormal(float x)           (1)
bool isnormal(double x)         (2)
bool isnormal(half x)           (3)

template<typename NonScalar>    (4)
/*return-type*/ isnormal(NonScalar x)

```

**Overloads (1) - (3):**

*Returns:* The value `true` only if `x` has a normal value.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- `NonScalar` is `marray`, `vec`, or the `__swizzle__` type; and
- The element type is `float`, `double`, or `half`.

*Returns:* If `NonScalar` is `marray`, returns true for each element of `x` only if `x[i]` has a normal value. If `NonScalar` is `vec` or the `__swizzle__` type, returns -1 for each element of `x` if `x[i]` has a normal value and returns 0 otherwise.

The return type depends on `NonScalar`:

NonScalar	Return Type
<code>marray&lt;float, N&gt;</code> <code>marray&lt;double, N&gt;</code> <code>marray&lt;half, N&gt;</code>	<code>marray&lt;bool, N&gt;</code>
<code>vec&lt;float, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;float, N&gt;</code>	<code>vec&lt;int32_t, N&gt;</code>
<code>vec&lt;double, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;double, N&gt;</code>	<code>vec&lt;int64_t, N&gt;</code>
<code>vec&lt;half, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;half, N&gt;</code>	<code>vec&lt;int16_t, N&gt;</code>

**isordered**

```

bool isordered(float x, float y)           (1)
bool isordered(double x, double y)        (2)
bool isordered(half x, half y)            (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ isordered(NonScalar1 x, NonScalar2 y)

```

**Overloads (1) - (3):**

*Effects:* Tests if **x** and **y** are ordered.

*Returns:* The value `isequal(x, x) && isequal(y, y)`.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for **NonScalar1** and **NonScalar2**:
  - Both **NonScalar1** and **NonScalar2** are **marray**; or
  - **NonScalar1** and **NonScalar2** are any combination of **vec** and the **\_\_swizzle\_\_** type;
- **NonScalar1** and **NonScalar2** have the same number of elements and the same element type; and
- The element type is **float**, **double**, or **half**.

*Effects:* Tests if each element of **x** and **y** are ordered.

*Returns:* If **NonScalar1** is **marray**, the value `isequal(x[i], x[i]) && isequal(y[i], y[i])` for each element of **x** and **y**. If **NonScalar1** is **vec** or the **\_\_swizzle\_\_** type, returns the value `((isequal(x[i], x[i]) && isequal(y[i], y[i]) ? -1 : 0)` for each element of **x** and **y**.

The return type depends on **NonScalar1**:

NonScalar1	Return Type
<b>marray</b> <float, N> <b>marray</b> <double, N> <b>marray</b> <half, N>	<b>marray</b> <bool, N>
<b>vec</b> <float, N> <b>__swizzle__</b> that is convertible to <b>vec</b> <float, N>	<b>vec</b> <int32_t, N>
<b>vec</b> <double, N> <b>__swizzle__</b> that is convertible to <b>vec</b> <double, N>	<b>vec</b> <int64_t, N>
<b>vec</b> <half, N> <b>__swizzle__</b> that is convertible to <b>vec</b> <half, N>	<b>vec</b> <int16_t, N>

**isunordered**

```

bool isunordered(float x, float y)         (1)
bool isunordered(double x, double y)      (2)
bool isunordered(half x, half y)          (3)

template<typename NonScalar1, typename NonScalar2> (4)
/*return-type*/ isunordered(NonScalar1 x, NonScalar2 y)

```

**Overloads (1) - (3):**

*Effects:* Tests if **x** and **y** are unordered.

*Returns:* The value `isnan(x) || isnan(y)`.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- One of the following conditions must hold for **NonScalar1** and **NonScalar2**:
  - Both **NonScalar1** and **NonScalar2** are `marray`; or
  - **NonScalar1** and **NonScalar2** are any combination of `vec` and the `__swizzle__` type;
- **NonScalar1** and **NonScalar2** have the same number of elements and the same element type; and
- The element type is `float`, `double`, or `half`.

*Effects:* Tests if each element of **x** and **y** are unordered.

*Returns:* If **NonScalar1** is `marray`, the value `isnan(x[i]) || isnan(y[i])` for each element of **x** and **y**. If **NonScalar1** is `vec` or the `__swizzle__` type, returns the value `((isnan(x[i]) || isnan(y[i]) ? -1 : 0)` for each element of **x** and **y**.

The return type depends on **NonScalar1**:

NonScalar1	Return Type
<code>marray&lt;float, N&gt;</code> <code>marray&lt;double, N&gt;</code> <code>marray&lt;half, N&gt;</code>	<code>marray&lt;bool, N&gt;</code>
<code>vec&lt;float, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;float, N&gt;</code>	<code>vec&lt;int32_t, N&gt;</code>
<code>vec&lt;double, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;double, N&gt;</code>	<code>vec&lt;int64_t, N&gt;</code>
<code>vec&lt;half, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;half, N&gt;</code>	<code>vec&lt;int16_t, N&gt;</code>

**signbit**

```

bool signbit(float x)           (1)
bool signbit(double x)         (2)
bool signbit(half x)           (3)

template<typename NonScalar>   (4)
/*return-type*/ signbit(NonScalar x)

```

**Overloads (1) - (3):**

*Returns:* The value `true` only if the sign bit of **x** is set.

**Overload (4):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar** is `marray`, `vec`, or the `__swizzle__` type; and

- The element type is `float`, `double`, or `half`.

*Returns:* If `NonScalar` is `marray`, returns true for each element of `x` only if the sign bit of `x[i]` is set. If `NonScalar` is `vec` or the `__swizzle__` type, returns -1 for each element of `x` if the sign bit of `x[i]` is set and returns 0 otherwise.

The return type depends on `NonScalar`:

NonScalar	Return Type
<code>marray&lt;float, N&gt;</code> <code>marray&lt;double, N&gt;</code> <code>marray&lt;half, N&gt;</code>	<code>marray&lt;bool, N&gt;</code>
<code>vec&lt;float, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;float, N&gt;</code>	<code>vec&lt;int32_t, N&gt;</code>
<code>vec&lt;double, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;double, N&gt;</code>	<code>vec&lt;int64_t, N&gt;</code>
<code>vec&lt;half, N&gt;</code> <code>__swizzle__</code> that is convertible to <code>vec&lt;half, N&gt;</code>	<code>vec&lt;int16_t, N&gt;</code>

any

```
template<typename GenInt>      (1)
/*return-type*/ any(GenInt x)

template<typename GenInt>      (2) /* deprecated */
bool any(GenInt x)
```

### Overload (1):

*Constraints:* Available only if `GenInt` is one of the following types:

- `marray<bool, N>`
- `vec<int8_t, N>`
- `vec<int16_t, N>`
- `vec<int32_t, N>`
- `vec<int64_t, N>`
- `__swizzle__` that is convertible to `vec<int8_t, N>`
- `__swizzle__` that is convertible to `vec<int16_t, N>`
- `__swizzle__` that is convertible to `vec<int32_t, N>`
- `__swizzle__` that is convertible to `vec<int64_t, N>`

*Returns:* When `x` is `marray`, returns a Boolean telling whether any element of `x` is true. When `x` is `vec` or the `__swizzle__` type, returns the value 1 if any element in `x` has its most significant bit set, otherwise returns the value 0.

The return type is `bool` if `GenInt` is `marray`. Otherwise, the return type is `int`.

### Overload (2):

This overload is deprecated in SYCL 2020.

*Constraints:* Available only if `GenInt` is one of the following types:

- `signed char`
- `short`
- `int`
- `long`
- `long long`
- `marray<signed char, N>`
- `marray<short, N>`
- `marray<int, N>`
- `marray<long, N>`
- `marray<long long, N>`

*Returns:* When `x` is a scalar, returns a Boolean telling whether the most significant bit of `x` is set. When `x` is `marray`, returns a Boolean telling whether the most significant bit of any element in `x` is set.

**all**

```
template<typename GenInt>      (1)
/*return-type*/ all(GenInt x)

template<typename GenInt>      (2) /* deprecated */
bool all(GenInt x)
```

#### Overload (1):

*Constraints:* Available only if `GenInt` is one of the following types:

- `marray<bool, N>`
- `vec<int8_t, N>`
- `vec<int16_t, N>`
- `vec<int32_t, N>`
- `vec<int64_t, N>`
- `__swizzle__` that is convertible to `vec<int8_t, N>`
- `__swizzle__` that is convertible to `vec<int16_t, N>`
- `__swizzle__` that is convertible to `vec<int32_t, N>`
- `__swizzle__` that is convertible to `vec<int64_t, N>`

*Returns:* When `x` is `marray`, returns a Boolean telling whether all elements of `x` are true. When `x` is `vec` or the `__swizzle__` type, returns the value 1 if all elements in `x` have their most significant bit set, otherwise returns the value 0.

The return type is `bool` if `GenInt` is `marray`. Otherwise, the return type is `int`.

#### Overload (2):

This overload is deprecated in SYCL 2020.

*Constraints:* Available only if `GenInt` is one of the following types:

- `signed char`
- `short`
- `int`
- `long`
- `long long`
- `marray<signed char, N>`
- `marray<short, N>`
- `marray<int, N>`
- `marray<long, N>`
- `marray<long long, N>`

**Returns:** When `x` is a scalar, returns a Boolean telling whether the most significant bit of `x` is set. When `x` is `marray`, returns a Boolean telling whether the most significant bit of all elements in `x` are set.

### bitselect

```
template<typename GenType1, typename GenType2, typename GenType3>
/*return-type*/ bitselect(GenType1 a, GenType2 b, GenType3 c)
```

**Constraints:** Available only if all of the following conditions are met:

- `GenType1` is one of the following types:
  - One of the *generic scalar types* as defined above;
  - `marray<T, N>`, where `T` is one of the *generic scalar types*;
  - `vec<T, N>`, where `T` is one of the *vector element types* as defined above; or
  - `__swizzle__` that is convertible to `vec<T, N>`, where `T` is one of the *vector element types*;
- If `GenType1` is not `vec` or the `__swizzle__` type, then `GenType2` and `GenType3` must be the same as `GenType1`; and
- If `GenType1` is `vec` or the `__swizzle__` type, then `GenType2` and `GenType3` must also be `vec` or the `__swizzle__` type, and all three must have the same element type and the same number of elements.

**Returns:** When the input parameters are scalars, returns a result where each bit of the result is the corresponding bit of `a` if the corresponding bit of `c` is 0. Otherwise it is the corresponding bit of `b`.

When the input parameters are not scalars, returns a result for each element where each bit of the result for element `i` is the corresponding bit of `a[i]` if the corresponding bit of `c[i]` is 0. Otherwise it is the corresponding bit of `b[i]`.

The return type is `GenType1` unless `GenType1` is the `__swizzle__` type, in which case the return type is the corresponding `vec`.

### select

```
template<typename Scalar> (1)
Scalar select(Scalar a, Scalar b, bool c)

template<typename NonScalar1, typename NonScalar2, typename NonScalar3> (2)
/*return-type*/ select(NonScalar1 a, NonScalar2 b, NonScalar3 c)
```

**Overload (1):**

*Constraints:* Available only if **Scalar** is one of the *generic scalar types* as defined above.

*Returns:* The value  $(c \text{ ? } b : a)$ .

**Overload (2):**

*Constraints:* Available only if all of the following conditions are met:

- **NonScalar1** is one of the following types:
  - **marray**<**T**, **N**>, where **T** is one of the *generic scalar types* as defined above;
  - **vec**<**T**, **N**>, where **T** is one of the *vector element types* as defined above; or
  - **\_\_swizzle\_\_** that is convertible to **vec**<**T**, **N**>, where **T** is one of the *vector element types*;
- If **NonScalar1** is **marray**, then:
  - **NonScalar2** must be the same as **NonScalar1**; and
  - **NonScalar3** must be **marray** with element type **bool** and the same number of elements as **NonScalar1**;
- If **NonScalar1** is **vec** or the **\_\_swizzle\_\_** type, then:
  - **NonScalar2** must also be **vec** or the **\_\_swizzle\_\_** type, and both must have the same element type and the same number of elements; and
  - **NonScalar3** must be **vec** or the **\_\_swizzle\_\_** type with the same number of elements as **NonScalar1**. The element type of **NonScalar3** must be a signed or unsigned integer with the same number of bits as the element type of **NonScalar1**.

*Returns:* If **NonScalar1** is **marray**, return the value  $(c[i] \text{ ? } b[i] : a[i])$  for each element of **a**, **b**, and **c**.

If **NonScalar1** is **vec** or the **\_\_swizzle\_\_** type, returns the value  $((\text{MSB of } c[i] \text{ is set}) \text{ ? } b[i] : a[i])$  for each element of **a**, **b**, and **c**.

The return type is **NonScalar1** unless **NonScalar1** is the **\_\_swizzle\_\_** type, in which case the return type is the corresponding **vec**.

# Chapter 5. SYCL Device Compiler

This section specifies the requirements of the SYCL device compiler. Most features described in this section relate to underlying [SYCL backend](#) capabilities of target devices and limiting the requirements of device code to ensure portability.

## 5.1. Offline compilation of SYCL source files

There are two alternatives for a SYCL [device compiler](#): a single-source device compiler and a device compiler that supports the technique of [SMCP](#).

A SYCL device compiler takes in a C++ source file, extracts only the SYCL kernels and outputs the device code in a form that can be enqueued from host code by the associated [SYCL runtime](#). How the [SYCL runtime](#) invokes the kernels is implementation-defined, but a typical approach is for a device compiler to produce a header file with the compiled kernel contained within it. By providing a command-line option to the host compiler, it would cause the implementation's SYCL header files to `#include` the generated header file. The SYCL specification has been written to allow this as an implementation approach in order to allow [SMCP](#). However, any of the mechanisms needed from the SYCL compiler, the [SYCL runtime](#) and build system are implementation-defined, as they can vary depending on the platform and approach.

A SYCL single-source device compiler takes in a C++ source file and compiles both host and device code at the same time. This specification specifies how a SYCL single-source device compiler sees and outputs device code for kernels, but does not specify the host compilation.

## 5.2. Naming of kernels

SYCL kernels are extracted from C++ source files and stored in an implementation-defined format. In the case of the shared-source compilation model, the kernels have to be uniquely identified by both host and device compiler. This is required in order for the host runtime to be able to load the kernel by using a backend-specific host runtime interface.

From this requirement the following rules apply for naming the kernels:

- The kernel name is a C++ typename.
- The kernel name must be forward declarable at namespace scope (including global namespace scope) and may not be forward declared other than at namespace scope. If it isn't forward declared but is specified as a template argument in a kernel invoking interface, as described in [Section 4.9.4.2](#), then it may not conflict with a name in any enclosing namespace scope.



The requirement that a kernel name be forward declarable makes some types for kernel names illegal, such as anything declared in the `std` namespace (adding a declaration to namespace `std` leads to undefined behavior).

- If the kernel is defined as a named function object type, the name can be the typename of the function object as long as it is either declared at namespace scope, or does not conflict with any name in an enclosing namespace scope.
- If the kernel is defined as a lambda, a typename can optionally be provided to the kernel invoking interface as described in [Section 4.9.4.2](#), so that the developer can control the kernel name for purposes such as debugging or referring to the kernel when applying build options.
- If a kernel function relies on template parameters, then those template parameters must be contained by the kernel name. If such a kernel name is specified as a template argument in a kernel invoking interface, then the template parameters on which the kernel depends must be forward declarable at namespace scope.



In both single-source and shared-source implementations, a device compiler should detect the kernel invocations (e.g. `parallel_for<kernelname>`) in the source code and compile the enclosed kernels, storing them with their associated type name.

The format of the kernel and the compilation techniques are details of an implementation and not specified. The interface between the compiler and the runtime for extracting and executing SYCL kernels on the device is a detail of an implementation and not specified.

## 5.3. Compilation of functions

The SYCL device compiler parses an entire C++ source file supplied by the user, including any header files referenced via `#include` directives. From this source file, the SYCL device compiler must compile kernels for the device, as well as any functions that the kernels call.

The device compiler identifies kernels by looking for calls to [Kernel invocation commands](#) such as `parallel_for`. One of the parameters is a function object which is known as a [SYCL kernel function](#), and this function must always return `void`. Any function called by the [SYCL kernel function](#) is also compiled for the device, and these functions together with the [SYCL kernel functions](#) are known as [device functions](#). The device compiler searches recursively for any functions called from a [device function](#), and these functions are also compiled for the device and known as [device functions](#).

To illustrate, the following source code shows three functions and a kernel invoke with comments explaining which functions need to be compiled for the device.

```

1 void f(handler& cgh) {
2     // Function "f" is not compiled for device
3
4     cgh.single_task( [= ] {
5         // This code is compiled for device
6         g(); // This line forces "g" to be compiled for device
7     });
8 }
9
10 void g() {
11     // Called from kernel, so "g" is compiled for device
12 }
13
14 void h() {
15     // Not called from a device function, so not compiled for device
16 }
```

In order for the SYCL device compiler to correctly compile [device functions](#), all functions in the source file, whether [device functions](#) or not, must be syntactically correct functions according to this specification. A syntactically correct function adheres to at least the minimum required C++ version defined in [Section 3.9.1](#).

## 5.4. Language restrictions for device functions

[Device functions](#) must abide by certain restrictions. The full set of C++ features are not available to these functions. Following is a list of these restrictions:

- Pointers and objects containing pointers may be shared. However, when a pointer is passed between SYCL devices or between the host and a SYCL device, dereferencing that pointer on the device produces undefined behavior unless the device supports [USM](#) and the pointer is an address within a [USM](#) memory region (see [Section 4.8](#)).

- Memory storage allocation is not allowed in kernels. All memory allocation for the device is done on the host using accessor classes or using [USM](#) as explained in [Section 4.8](#). Consequently, the default allocation `operator new` overloads that allocate storage are disallowed in a SYCL kernel. The placement `new` operator and any user-defined overloads that do not allocate storage are permitted.
- Kernel functions must always have a `void` return type. A kernel lambda trailing-return-type that is not `void` is therefore illegal, as is a return statement (that would return from the kernel function) with an expression that does not convert to `void`.
- The odr-use of polymorphic classes and classes with virtual inheritance is allowed. However, no virtual member functions are allowed to be called in a [device function](#).
- No function pointers or references are allowed to be called in a [device function](#).
- RTTI is disabled inside [device functions](#).
- No variadic functions are allowed to be called in a [device function](#).
- Exception-handling cannot be used inside a [device function](#). `noexcept` is allowed.
- Recursion is not allowed in a [device function](#).
- Variables with thread storage duration (`thread_local` storage class specifier) are not allowed to be odr-used in a [device function](#).
- Variables with static storage duration that are odr-used inside a [device function](#), must be either `const` or `constexpr`, and must also be either zero-initialized or constant-initialized.



Amongst other things, this restriction makes it illegal for a [device function](#) to access a global variable that isn't `const` or `constexpr`.

- The rules for kernels apply to both the kernel function objects themselves and all functions, operators, member functions, constructors and destructors called by the kernel. This means that kernels can only use library functions that have been adapted to work with SYCL. Implementations are not required to support any library routines in kernels beyond those explicitly mentioned as usable in kernels in this spec. Developers should refer to the SYCL built-in functions in [Section 4.17](#) to find functions that are specified to be usable in kernels.
- Interacting with a special [SYCL runtime](#) class (e.g. SYCL `accessor` or `stream`) that is stored within a C++ union is undefined behavior.
- Any variable or function that is odr-used from a [device function](#) must be defined in the same translation unit as that use. However, a function may be defined in another translation unit if the implementation defines the `SYCL_EXTERNAL` macro as described in [Section 5.10.1](#).

Inside a [discarded statement](#) or in the case of a [manifestly constant-evaluated expression or conversion](#), any code accepted by the C++ standard in this case is also accepted in a SYCL [device function](#).



The restriction waiver in [discarded statement](#) or [manifestly constant-evaluated expression or conversion](#) allows any kind of meta-programming in a [device function](#).

## 5.5. Built-in scalar data types

In a SYCL device compiler, the device definition of all standard C++ fundamental types from [Table 153](#) must match the host definition of those types, in both size and alignment. A device compiler may have this preconfigured so that it can match them based on the definitions of those types on the platform, or there may be a necessity for a device compiler command-line option to ensure the types are the same.

The standard C++ fixed width types, e.g. `int8_t`, `int16_t`, `int32_t`, `int64_t`, should have the same size as defined by the C++ standard for host and device.

*Table 153. Fundamental data types supported by SYCL*

Fundamental data type	Description
<code>bool</code>	A conditional data type which can be either true or false. The value true expands to the integer constant 1 and the value false expands to the integer constant 0.
<code>char</code>	A signed or unsigned 8-bit integer, as defined by the C++ core language
<code>signed char</code>	A signed 8-bit integer, as defined by the C++ core language
<code>unsigned char</code>	An unsigned 8-bit integer, as defined by the C++ core language
<code>short int</code>	A signed integer of at least 16-bits, as defined by the C++ core language
<code>unsigned short int</code>	An unsigned integer of at least 16-bits, as defined by the C++ core language
<code>int</code>	A signed integer of at least 16-bits, as defined by the C++ core language
<code>unsigned int</code>	An unsigned integer of at least 16-bits, as defined by the C++ core language
<code>long int</code>	A signed integer of at least 32-bits, as defined by the C++ core language
<code>unsigned long int</code>	An unsigned integer of at least 32-bits, as defined by the C++ core language
<code>long long int</code>	An integer of at least 64-bits, as defined by the C++ core language
<code>unsigned long long int</code>	An unsigned integer of at least 64-bits, as defined by the C++ core language
<code>float</code>	A 32-bit floating-point. The float data type must conform to the IEEE 754 single precision storage format.
<code>double</code>	A 64-bit floating-point. The double data type must conform to the IEEE 754 double precision storage format. This type is only supported on devices that have <code>aspect::fp64</code> .

## 5.6. Preprocessor directives and macros

The standard C++ preprocessing directives and macros are supported. The following preprocessor macros must be defined by all conformant implementations:

- `SYCL_LANGUAGE_VERSION` is defined to an integer literal that indicates the version of the SYCL specification to which the implementation conforms.

SYCL version	Macro defined as
SYCL 2020	<code>202012L</code>

Future versions of the SYCL specification will define this macro to an integer literal with greater value.

- `SYCL_DEVICE_COPYABLE` is defined to 1 if the implementation supports explicitly specified [device copyable](#) types as described in [Section 3.13.1](#). Otherwise, the implementation's definition of device copyable falls back to C++ trivially copyable and `sycl::is_device_copyable` is ignored;
- `__SYCL_DEVICE_ONLY__` is defined to 1 if the source file is being compiled with a SYCL device compiler which does not produce host binary;
- `__SYCL_SINGLE_SOURCE__` is defined to 1 if the source file is being compiled with a SYCL single-source compiler which produces host as well as device binary;
- `SYCL_FEATURE_SET_FULL` is defined to 1 if the SYCL implementation supports the full feature set and is not defined otherwise. For more details see [Appendix B](#);
- `SYCL_FEATURE_SET_REDUCED` is defined to 1 if the SYCL implementation supports the reduced feature set and not the full feature set, otherwise it is not defined. For more details see [Appendix B](#);
- `SYCL_EXTERNAL` is an optional macro which enables external linkage of SYCL functions and member functions to be included in a SYCL kernel. The macro is only defined if the implementation supports external linkage. For more details see [Section 5.10.1](#).

In addition, for each [SYCL backend](#) supported, the preprocessor macros described in [Section 4.1](#) must be defined by all conformant implementations.

## 5.7. Optional kernel features

A number of kernel features defined by this SYCL specification are optional; they may be supported on some devices but not on other devices.

As stated in [Section 5.4](#), the restrictions for optional kernel features do not apply to discarded statements or to manifestly constant-evaluated expressions or conversions in device code. Device code may use optional features in [discarded statement](#) or [manifestly constant-evaluated expression or conversion](#) even if the device does not support the optional feature.

As described in [Section 4.6.4.5](#), an application can test whether a device supports an optional feature by testing whether the device has an associated aspect. The following aspects are those that correspond to optional kernel features:

- `fp16`
- `fp64`
- `atomic64`

In addition, the following C++ attributes from [Section 5.8.1](#) also correspond to optional kernel features because they force the kernel to be compiled in a way that might not run on all devices:

- `reqd_work_group_size()`

- `reqd_sub_group_size()`

In order to guarantee source code portability of SYCL applications that use optional kernel features, all SYCL implementations must be able to compile device code that uses these optional features regardless of whether the implementation supports the features on any of its devices.

Of course, applications that make use of optional kernel features should ensure that a kernel using such a feature is submitted only to a device that supports the feature. If the application submits a [command group](#) using a secondary queue, then any kernel submitted from the [command group](#) should use only features that are supported by both the primary queue's device and the secondary queue's device. If an application fails to do this, the implementation must throw a synchronous exception with the `errc::kernel_not_supported` error code from the [kernel invocation command](#) (e.g. `parallel_for()`).

It is legal for a SYCL application to define several kernels in the same translation unit even if they use different optional features, as shown in the following example:

```

1 queue q1(dev1);
2 if (dev1.has(aspect::fp16)) {
3   q1.submit([&](handler& cgh) {
4     cgh.parallel_for<KernelA>(range{N}, [=](id i) {
5       half fpShort = 1.0;
6       /* ... */
7     });
8   });
9 }
10
11 queue q2(dev2);
12 if (dev2.has(aspect::atomic64)) {
13   q2.submit([&](handler& cgh) {
14     cgh.parallel_for<KernelB>(range{N}, [=](id i) {
15       /* ... */
16       sycl::atomic_ref longAtomic(longValue);
17       longAtomic.fetch_add(1);
18     });
19   });
20 }
```

An implementation may not raise a compile time diagnostic or a run time exception merely due to speculative compilation of a kernel for a device when the application does not actually submit the kernel to that device. To illustrate using the example above, assume that device `dev1` does not have `aspect::atomic64` and device `dev2` does not have `aspect::fp16`. An implementation cannot raise a diagnostic due to compilation of `KernelA` for device `dev2` or for compilation of `KernelB` for device `dev1` because the application does not submit these kernels to those devices.



It is expected that this requirement will have an impact on the way an implementation bundles kernels into device images. For example, naively bundling `KernelA` and `KernelB` into the same device image could run afoul of this requirement if the implementation compiles the entire device image when `KernelA` is submitted to device `dev1`.

## 5.8. Attributes for device code

C++ attributes may be used to decorate kernels and device functions in order to influence the code generated by the device compiler. These attributes are all defined in the `[[sycl::]]` namespace.

If one of the attributes defined in this section is applied to a kernel or device function, it must be applied

to the first declaration of that kernel or device function in the translation unit. Programs which fail to do this are ill formed and the compiler must issue a diagnostic. Redeclarations of the kernel or device function in the same translation unit may optionally have the same attribute applied (so long as the attribute arguments are the same between the declarations), but this is not required. The attribute remains in effect regardless of whether it appears in the redeclaration.

Unless an attribute's description specifically allows it, a kernel or device function may not be declared with the more than one instance of the same attribute unless all instances have the same attribute arguments. The compiler must issue a diagnostic for programs which violate this requirement. When two or more instances of the same attribute appear on the declaration of a kernel or device function, the effect is as though a single instance appeared (assuming that all instances have the same attribute arguments).

If a kernel or device function is declared with an attribute in one translation unit and the same kernel or device function is declared without the same attribute (and its same attribute arguments) in another translation unit, the program is ill formed and no diagnostic is required.

If any of these attributes are applied to a device function that is also compiled for the host, they have no effect when the function is compiled for the host.

Applying these attributes to any language construct other than those specified in this section has implementation-defined effect.

### 5.8.1. Kernel attributes

The attributes listed in [Table 154](#) have a different position depending on whether the kernel is defined as a lambda expression or as a named function object. If the kernel is a named function object, the attribute is applied to the declarator-id in the function declaration. However, if the kernel is a lambda expression, the attribute is applied to the lambda declarator.



The reason for the different positions is because the C++ core language does not currently define a position for attributes to appertain to the lambda's corresponding function operator or operator template, only to the corresponding *type* of the function operator or operator template. This is expected to be remedied in a future version of the C++ core language specification.

The example below demonstrates these attribute positions using the `[[sycl::reqd_work_group_size(16)]]` attribute. Note that the C++ core language allows two possible positions for kernels that are defined as a named function object.

```

1 // Kernel defined as a lambda
2 myQueue.submit([&](handler& h) {
3     h.parallel_for(range<1>(16),
4                     [=](item<1> it) [[sycl::reqd_work_group_size(16)]] {
5                         //[kernel code]
6                     });
7 });
8
9 // Kernel defined as a named function object
10 class KernelFunctor1 {
11 public:
12     [[sycl::reqd_work_group_size(16)]] void operator()(item<1> it) const {
13         //[kernel code]
14     };
15 };
16
17 // Kernel defined as a named function object

```

```

18 class KernelFunctor2 {
19 public:
20     void operator() [[sycl::reqd_work_group_size(16)]] (item<1> it) const {
21         //[kernel code]
22     };
23 };

```

Table 154. Attributes for kernel functions

SYCL attribute	Description
<pre> reqd_work_group_size(dim0) reqd_work_group_size(dim0, dim1) reqd_work_group_size(dim0, dim1, dim2) </pre>	<p>Indicates that the kernel must be launched with the specified work-group size. The number of arguments must match the dimensionality of the work-group used to invoke the kernel, and the order of the arguments matches the order of the dimension extents to the <code>range</code> constructor. Each argument must be an integral constant expression that is representable by <code>size_t</code>.</p> <p>Kernels that are decorated with this attribute may not call functions that are defined in another translation unit via the <code>SYCL_EXTERNAL</code> macro.</p> <p>Each device may have limitations on the work-group sizes that it supports. If a kernel is decorated with this attribute and then submitted to a device that does not support the work-group size, the implementation must throw a synchronous <code>exception</code> with the <code>errc::kernel_not_supported</code> error code. If the kernel is submitted to a device that does support the work-group size, but the application provides an <code>nd_range</code> that does not match the size from the attribute, then the implementation must throw a synchronous <code>exception</code> with the <code>errc::nd_range</code> error code.</p>



SYCL attribute	Description
<pre>work_group_size_hint(dim0) work_group_size_hint(dim0, dim1) work_group_size_hint(dim0, dim1, dim2)</pre>	<p>Provides a hint to the compiler about the work-group size most likely to be used when launching the kernel at runtime. The number of arguments must match the dimensionality of the work-group used to invoke the kernel, and the order of the arguments matches the order of the dimension extents to the <code>range</code> constructor. Each argument must be an integral constant expression that is representable by <code>size_t</code>. The effect of this attribute, if any, is implementation-defined.</p>
<pre>vec_type_hint(&lt;type&gt;)</pre>	<p>Hint to the compiler on the vector computational width of the kernel. The argument must be one of the vector types defined in <a href="#">Section 4.14.2</a>. The effect of this attribute, if any, is implementation-defined.</p> <p>This attribute is deprecated (available for use, but will likely be removed in a future version of the specification and is not recommended for use in new code).</p>



SYCL attribute	Description
<code>reqd_sub_group_size(size)</code>	<p>Indicates that the kernel must be compiled and executed with the specified sub-group size. The argument to the attribute must be an integral constant expression that is representable by <code>uint32_t</code>.</p> <p>Kernels that are decorated with this attribute may not call functions that are defined in another translation unit via the <code>SYCL_EXTERNAL</code> macro.</p> <p>Each device supports only certain sub-group sizes as defined by <code>info::device::sub_group_sizes</code>. In addition, some device features may be incompatible with certain sub-group sizes. If a kernel is decorated with this attribute and then submitted to a device that does not support the sub-group size or if the kernel uses a feature that the device does not support with this sub-group size, the implementation must throw a synchronous <code>exception</code> with the <code>errc::kernel_not_supported</code> error code.</p>

SYCL attribute	Description
<div data-bbox="164 185 472 219" style="border: 1px solid #ccc; padding: 5px; margin-bottom: 10px;"> <code>device_has(<span style="color: #d62728;">aspect</span>, ...)</code> </div>	<p>This attribute may be used to decorate either the declaration of a kernel function that is defined in the current translation unit or to decorate the declaration of a non-kernel device function. The following description applies when the attribute decorates a kernel function.</p> <p>The parameter list to the <code>sycl::device_has()</code> attribute consists of zero or more integral constant expressions, where each integer is interpreted as one of the enumerated values in the <code>sycl::aspect</code> enumeration type.</p> <p>Specifying this attribute on a kernel has two effects. First, it causes the <a href="#">kernel invocation command</a> to throw a synchronous exception with the <code>errc::kernel_not_supported</code> error code if the kernel is submitted to a device that does not have one of the listed aspects. (This includes the device associated with the secondary queue if the kernel is submitted from a <a href="#">command group</a> that has a secondary queue.) Second, it causes the compiler to issue a diagnostic if the kernel (or any of the functions it calls) uses an optional feature that is associated with an aspect that is not listed in the attribute.</p> <p>The value of each parameter to this attribute must be equal to one of the values in the <code>sycl::aspect</code> enumeration type (including any extended values the implementation may provide). If it does not, the program is ill formed and the compiler must issue a diagnostic.</p> <p>See below for an example of this attribute.</p>

*Example of the `sycl::device_has()` attribute*

```

1 class KernelFunctor {
2 public:
3   [[sycl::device_has(aspect::fp16)]] void operator()(item<1> it) const {
4     foo();

```

```

5     bar();
6 };
7
8 private:
9     void foo() const {
10         half fp = 1.0; // No compiler diagnostic here
11     }
12
13     void bar() const {
14         sycl::atomic_ref longAtomic(longValue);
15         longAtomic.fetchAdd(1); // ERROR: Compiler issues diagnostic because
16                                // "aspect::atomic64" missing from "device_has()"
17     }
18 };
19
20 // Using "sycl::device_has()" does not provide any guarantee that the device
21 // actually supports the required features. Therefore, the host code should
22 // still check the device's aspects before submitting the kernel.
23 if (myQueue.get_device().has(aspect::fp16)) {
24     myQueue.submit(
25         [&](handler& h) { h.parallel_for(range{16}, KernelFunctor{}); });
26 }

```

## 5.8.2. Device function attributes

The attributes in [Table 155](#) are applied to the declaration of a non-kernel device function. The position of the attribute is the same as for the kernel function attributes defined above in [Section 5.8.1](#).

*Table 155. Attributes for non-kernel device functions*

SYCL attribute	Description
<div data-bbox="137 159 987 244" style="border: 1px solid #ccc; padding: 10px; margin-bottom: 10px;"> <code>device_has(<span style="color: #e91e63;">aspect</span>, ...)</code> </div>	<p>This attribute may be used to decorate either the declaration of a kernel function that is defined in the current translation unit or to decorate the declaration of a non-kernel device function. The following description applies when the attribute decorates a non-kernel device function declaration.</p> <p>The syntax of this attribute's parameter list is the same as the syntax for the form of <code>sycl::device_has()</code> that is specified on a kernel function (see <a href="#">Table 154</a>).</p> <p>This attribute is required when a non-kernel device function that uses optional device features is called in one translation unit and defined in another translation unit via the <code>SYCL_EXTERNAL</code> macro.</p> <p>When this attribute appears in a translation unit that calls the decorated device function, it is an assertion that the device function uses optional features that correspond to the aspects listed in the attribute. The program is ill formed if the called device function uses optional features that do not correspond to any of the aspects listed in the attribute, or if the function uses optional features and the attribute is not specified. No diagnostic is required in this case.</p> <p>When this attribute appears in a translation unit that defines the decorated device function, it causes the compiler to issue a diagnostic if the device function (or any of the functions it calls) uses an optional feature that is associated with an aspect that is not listed in the attribute.</p>

## 5.9. Address-space deduction

C++ has no type-level support to represent address spaces. As a consequence, the SYCL generic programming model does not directly affect the C++ type of unannotated pointers and references.

Source level guarantees about address spaces in the SYCL generic programming model can only be achieved using pointer classes (instances of `multi_ptr`), which are regular classes that represent pointers to data stored in the corresponding address spaces.

In SYCL, the address space of pointer and references are derived from:

- Accessors that give access to shared data. They can be bound to a memory object in a command group and passed into a kernel. Accessors are used in scheduling of kernels to define ordering. Accessors to buffers have a compile-time address space based on their access mode.
- Explicit pointer classes (e.g. `global_ptr`) holds a pointer which is known to be addressing the address space represented by the `access::address_space`. This allows the compiler to determine whether the pointer references global, local, constant or private memory and generate code accordingly.
- Raw C++ pointer and reference types (e.g. `int*`) are allowed within SYCL kernels. They can be constructed from the address of local variables, explicit pointer classes, or accessors.

### 5.9.1. Address space assignment

In order to understand where data lives, the device compiler is expected to assign address spaces while lowering types for the underlying target based on the context. Depending on the [SYCL backends](#) and mode, address space deducing rules differ slightly.

If the target of the SYCL backend can represent the generic address space, then the "common address space deduction rules" in [Section 5.9.2](#) and the "generic as default address space rules" in [Section 5.9.3](#) apply. If the target of the SYCL backend cannot represent the generic address space, then the "common address space deduction rules" in [Section 5.9.2](#) and the "inferred address space rules" in [Section 5.9.4](#) apply.



SYCL address space does not affect the type, address space shall be understood as memory segment in which data is allocated. For instance, if `int i;` is allocated to the global address space, then `decltype(&i)` shall evaluate to `int*`.

### 5.9.2. Common address space deduction rules

The variable declarations get assigned to an address space depending on their scope and storage class:

- Namespace scope
  - If the type is `const`, the declaration is assigned to an implementation-defined address space. If the target of the SYCL backend can represent the generic address space, then the assigned address space must be compatible with the generic address space.



Namespace scope non-`const` declarations cannot be used within a kernel, as restricted in [Section 5.4](#). This means that non-`const` global variables cannot be accessed by any device kernel or code called by the device kernel.

- Block scope and function parameter scope
  - Declarations with static storage duration are treated the same way as variables in namespace scope
  - Otherwise the declaration is assigned to the local address space if declared in a hierarchical context
  - Otherwise the declaration is assigned to the private address space
- Class scope
  - Static data members are treated the same way as for variable in namespace scope

If a prvalue-to-xvalue conversion happens as part of an initialization expression, then the result is assigned to the same address space as the entity being initialized. Otherwise, if the conversion happens in a block scope or function parameter scope, the result is assigned to the local address space if it happens in a hierarchical context otherwise it is assigned to the private address space. If the prvalue-to-xvalue conversion happens in another scope, the result is assigned in the same way as declaration in namespace scope.

### 5.9.3. Generic as default address space

For SYCL backends that can represent the generic address space (see [Section 5.9.1](#)), unannotated pointers and references are considered to be pointing to the generic address space.

### 5.9.4. Inferred address space



*Note for this version*

The address space deduction feature described next is inherited from the SYCL 1.2.1 specifications. This section will be changed in a future version to better align with addition of generic address space and generic as default address space.

For SYCL backends that cannot represent the generic address space (see [Section 5.9.1](#)), inside kernels the SYCL device compiler will need to auto-deduce the memory region of unannotated pointer and reference types during the lowering of types from C++ to the underlying representation.

If a kernel function or device function contains a pointer or reference type, then the address space deduction must be attempted using the following rules:

- If an explicit pointer class is converted into a C++ pointer value, then the C++ pointer value will point to same address space as the one represented by the explicit pointer class.
- If a variable is declared as a pointer type, but initialized in its declaration to a pointer value with an already-deduced address space, then that variable will have the same address space as its initializer.
- If a function parameter is declared as a pointer type, and the argument is a pointer value with a deduced address space, then the function will be compiled as if the parameter had the same address space as its argument. It is legal for a function to be called in different places with different address spaces for its arguments: in this case the function is said to be “duplicated” and compiled multiple times. Each duplicated instance of the function must compile legally in order to have defined behavior.
- If a function return type is declared as a pointer type and return statements use address space deduced expressions, then the function will be compiled as if the return type had the same address space. To compile legally, all return expressions must deduce to the same address space.
- The rules for pointer types also apply to reference types. i.e. a reference variable takes its address space from its initializer. A function with a reference parameter takes its address space from its argument.
- If no other rule above can be applied to a declaration of a pointer, then it is assumed to be in the private address space.

It is illegal to assign a pointer value addressing one address space to a pointer variable addressing a different address space.

## 5.10. SYCL offline linking

### 5.10.1. SYCL functions and member functions linkage

By default, any function that is odr-used from a [device function](#) must be defined in the same translation

unit as that use. However, this restriction is relaxed if both of the following conditions are met:

- The implementation defines the `SYCL_EXTERNAL` macro;
- The translation unit that calls the function declares the function with `SYCL_EXTERNAL` as described below.

When a function is declared with `SYCL_EXTERNAL`, that macro must be used on the first declaration of that function in the translation unit. Redclarations of the function in the same translation unit may optionally use `SYCL_EXTERNAL`, but this is not required.

When a function is declared with `SYCL_EXTERNAL`, that function must also be defined in some translation unit, where the function is declared with `SYCL_EXTERNAL`.

A function may only be declared with `SYCL_EXTERNAL` if it has external linkage by normal C++ rules.

A function declared with `SYCL_EXTERNAL` may be called from both host and device code. The macro has no effect when the function is called from host code.

In order to declare a function with `SYCL_EXTERNAL`, the macro name `SYCL_EXTERNAL` must appear before the function declaration. If the function is also decorated with C++ attributes that appear before the declaration, the `SYCL_EXTERNAL` may appear before, after, or between these attributes. The following example demonstrates the use of `SYCL_EXTERNAL`.

```
1 #include <sycl/sycl.hpp>
2
3 SYCL_EXTERNAL void Foo();
4
5 SYCL_EXTERNAL void Bar() { /* ... */ }
6
7 SYCL_EXTERNAL extern void Baz();
8
9 [[nodiscard]] SYCL_EXTERNAL void Important();
10
11 SYCL_EXTERNAL [[nodiscard]] void AlsoImportant();
```

Functions that are declared using `SYCL_EXTERNAL` have the following additional restrictions beyond those imposed on other device functions:

- If the SYCL backend does not support the generic address space then the function cannot use raw pointers as parameter or return types. Explicit pointer classes must be used instead;
- The function cannot call `group::parallel_for_work_item`;
- The function cannot be called from a `parallel_for_work_group` scope.

## Chapter 6. SYCL Extensions

This chapter describes the mechanism by which the [core SYCL specification](#) can be extended. Some parts of this chapter are requirements that all implementations must follow if they extend the [core SYCL specification](#), while other parts of the chapter are merely guidelines. Unless a requirement is specifically stated as normative, all content in this chapter is a non-normative guideline.

An extension can be either of two flavors: an extension ratified by the Khronos SYCL group or a vendor supplied extension. In both cases, an extension is an optional feature set which an implementation need not implement in order to be conformant with the [core SYCL specification](#).

Vendors may choose to define extensions in order to expose custom features or to gather feedback on an API that is not yet ready for inclusion in the [core SYCL specification](#). Once a vendor extension has stabilized, the vendor is encouraged to promote it to a future version of the [core SYCL specification](#) or to a ratified Khronos extension. Thus, vendor extensions can be viewed as a pipeline of features for consideration in future SYCL versions.

The Khronos SYCL group may define extensions for features that are not yet ready for the [core SYCL specification](#) but are implemented by more than one vendor. These extensions also may be considered for inclusion in a future version of the [core SYCL specification](#).

This chapter does not describe any particular extension to SYCL. Rather, it describes the *mechanism* for defining an extension. Each extension is defined by its own separate document. If an extension is ratified by the Khronos SYCL group, that group will release a document describing the extension. If a vendor defines an extension, the vendor is responsible for releasing its documentation.

### 6.1. Definition of an extension

An extension can take many possible forms. Some examples include:

- adding new types or free functions to the SYCL runtime;
- modifying existing SYCL classes, structs, or enumeration types by adding new members, member functions, or enumerated values;
- adding new overloads for existing free functions or member functions;
- defining new specializations for existing SYCL templates;
- adding new C++ attributes;
- adding new predefined macros;
- adding new keywords to the language;
- adding a new backend.

An extension may also broaden the definition of existing functions defined in the [core SYCL specification](#) by defining semantics for cases that are left unspecified by the [core SYCL specification](#).

### 6.2. Requirements for an extension

This section is normative. All vendors which provide an extension must abide by the requirements described here.

An extension may not change the definition of existing functions defined by the [core SYCL specification](#) in a way that changes their specified behavior. Also, an extension may not remove any feature defined by the [core SYCL specification](#).

The vendor must choose at least one `<vendorstring>` which uniquely identifies the vendor's SYCL imple-



mentation. The Khronos SYCL group does not provide any registry of the strings, so each vendor is responsible for choosing its own. One way to choose a unique string is to use the vendor's company name or a marketing name that is associated with the vendor's implementation. Ultimately, it is each vendor's responsibility to choose a string that is unique. The strings "khr" and "KHR" are reserved for the Khronos SYCL group for its own extensions, so vendors may not use these as a `<vendorstring>`.

The implementation must predefine at least one macro of the form `SYCL_IMPLEMENTATION_<vendorstring>` which allows applications to test whether they are being compiled with that vendor's implementation. For example, the Acme vendor could predefine a macro whose name is `SYCL_IMPLEMENTATION_ACME`.

## 6.3. Guidelines for portable extensions

Vendors who want to ensure that their extension does not collide with other vendors' extensions or with future versions of the [core SYCL specification](#) should follow the additional rules specified in this section. However, this is not a requirement for conformance.

### 6.3.1. Extension namespace

If an extension adds new types or free functions, it should avoid adding these directly in the `sycl::` namespace since future versions of the [core SYCL specification](#) may also add new identifiers in this namespace. The namespace `sycl::ext::<vendorstring>` is reserved for use by extensions. For example, the Acme vendor could define extended types and free functions in the namespace `sycl::ext::acme`, and this would guarantee that they will not collide with definitions in other vendors' extensions or with future versions of the [core SYCL specification](#).

### 6.3.2. Names for extensions to existing classes or enumerations

An extension may add new members or member functions to existing SYCL classes or new values to existing SYCL enumeration types. To ensure these extensions do not collide, vendors are encouraged to name them with the prefix `ext_<vendorstring>_`. For example, the Acme vendor could add a new member function to the `sycl::device` class named `device::ext_acme_fancy()` or a new value to the `sycl::aspect` enumeration named `aspect::ext_acme_fancier`.

In some cases, an extension does not have the freedom to choose a specific function name. For example, this could happen if the extension adds a new constructor overload for an existing SYCL class. In cases like this, the extension should ensure that one of the function parameters has a type that is defined in the extension's namespace. For example, the Acme vendor could add a new constructor for `sycl::context` with the signature `context(ext::acme::frobber&)`.

A similar situation can occur if an existing SYCL template is specialized with an extended enumerated value. Obviously, the extension cannot rename the template in this case. Instead, it is sufficient that the template is specialized with an extended enumerated value, and this guarantees that the extended specialization will not collide.



Vendors are encouraged to use the `ext_<vendorstring>_` prefix form when possible for additions to existing SYCL classes because this form makes the extension's vendor name apparent. People reading application code will immediately know that a member function is an extension, and they will immediately know which vendor's documentation to consult.

### 6.3.3. Feature test macros

Vendors are encouraged to group a related set of extensions together into a "feature" and to predefine a feature-test macro when the implementation supports the extensions in that feature. The feature-test macro should have the following form to ensure it is unique: `SYCL_EXT_<vendorstring>_<featurename>`. For

example, the Acme vendor might define a feature-test macro named `SYCL_EXT_ACME_FANCYFEATURE`. This allows applications to protect code using the extension with `#ifdef`, so that the code is skipped when compiled with an implementation that doesn't support the feature.

Since the interface to an extension might change from one release to another, vendors are also encouraged to predefine the macro's value to the version of the extension. Vendors should use a numerical value that monotonically increases for each revision of the extension API.

Of course, an extension may also predefine other macros. In order to ensure that these macro names do not collide with other extensions or future versions of the [core SYCL specification](#), the name should start with the prefix `SYCL_EXT_<vendorstring>` or `SYCL_IMPLEMENTATION_<vendorstring>`.

### 6.3.4. Attribute namespace

An extension may define new C++ attributes. The attribute namespace `sycl::` is reserved for the [core SYCL specification](#), so vendors should choose a different namespace for any attributes they add.

### 6.3.5. Include file paths

An extension may define new `#include` files under the `"sycl"` path. The path prefix `"sycl/ext/<vendorstring>"` is reserved for this purpose. For example, the Acme vendor could add a header file `"sycl/ext/acme/fancy.h"` and be guaranteed that it would not conflict with other extensions or with future versions of the [core SYCL specification](#).

### 6.3.6. Optional kernel features

An extension may also add new optional kernel features—features which are supported on some devices but not on others. Vendors are encouraged to follow the same mechanism outlined in [Section 5.7](#). Therefore, an extended optional kernel feature should have a matching extension to the `sycl::aspect` enumerated type.

### 6.3.7. Adding a backend

An extension may also add a new backend. If it does, the naming of the backend APIs follows the normal guidelines for extensions and also follows the naming pattern for backends that are defined in the [core SYCL specification](#). To illustrate:

- The extension should add a new value to the `sycl::backend` enumeration type using a naming scheme like `ext_<vendorstring>_<backendname>`. For example, if the Acme vendor adds a backend named "foo", it would add an enumerated value named `sycl::backend::ext_acme_foo`.
- The extension should define the backend's interop API in a namespace named `sycl::ext::<vendorstring>::<backendname>`. For our hypothetical Acme example, this would be a namespace named `sycl::ext::acme::foo`.
- If the backend interop API is available through a separate header file, that header should be named `"sycl/ext/<vendorstring>/backend/<backendname>.hpp"`. For our hypothetical Acme example this would be `"sycl/ext/acme/backend/foo.hpp"`.
- The extension should predefine a macro for the backend when it is "active". The name of this macro should be `SYCL_EXT_<vendorstring>_BACKEND_<backendname>`. For our hypothetical Acme example this would be `SYCL_EXT_ACME_BACKEND_FOO`.

# Appendix A: Information descriptors

This appendix contains the definitions of all the SYCL information descriptors introduced in [Chapter 4](#).

## A.1. Platform information descriptors

The following interface includes all the information descriptors for the `platform` class.

```
1 namespace sycl {
2 namespace info {
3 namespace platform {
4
5 struct version;
6 struct name;
7 struct vendor;
8 struct extensions; // Deprecated
9
10 } // namespace platform
11 } // namespace info
12 } // namespace sycl
```

## A.2. Context information descriptors

The following interface includes all the information descriptors for the `context` class.

```
1 namespace sycl {
2 namespace info {
3 namespace context {
4
5 struct platform;
6 struct devices;
7 struct atomic_memory_order_capabilities;
8 struct atomic_fence_order_capabilities;
9 struct atomic_memory_scope_capabilities;
10 struct atomic_fence_scope_capabilities;
11
12 } // namespace context
13 } // namespace info
14 } // namespace sycl
```

## A.3. Device information descriptors

The following interface includes all the information descriptors for the `device` class.

```
1 namespace sycl {
2 namespace info {
3 namespace device {
4
5 struct device_type;
6 struct vendor_id;
7 struct max_compute_units;
```

```

8 struct max_work_item_dimensions;
9 template <int Dimensions = 3> struct max_work_item_sizes;
10 struct max_work_group_size;
11 struct max_num_sub_groups;
12 struct sub_group_sizes;
13 struct preferred_vector_width_char;
14 struct preferred_vector_width_short;
15 struct preferred_vector_width_int;
16 struct preferred_vector_width_long;
17 struct preferred_vector_width_float;
18 struct preferred_vector_width_double;
19 struct preferred_vector_width_half;
20 struct native_vector_width_char;
21 struct native_vector_width_short;
22 struct native_vector_width_int;
23 struct native_vector_width_long;
24 struct native_vector_width_float;
25 struct native_vector_width_double;
26 struct native_vector_width_half;
27 struct max_clock_frequency;
28 struct address_bits;
29 struct max_mem_alloc_size;
30 struct image_support; // Deprecated
31 struct max_read_image_args;
32 struct max_write_image_args;
33 struct image2d_max_height;
34 struct image2d_max_width;
35 struct image3d_max_height;
36 struct image3d_max_width;
37 struct image3d_max_depth;
38 struct image_max_buffer_size;
39 struct max_samplers;
40 struct max_parameter_size;
41 struct mem_base_addr_align;
42 struct half_fp_config;
43 struct single_fp_config;
44 struct double_fp_config;
45 struct global_mem_cache_type;
46 struct global_mem_cache_line_size;
47 struct global_mem_cache_size;
48 struct global_mem_size;
49 struct max_constant_buffer_size; // Deprecated
50 struct max_constant_args; // Deprecated
51 struct local_mem_type;
52 struct local_mem_size;
53 struct error_correction_support;
54 struct host_unified_memory;
55 struct atomic_memory_order_capabilities;
56 struct atomic_fence_order_capabilities;
57 struct atomic_memory_scope_capabilities;
58 struct atomic_fence_scope_capabilities;
59 struct profiling_timer_resolution;
60 struct is_endian_little;
61 struct is_available;
62 struct is_compiler_available; // Deprecated
63 struct is_linker_available; // Deprecated

```

```

64 struct execution_capabilities;
65 struct queue_profiling; // Deprecated
66 struct built_in_kernels; // Deprecated
67 struct built_in_kernel_ids;
68 struct platform;
69 struct name;
70 struct vendor;
71 struct driver_version;
72 struct profile;
73 struct version;
74 struct backend_version;
75 struct aspects;
76 struct extensions; // Deprecated
77 struct printf_buffer_size;
78 struct preferred_interop_user_sync;
79 struct parent_device;
80 struct partition_max_sub_devices;
81 struct partition_properties;
82 struct partition_affinity_domains;
83 struct partition_type_property;
84 struct partition_type_affinity_domain;
85
86 } // namespace device
87
88 enum class device_type : /* unspecified */ {
89     cpu, // Maps to OpenCL CL_DEVICE_TYPE_CPU
90     gpu, // Maps to OpenCL CL_DEVICE_TYPE_GPU
91     accelerator, // Maps to OpenCL CL_DEVICE_TYPE_ACCELERATOR
92     custom, // Maps to OpenCL CL_DEVICE_TYPE_CUSTOM
93     automatic, // Maps to OpenCL CL_DEVICE_TYPE_DEFAULT
94     host,
95     all // Maps to OpenCL CL_DEVICE_TYPE_ALL
96 };
97
98 enum class partition_property : /* unspecified */ {
99     no_partition,
100     partition_equally,
101     partition_by_counts,
102     partition_by_affinity_domain
103 };
104
105 enum class partition_affinity_domain : /* unspecified */ {
106     not_applicable,
107     numa,
108     L4_cache,
109     L3_cache,
110     L2_cache,
111     L1_cache,
112     next_partitionable
113 };
114
115 enum class local_mem_type : /* unspecified */ { none, local, global };
116
117 enum class fp_config : /* unspecified */ {
118     denorm,
119     inf_nan,

```

```

120 round_to_nearest,
121 round_to_zero,
122 round_to_inf,
123 fma,
124 correctly_rounded_divide_sqrt,
125 soft_float
126 };
127
128 enum class global_mem_cache_type : /* unspecified */ {
129     none,
130     read_only,
131     read_write
132 };
133
134 enum class execution_capability : /* unspecified */ {
135     exec_kernel,
136     exec_native_kernel
137 };
138
139 } // namespace info
140 } // namespace sycl

```

## A.4. Queue information descriptors

The following interface includes all the information descriptors for the `queue` class.

```

1 namespace sycl {
2 namespace info {
3 namespace queue {
4
5 struct context;
6 struct device;
7
8 } // namespace queue
9 } // namespace info
10 } // namespace sycl

```

## A.5. Kernel information descriptors

The following interface includes all the information descriptors that apply to kernels as described in [Table 115](#) and in [Table 116](#).

```

1 namespace sycl {
2 namespace info {
3 namespace kernel {
4
5 struct num_args;
6 struct attributes;
7
8 } // namespace kernel
9
10 namespace kernel_device_specific {
11

```

```

12 struct global_work_size;
13 struct work_group_size;
14 struct compile_work_group_size;
15 struct preferred_work_group_size_multiple;
16 struct private_mem_size;
17 struct max_num_sub_groups;
18 struct compile_num_sub_groups;
19 struct max_sub_group_size;
20 struct compile_sub_group_size;
21
22 } // namespace kernel_device_specific
23
24 } // namespace info
25 } // namespace sycl

```

## A.6. Event information descriptors

The following interface includes all the information descriptors for the `event` class as described in [Table 17](#) and [Table 19](#).

```

1 namespace sycl {
2 namespace info {
3 namespace event {
4
5 struct command_execution_status;
6
7 } // namespace event
8
9 enum class event_command_status : /* unspecified */ {
10     submitted,
11     running,
12     complete
13 };
14
15 namespace event_profiling {
16
17 struct command_submit;
18 struct command_start;
19 struct command_end;
20
21 } // namespace event_profiling
22 } // namespace info
23 } // namespace sycl

```

## Appendix B: Feature sets

As of SYCL 2020 there are now two distinct feature sets which a SYCL implementation can conform to, in order to better fit the requirements of different domains, such as embedded, mobile, and safety critical, which may have limitations because of the toolchains used.

A SYCL implementation can choose to conform to either the full feature set or the reduced feature set.

### B.1. Full feature set

The full feature set includes all features specified in the [core SYCL specification](#) with no exceptions.

### B.2. Reduced feature set

The reduced feature set makes certain features optional or restricted to specific forms. The following list defines all the differences between the reduced feature set and the full feature set.

1. **Un-named SYCL kernel functions:** [SYCL kernel functions](#) which are defined using a lambda expression and therefore have no standard name are required to be provided a name via the kernel name template parameter of kernel invocation functions such as `parallel_for`. This overrides the [core SYCL specification](#) rules for [SYCL kernel function](#) naming as specified in [Section 4.9.4.2](#).
2. **Address space mode:** The [address space assignment](#) mode used in the reduced feature set is not required to be [generic address space](#), regardless of SYCL backend in use. Instead the [inferred address space](#) mode may always be used.
3. **Declarations:** In addition to the requirements specified in [Section 5.9.2](#), the reduced feature set does not require support for odr-use inside [device functions](#) of variables declared `const` or `constexpr` with static storage duration.

### B.3. Compatibility

In order to avoid introducing any kind of divergence the reduced and full feature sets are defined such that the full feature set is a subsumption of the reduced feature set. This means that any applications which are developed for the reduced feature set will be compatible with both a SYCL reduced implementation and a SYCL full implementation.

### B.4. Conformance

One of the reasons for having this be defined in the specification is that hardware vendors which wish to support SYCL on their platform(s) want to be able to demonstrate their support for it by passing conformance. However, if passing conformance means adopting features which they do not believe to be necessary at an additional development effort then this may deter them.

Each feature set has its own route for passing conformance allowing adopters of SYCL to specify the feature set they wish to test conformance against. The conformance test suite would then alter or disable the tests within the test suite according to how the feature sets are differentiated above.



# Appendix C: OpenCL backend specification

This chapter describes how the SYCL general programming model is mapped on top of OpenCL, and how the SYCL generic interoperability interface must be implemented by vendors providing SYCL for OpenCL implementations to ensure SYCL applications written for the OpenCL backend are interoperable.

## C.1. SYCL application interoperability native backend objects

For each SYCL runtime class which supports SYCL application interoperability, specializations of `backend_traits::input_type` and `backend_traits::return_type` must be defined as the type of SYCL application interoperability native backend object associated with `SyclType` for the SYCL backend.

The types of the native backend objects for SYCL application interoperability are described in Table 160.

## C.2. Kernel function interoperability native backend objects

For each SYCL runtime class which supports kernel function interoperability, a specialization of `backend_traits::return_type` must be defined as the type of kernel function interoperability native backend object associated with `SyclType` for the SYCL backend.

The types of the native backend objects for kernel function interoperability are described in Table 156.

Table 156. Types of native backend objects kernel function interoperability

SyclType	backend_return_t<backend::opencl, SyclType>
<code>accessor&lt;T, Dims, Mode, target::device&gt;</code>	<code>__global T*</code>
<code>accessor&lt;T, Dims, Mode, target::constant_buffer&gt;</code>	<code>__constant T*</code>
<code>accessor&lt;T, Dims, Mode, target::local&gt;</code>	<code>__local T*</code>
<code>local_accessor&lt;T, Dims&gt;</code>	<code>__local T*</code>
<code>sampler_image_accessor&lt;T, 1, Mode, image_target::device&gt;</code>	<code>sampler_1dimage_pair_t</code>
<code>sampler_image_accessor&lt;T, 2, Mode, image_target::device&gt;</code>	<code>sampler_2dimage_pair_t</code>
<code>sampler_image_accessor&lt;T, 3, Mode, image_target::device&gt;</code>	<code>sampler_3dimage_pair_t</code>
<code>unsampler_image_accessor&lt;T, 1, Mode, image_target::device&gt;</code>	<code>image1d_t</code>
<code>unsampler_image_accessor&lt;T, 2, Mode, image_target::device&gt;</code>	<code>image2d_t</code>
<code>unsampler_image_accessor&lt;T, 3, Mode, image_target::device&gt;</code>	<code>image3d_t</code>
<code>stream</code>	<code>__global cl_char*</code>
<code>device_event</code>	<code>event_t</code>

The `sampler_1dimage_pair_t`, `sampler_2dimage_pair_t` and `sampler_3dimage_pair_t` types must be implemented as described below.

```

1 struct sampler_1dimage_pair_t {
2     sampler_t sampler;
3     image1d_t image;
4 }
5
6 struct sampler_2dimage_pair_t {
7     sampler_t sampler;
```

```

8  image2d_t image;
9 }
10
11 struct sampler_3dimage_pair_t {
12     sampler_t sampler;
13     image3d_t image;
14 }

```

## C.3. Destruction of interop constructed objects with reference semantics

On destruction of the last copy of an instance of a SYCL class which is specified to have reference semantics as described in [Section 4.5.2](#) that was constructed using one of the [SYCL backend](#) interoperability `make_*` functions specified in [Section 4.5.1.3](#) additional lifetime related operations may be performed which are required for the underlying [native backend object](#).

The additional behavior performed by the OpenCL [SYCL backend](#) for each SYCL class is described in [Table 157](#).

Table 157. Destructor behavior of interop constructed objects with reference semantics

SYCL object	Destructor behavior
accessor	No additional behavior is performed.
buffer	<code>clReleaseMemObject</code> will be called on the native <code>cl_mem</code> object provided during construction.
context	<code>clReleaseContext</code> will be called on the native <code>cl_context</code> object provided during construction.
device	<code>clReleaseDevice</code> will be called on the native <code>cl_device</code> object provided during construction.
event	<code>clReleaseEvent</code> will be called on the native <code>cl_event</code> object provided during construction.
kernel	<code>clReleaseKernel</code> will be called on the native <code>cl_kernel</code> objects provided during construction.
kernel_bundle	<code>clReleaseProgram</code> will be called on the native <code>cl_program</code> objects provided during construction.
platform	No additional behavior is performed.
queue	<code>clReleaseCommandQueue</code> will be called on the native <code>cl_command_queue</code> object provided during construction.
sampld_image	<code>clReleaseMemObject</code> will be called on the native <code>cl_mem</code> object provided during construction.
unsampld_image	<code>clReleaseMemObject</code> will be called on the native <code>cl_mem</code> object provided during construction.

## C.4. SYCL for OpenCL framework

The SYCL framework allows applications to use a host and one or more OpenCL devices as a single heterogeneous parallel computer system. The framework contains the following components:

- [SYCL C++ template library](#): The template library provides a set of C++ templates and classes which

provide the programming model to the user. It enables the creation of runtime classes such as SYCL queues, buffers and images, as well as access to some underlying OpenCL runtime object, such as contexts, platforms, devices and program objects.

- **SYCL runtime:** The **SYCL runtime** interfaces with the underlying OpenCL implementations and handles scheduling of commands in queues, moving of data between host and devices, manages contexts, programs, kernel compilation and memory management.
- **OpenCL Implementation(s):** The SYCL system assumes the existence of one or more OpenCL implementations available on the host machine.
- **SYCL device compilers:** The SYCL **device compilers** compile SYCL C++ kernels into a format which can be executed on an OpenCL device at runtime. There may be more than one SYCL device compiler in a SYCL implementation. The format of the compiled SYCL kernels is not defined. A SYCL device compiler may, or may not, also compile the host parts of the program.

The OpenCL backend is enabled using the `sycl::backend::opencl` value of `enum class backend`. That means that when the OpenCL backend is active, the value of `sycl::is_backend_active<sycl::backend::opencl>::value` will be `true`.

## C.5. Mapping of SYCL programming model on top of OpenCL

The SYCL programming model was originally designed as a high-level model for the OpenCL API, hence the mapping of SYCL on the OpenCL API is mostly straightforward.

When the OpenCL backend is active on a SYCL application, all visible OpenCL platforms are exported as SYCL platforms.

When a SYCL implementation executes kernels on an OpenCL device, it achieves this by enqueueing OpenCL **commands** to execute computations on the processing elements within a device. The processing elements within an OpenCL compute unit may execute a single stream of instructions as ALUs within a SIMD unit (which execute in lockstep with a single stream of instructions), as independent SPMD units (where each PE maintains its own program counter) or as some combination of the two.

### C.5.1. Backend specific information descriptors

Some of the SYCL information descriptors are backend-defined. For the OpenCL backend these information descriptors map directly to OpenCL properties as described in the table below:

Table 158. Mapping of SYCL information descriptors to OpenCL properties

SYCL	OpenCL
<code>info::platform::version</code>	<code>CL_PLATFORM_VERSION</code>
<code>info::device::version</code>	<code>CL_DEVICE_VERSION</code>

### C.5.2. OpenCL memory model

The memory model for SYCL devices running on OpenCL platforms follows the memory model of the OpenCL version they conform to.

In addition to **global memory**, **local memory** and **private memory** memory, the OpenCL backend permits the use of **constant memory** space in SYCL:

- **Constant-memory** is a region of memory that remains constant during the execution of a kernel. A

pointer to the generic address space cannot represent an address to this memory region.

Work-items executing in a kernel have access to four distinct memory regions, with the mapping between SYCL and OpenCL described in [Table 159](#).

*Table 159. Mapping of SYCL memory regions into OpenCL memory regions*

SYCL	OpenCL
Global	Global memory
Constant	Constant memory
Local	Local memory
Private	Private memory

### C.5.3. OpenCL interface for buffer command accessors

The enumerator `target::constant_buffer` is deprecated, but will remain a part of the OpenCL backend as an extension. This enables SYCL kernel functions to access the contents of a buffer through the OpenCL device's constant memory.

### C.5.4. OpenCL resources managed by SYCL application

In OpenCL, a developer must create a [context](#) to be able to execute commands on a device. Creating a context involves choosing a [platform](#) and a list of [devices](#). In SYCL, contexts, platforms and devices all exist, but the user can choose whether to specify them or have the SYCL implementation create them automatically. The minimum required object for submitting work to devices in SYCL is the [queue](#), which contains references to a platform, device and context internally.

The resources managed by SYCL are:

1. **Platforms:** all features of OpenCL are implemented by platforms. A platform can be viewed as a given hardware vendor's runtime and the devices accessible through it. Some devices will only be accessible to one vendor's runtime and hence multiple platforms may be present. SYCL manages the different platforms for the user. In SYCL, a platform resource is accessible through a `sycl::platform` object.
2. **Contexts:** any OpenCL resource that is acquired by the user is attached to a context. A context contains a collection of devices that the host can use and manages memory objects that can be shared between the devices. Data movement between devices within a context may be efficient and hidden by the underlying OpenCL runtime while data movement between contexts may involve the host. A given context can only wrap devices owned by a single platform. In SYCL, a context resource is accessible through a `sycl::context` object.
3. **Devices:** platforms provide one or more devices for executing kernels. In SYCL, a device is accessible through a `sycl::device` object.
4. **Kernel bundles:** OpenCL objects that store implementation data for the SYCL kernels. These objects are only required for advanced use in SYCL and are encapsulated in the `sycl::kernel_bundle` class.
5. **Queues:** SYCL kernels execute in command queues. The user must create a queue, which references an associated context, platform and device. The context, platform and device may be chosen automatically, or specified by the user. In SYCL, command queues are accessible through `sycl::queue` objects.

## C.6. Interoperability with the OpenCL API

The OpenCL backend for SYCL ensures maximum compatibility between SYCL and OpenCL kernels and API. This includes supporting devices with different capabilities and support for different versions of the OpenCL C language, in addition to supporting SYCL kernels written in C++.

**SYCL runtime** classes which encapsulate an OpenCL opaque type such as SYCL **context** or SYCL **queue** must provide an interoperability constructor taking an instance of the OpenCL opaque type. When the OpenCL object supports reference counting, these constructors must retain that instance to increase the reference count of the OpenCL resource. Likewise, the destructor for the **SYCL runtime** classes which encapsulate a reference counted OpenCL opaque type must release that instance to decrease the reference count of the OpenCL resource. Since the OpenCL **platform\_id** is not reference counted, the encapsulating SYCL **platform** class neither retains nor releases this OpenCL resource.

Note that an instance of a **SYCL runtime** class which encapsulates an OpenCL opaque type can encapsulate any number of instances of the OpenCL type, unless it was constructed via the interoperability constructor, in which case it can encapsulate only a single instance of the OpenCL type.

The lifetime of a **SYCL runtime** class that encapsulates an OpenCL opaque type and the instance of that opaque type retrieved via the **get\_native()** free function are not tied in either direction given correct usage of OpenCL reference counting. For example if a user were to retrieve a **cl\_command\_queue** instance from a SYCL **queue** instance and then immediately destroy the SYCL **queue** instance, the **cl\_command\_queue** instance is still valid. Or if a user were to construct a SYCL **queue** instance from a **cl\_command\_queue** instance and then immediately release the **cl\_command\_queue** instance, the SYCL **queue** instance is still valid.

Note that a **SYCL runtime** class that encapsulates an OpenCL opaque type is not responsible for any incorrect use of OpenCL reference counting outside of the **SYCL runtime**. For example if a user were to retrieve a **cl\_command\_queue** instance from a SYCL **queue** instance and then release the **cl\_command\_queue** instance more than once without any prior retain then the SYCL **queue** instance that the **cl\_command\_queue** instance was retrieved from is now undefined.

Note that an instance of the SYCL **buffer** or SYCL **image** class templates constructed via the interoperability constructor is free to copy from the **cl\_mem** into another memory allocation within the **SYCL runtime** to achieve normal SYCL semantics, for as long as the SYCL **buffer** or SYCL **image** instance is alive.

Table 160 relates SYCL objects to their OpenCL native type in the SYCL application.

Table 160. List of native types per SYCL object in the OpenCL backend

SyclType	backend_input_t<backend::opencl, SyclType>	backend_return_t<backend::opencl, SyclType>	Description
<b>platform</b>	<b>cl_platform_id</b>	<b>cl_platform_id</b>	A SYCL platform object encapsulates an OpenCL platform ID.
<b>device</b>	<b>cl_device_id</b>	<b>cl_device_id</b>	A SYCL device object encapsulates an OpenCL device ID.
<b>context</b>	<b>cl_context</b>	<b>cl_context</b>	A SYCL context object encapsulates an OpenCL context object.
<b>queue</b>	<b>cl_command_queue</b>	<b>cl_command_queue</b>	A SYCL queue object encapsulates an OpenCL queue object.
<b>kernel</b>	<b>cl_kernel</b>	<b>cl_kernel</b>	A SYCL kernel object encapsulates an OpenCL kernel object.

SyclType	backend_in- put_t<back- end::opengl, SyclType>	backend_re- turn_t<back- end::opengl, SyclType>	Description
template <bundle_s tate State> kernel_bu ndle<Stat e>	cl_program	std::vec- tor<cl_pro- gram>	A SYCL kernel bundle can encapsulate one or more OpenCL program objects. It can also encapsulate one or more OpenCL kernel objects which can be retrieved using the appropriate <b>kernel</b> object.
event	std::vec- tor<cl_event>	std::vec- tor<cl_event>	A SYCL event can encapsulate one or multiple OpenCL events, representing a number of dependencies in the same or different contexts, that must be satisfied for the SYCL event to be complete.
buffer	cl_mem	std::vec- tor<cl_mem>	SYCL buffers containing OpenCL memory objects can handle multiple <b>cl_mem</b> objects in the same or different context. The interoperability interface will return a list of active buffers in the SYCL runtime.
sampld_i mage	cl_mem	std::vec- tor<cl_mem>	SYCL sampled images containing OpenCL image objects can handle multiple underlying <b>cl_mem</b> objects at the same time in the same or different OpenCL contexts. The interoperability interface will return a list of active images in the SYCL runtime.
unsampled _image	cl_mem	std::vec- tor<cl_mem>	SYCL unsampled images containing OpenCL image objects can handle multiple underlying <b>cl_mem</b> objects at the same time in the same or different OpenCL contexts. The interoperability interface will return a list of active images in the SYCL runtime.

Inside the SYCL kernel, the SYCL API offers interoperability with OpenCL device types. [Table 161](#) describes the mapping of kernel types.

Table 161. List of native types per SYCL object on kernel code

SYCL kernel native types in OpenCL	Description
multi_ptr::get_decorated()	Returns a pointer in the OpenCL address space corresponding to the type of multi pointer object

When a buffer or image is allocated on more than one OpenCL device, if these devices are on separate contexts then multiple **cl\_mem** objects may be allocated for the memory object, depending on whether the object has actively been used on these devices yet or not.

The OpenCL C function qualifier **\_\_kernel** and the access qualifiers: **\_\_read\_only**, **\_\_write\_only** and **\_\_read\_write** are not exposed in SYCL via keywords, but are instead encapsulated in SYCL's parameter passing system inside accessors. Users wishing to achieve the OpenCL equivalent of these qualifiers in SYCL should instead use SYCL accessors with equivalent semantics.

Any OpenCL C function included in a pre-built OpenCL library can be defined as an **extern "C"** function and the OpenCL program has to be linked against any SYCL program that contains kernels using the external function. In this case, the data types used have to comply with the interoperability aliases defined in [Table 163](#).



## C.7. Programming interface

The following section describes the OpenCL-specific API.

### C.7.1. Construct SYCL objects from OpenCL ones

The OpenCL backend provides the following specializations of the `make_{sycl_class}` template functions which are defined in [Section 4.5.1.3](#). These functions are in the `sycl` namespace.

OpenCL interoperability function	Description
<pre>context make_context(const cl_context&amp; clContext,                     const async_handler&amp; asyncHandler = {})</pre>	Constructs a SYCL <code>context</code> instance from an OpenCL <code>cl_context</code> in accordance with the requirements described in <a href="#">Section 4.5.1</a> .
<pre>event make_event(const std:: vector&lt;cl_event&gt;&amp; clEvents,                 const context&amp; syclContext)</pre>	Constructs a SYCL <code>event</code> instance from a vector of OpenCL <code>cl_event</code> objects in accordance with the requirements described in <a href="#">Section 4.5.1</a> .
<pre>device make_device(const cl_device_id&amp; clDeviceId)</pre>	Constructs a SYCL <code>device</code> instance from an OpenCL <code>cl_device_id</code> in accordance with the requirements described in <a href="#">Section 4.5.1</a> .
<pre>platform make_platform(const cl_platform_id&amp; clPlatformId)</pre>	Constructs a SYCL <code>platform</code> instance from an OpenCL <code>cl_platform_id</code> in accordance with the requirements described in <a href="#">Section 4.5.1</a> .
<pre>queue make_queue(const cl_command_queue&amp; clQueue, const context&amp; syclContext,                 const async_handler&amp; asyncHandler = {})</pre>	Constructs a SYCL <code>queue</code> instance with an optional <code>async_handler</code> from an OpenCL <code>cl_command_queue</code> in accordance with the requirements described in <a href="#">Section 4.5.1</a> .
<pre>template &lt;typename T, int Dimensions = 1,         typename AllocatorT = buffer_allocator&lt;std::remove_const_ _t&lt;T&gt;&gt;&gt; buffer&lt;T, Dimensions, AllocatorT&gt; make_buffer(const cl_mem&amp; clMemObject,  const context&amp; syclContext,  event availableEvent)</pre>	Available only when: <code>Dimensions == 1</code> .  Constructs a SYCL <code>buffer</code> instance from an OpenCL <code>cl_mem</code> in accordance with the requirements described in <a href="#">Section 4.5.1</a> . The instance of the SYCL <code>buffer</code> class template being constructed must wait for the SYCL <code>event</code> parameter, <code>availableEvent</code> to signal that the <code>cl_mem</code> instance is ready to be used. The SYCL <code>context</code> parameter <code>syclContext</code> is the context associated with the memory object.

OpenCL interoperability function	Description
<pre> template &lt;typename T, int Dimensions = 1,         typename AllocatorT = buffer_allocator&lt;std::remove_const_ _t&lt;T&gt;&gt;&gt; buffer&lt;T, Dimensions, AllocatorT&gt; make_buffer(const cl_mem&amp; clMemObject,  const context&amp; syclContext) </pre>	<p>Available only when: <code>Dimensions == 1</code>.</p> <p>Constructs a SYCL <code>buffer</code> instance from an OpenCL <code>cl_mem</code> in accordance with the requirements described in <a href="#">Section 4.5.1</a>.</p>
<pre> template &lt;int Dimensions = 1, typename AllocatorT = image_allocator&gt; sampled_image&lt;Dimensions, AllocatorT&gt; make_sampled_image(const cl_mem&amp; clMemObject, const context&amp; syclContext,                      image_sampler syclImageSampler, event availableEvent) </pre>	<p>Constructs a SYCL <code>sampled_image</code> instance from an OpenCL <code>cl_mem</code> in accordance with the requirements described in <a href="#">Section 4.5.1</a>. The instance of the SYCL <code>image</code> class template being constructed must wait for the SYCL <code>event</code> parameter, <code>availableEvent</code> to signal that the <code>cl_mem</code> instance is ready to be used. The SYCL <code>context</code> parameter <code>syclContext</code> is the context associated with the memory object.</p>
<pre> template &lt;int Dimensions = 1, typename AllocatorT = image_allocator&gt; sampled_image&lt;Dimensions, AllocatorT&gt; make_sampled_image(const cl_mem&amp; clMemObject, const context&amp; syclContext,                      image_sampler syclImageSampler) </pre>	<p>Constructs a SYCL <code>sampled_image</code> instance from an OpenCL <code>cl_mem</code> in accordance with the requirements described in <a href="#">Section 4.5.1</a>. The SYCL <code>context</code> parameter <code>syclContext</code> is the context associated with the memory object.</p>
<pre> template &lt;int Dimensions = 1, typename AllocatorT = image_allocator&gt; unsampled_image&lt;Dimensions, AllocatorT&gt; make_unsampled_image(const cl_mem&amp; clMemObject, const context&amp; syclContext,                      event availableEvent) </pre>	<p>Constructs a SYCL <code>unsampled_image</code> instance from an OpenCL <code>cl_mem</code> in accordance with the requirements described in <a href="#">Section 4.5.1</a>. The instance of the SYCL <code>image</code> class template being constructed must wait for the SYCL <code>event</code> parameter, <code>availableEvent</code> to signal that the <code>cl_mem</code> instance is ready to be used. The SYCL <code>context</code> parameter <code>syclContext</code> is the context associated with the memory object.</p>



OpenCL interoperability function	Description
<pre>template &lt;int Dimensions = 1, typename AllocatorT = image_allocator&gt; unsampled_image&lt;Dimensions, AllocatorT&gt; make_unsampled_image(const cl_mem&amp; clMemObject, const context&amp; syclContext)</pre>	<p>Constructs a SYCL <code>unsampled_image</code> instance from an OpenCL <code>cl_mem</code> in accordance with the requirements described in <a href="#">Section 4.5.1</a>.</p>
<pre>kernel make_kernel(const cl_kernel&amp; clKernel, const context&amp; syclContext);</pre>	<p>Constructs a SYCL <code>kernel</code> instance from an OpenCL kernel object.</p>
<pre>template &lt;bundle_state State&gt; kernel_bundle&lt;State&gt; make_kernel_bundle(const cl_program&amp; clProgram,  const context&amp; syclContext)</pre>	<p>Constructs a SYCL <code>kernel_bundle</code> instance from an OpenCL <code>cl_program</code> for the devices in <code>syclContext</code> in accordance with the requirements described in <a href="#">Section 4.5.1</a>. The SYCL <code>context</code> must represent the same underlying OpenCL context associated with the OpenCL program object.</p> <p>The <code>state</code> specifies the expected <code>kernel_bundle</code> state. The mapping between the <code>kernel_bundle</code> state and OpenCL program state (<code>CL_PROGRAM_BINARY_TYPE</code>) is as follows:</p> <ul style="list-style-type: none"> <li>• <code>bundle_state::input</code> - <code>CL_PROGRAM_BINARY_TYPE_NONE</code></li> <li>• <code>bundle_state::object</code> - <code>CL_PROGRAM_BINARY_TYPE_COMPILED_OBJECT</code> or <code>CL_PROGRAM_BINARY_TYPE_INTERMEDIATE</code> or <code>CL_PROGRAM_BINARY_TYPE_LIBRARY</code>.</li> <li>• <code>bundle_state::executable</code> - <code>CL_PROGRAM_BINARY_TYPE_EXECUTABLE</code></li> </ul> <p>If the internal state of the OpenCL program doesn't match <code>state</code>, the kernel bundle will be compiled and linked as necessary. If the OpenCL program is already an executable binary, but the specified <code>state</code> is not <code>bundle_state::executable</code>, an <code>exception</code> with the <code>errc::invalid</code> error code is thrown. If the specified <code>state</code> is <code>bundle_state::input</code>, but the OpenCL program already has a binary associated with it, an <code>exception</code> with the <code>errc::invalid</code> error code is thrown.</p> <p>Throws an <code>exception</code> with the <code>errc::invalid</code> error code if any error is produced by the <a href="#">SYCL backend</a>.</p>

## C.7.2. Extension query

Platforms and devices with an OpenCL backend may support extensions. For convenience, the extensions supported by a platform or device can be queried through the following functions provided in the `sycl::opencl` namespace.

Extension query	Description
<pre>bool has_extension(const sycl::platform&amp; syclPlatform,                     const std::string&amp; extension)</pre>	Returns true if the OpenCL platform associated with <code>syclPlatform</code> supports the extension identified by <code>extension</code> , otherwise it returns false. If <code>syclPlatform.get_backend() != sycl::backend::opencl</code> an exception with the <code>errc::backend_mismatch</code> error code is thrown.
<pre>bool has_extension(const sycl::device&amp; syclDevice, const std::string&amp; extension)</pre>	Returns true if the OpenCL device associated with <code>syclDevice</code> supports the extension identified by <code>extension</code> , otherwise it returns false. If <code>syclDevice.get_backend() != sycl::backend::opencl</code> an exception with the <code>errc::backend_mismatch</code> error code is thrown.

### C.7.3. Reference counting

Most OpenCL objects are reference counted. The SYCL general programming model doesn't require that native objects are reference counted. However, for convenience, the following function is provided in the `sycl::opencl` namespace.

Reference counting	Description
<pre>template &lt;typename openCL&gt; cl_uint get_reference_count (openCL obj)</pre>	Returns the reference count of the given object

### C.7.4. Errors and limitations

If there is an OpenCL error associated with an exception triggered, then the OpenCL error code can be obtained by the free function `cl_int sycl::opencl::get_error_code(sycl::exception&)`. In the case where there is no OpenCL error associated with the exception triggered, the OpenCL error code will be `CL_SUCCESS`.

### C.7.5. Interoperability with kernel bundles

In [OpenCL](#) any kernel function that is enqueued over an nd-range is represented by a `cl_kernel` and must be compiled and linked via a `cl_program` using `clBuildProgram`, `clCompileProgram` and `clLinkProgram`.

For OpenCL [SYCL backend](#) this detail is abstracted away by [kernel bundles](#) and a `kernel_bundle` object containing all [SYCL kernel functions](#) is retrieved by calling the free function `get_kernel_bundle`.

The OpenCL [SYCL backend](#) specification provides additional free functions which provide convenience functions for constructing kernel bundles from OpenCL specific objects.

```
1 namespace sycl::opencl {
2
3 template <bundle_state State>
4 kernel_bundle<State> create_bundle(const context& ctxt,
5                                   const std::vector<device>& devs,
6                                   const std::vector<cl_program>& clPrograms);
7
8 kernel_bundle<bundle_state::executable>
9 create_bundle(const context& ctxt, const std::vector<device>& devs,
10              const std::vector<cl_kernel>& clKernels);
```

```

11
12 } // namespace sycl::opengl

```

```

1 template <bundle_state State>
2 kernel_bundle<State> create_bundle(const context& ctxt,
3                                   const std::vector<device>& devs,
4                                   const std::vector<cl_program>& clPrograms)

```

1. *Preconditions:* The `context` specified by `ctxt` must be associated with the OpenCL [SYCL backend](#). All devices in `devs` must be associated with `ctxt`. All OpenCL programs in `clPrograms` must be associated with `ctxt`.

*Effects:* Constructs a `kernel bundle` in the specified `bundle_state` from the provided list of OpenCL programs and associated with the `context` specified by `syclContext` by invoking the necessary OpenCL APIs. Follows the same rules as calling `make_kernel_bundle` on a single OpenCL program, except that the rules apply to all OpenCL programs in `clPrograms`. Multiple programs will be linked together into a single one if required by the requested `State`. The constructed `kernel_bundle` will retain all provided OpenCL programs and will also release them on destruction.

*Throws:* An `exception` with the `errc::build` error code if any error is produced by invoking the OpenCL APIs.

```

1 kernel_bundle<bundle_state::executable>
2 create_bundle(const context& ctxt, const std::vector<device>& devs,
3               const std::vector<cl_kernel>& clKernels)

```

1. *Preconditions:* The `context` specified by `ctxt` must be associated with the OpenCL [SYCL backend](#). All devices in `devs` must be associated with `ctxt`. All OpenCL kernels in `clKernels` must be associated with `ctxt`.

*Effects:* Constructs an executable `kernel bundle` from the provided list of OpenCL kernels and associated with the `context` specified by `syclContext` by invoking the necessary OpenCL APIs. `cl_kernel` objects might be associated with different `cl_program` objects, the kernel bundle will encapsulate all of them.

*Throws:* An `exception` with the `errc::build` error code if any error is produced by invoking the OpenCL APIs.

## C.7.6. Interoperability with kernels

A `kernel_bundle` object contains one or multiple OpenCL programs and one or multiple OpenCL kernels. Calling `kernel_bundle::get_kernel` returns a `kernel` object which can be invoked by any of [kernel invocation commands](#) such as `parallel_for` which take a `kernel` but not [SYCL kernel function](#).

Calling `make_kernel` must trigger a call to `clRetainKernel` and the resulting `kernel` object must call `clReleaseKernel` on destruction.

It is also possible to construct a `kernel bundle` from previously created OpenCL `cl_kernel` objects by calling the free function `create_bundle` as described in [Section C.7.5](#).

The kernel arguments for the OpenCL C kernel can either be set prior to creating the `kernel` object or by calling `set_arg` or `set_args` member functions of the `handler` class.

If kernel arguments are set prior to creating the `kernel` object the [SYCL runtime](#) is not responsible for managing the data of these arguments.

## C.7.7. OpenCL kernel conventions and SYCL

OpenCL and SYCL use opposite conventions for the unit stride dimension. SYCL aligns with C++ conventions, which is important to understand from a performance perspective when porting code to SYCL. The unit stride dimension, at least for data, is implicit in the linearization equations in SYCL ([Section 3.11.1](#)) and OpenCL. SYCL aligns with C++ array subscript ordering `arr[a][b][c]`, in that range constructor dimension ordering used to launch a kernel (e.g. `range<3> R{a,b,c}`) and range and ID queries within a kernel, are ordered in the same way as the C++ multi-dimensional subscript operators (unit stride on the right).

When specifying a `range` as the global or local size in a `parallel_for` that invokes an OpenCL interop kernel (through `cl_kernel` interop), the highest dimension of the range in SYCL will map to the lowest dimension within the OpenCL kernel. That statement applies to both an underlying enqueue operation such as `clEnqueueNDRangeKernel` in OpenCL, and also ID and size queries within the OpenCL kernel. For example, a 3D global range specified in SYCL as:

```
range<3> R { r0, r1, r2 };
```

maps to an `clEnqueueNDRangeKernel` `global_work_size` argument of:

```
size_t cl_interop_range[3] = { r2, r1, r0 };
```

Likewise, a 2D global range specified in SYCL as:

```
range<2> R { r0, r1 };
```

maps to an `clEnqueueNDRangeKernel` `global_work_size` argument of:

```
size_t cl_interop_range[2] = { r1, r0 };
```

The mapping of highest dimension in SYCL to lowest dimension in OpenCL applies to all operations where a multi-dimensional construct must be mapped, such as when mapping SYCL explicit memory operations to OpenCL APIs like `clEnqueueCopyBufferRect`.

Work-item and work-group ID and range queries have the same reversed convention for unit stride dimension between SYCL and OpenCL. For example, with three, two, or one dimensional SYCL global ranges, OpenCL and SYCL kernel code queries relate to the range as shown in [Table 162](#). The "SYCL kernel query" column applies for SYCL-defined kernels, and the "OpenCL kernel query" column applies for kernels defined through OpenCL interop.

Table 162. Example range mapping from SYCL enqueued three dimensional global `range` to OpenCL and SYCL queries

SYCL kernel query	OpenCL kernel query	Returned Value
With enqueued 3D SYCL global <code>range</code> of <code>range&lt;3&gt; R{r0,r1,r2}</code>		
<code>nd_item::get_global_range(0) / item::get_range(0)</code>	<code>get_global_size(2)</code>	<code>r0</code>
<code>nd_item::get_global_range(1) / item::get_range(1)</code>	<code>get_global_size(1)</code>	<code>r1</code>
<code>nd_item::get_global_range(2) / item::get_range(2)</code>	<code>get_global_size(0)</code>	<code>r2</code>
<code>nd_item::get_global_id(0) / item::get_id(0)</code>	<code>get_global_id(2)</code>	Value in range <code>0..(r0-1)</code>

SYCL kernel query	OpenCL kernel query	Returned Value
<code>nd_item::get_global_id(1) / item::get_id(1)</code>	<code>get_global_id(1)</code>	Value in range $0..(r1-1)$
<code>nd_item::get_global_id(2) / item::get_id(2)</code>	<code>get_global_id(0)</code>	Value in range $0..(r2-1)$
With enqueued 2D SYCL global <b>range</b> of <b>range&lt;2&gt; R{r0,r1}</b>		
<code>nd_item::get_global_range(0) / item::get_range(0)</code>	<code>get_global_size(1)</code>	<b>r0</b>
<code>nd_item::get_global_range(1) / item::get_range(1)</code>	<code>get_global_size(0)</code>	<b>r1</b>
<code>nd_item::get_global_id(0) / item::get_id(0)</code>	<code>get_global_id(1)</code>	Value in range $0..(r0-1)$
<code>nd_item::get_global_id(1) / item::get_id(1)</code>	<code>get_global_id(0)</code>	Value in range $0..(r1-1)$
With enqueued 1D SYCL global <b>range</b> of <b>range&lt;1&gt; R{r0}</b>		
<code>nd_item::get_global_range(0) / item::get_range(0)</code>	<code>get_global_size(0)</code>	<b>r0</b>
<code>nd_item::get_global_id(0) / item::get_id(0)</code>	<code>get_global_id(0)</code>	Value in range $0..(r0-1)$

## C.7.8. Data types

The OpenCL C language standard [Section 6.11](#) defines its own built-in scalar data types, and these have additional requirements in terms of size and signedness on top of what is guaranteed by ISO C++. For the purpose of interoperability and portability, SYCL defines a set of aliases to C++ types within the `sycl::opencl` namespace using the `cl_` prefix. These aliases are described in [Table 163](#).

Table 163. Scalar data type aliases supported by SYCL OpenCL backend

Scalar data type alias	Description
<code>cl_bool</code>	Alias to a conditional data type which can be either true or false. The value true expands to the integer constant 1 and the value false expands to the integer constant 0.
<code>cl_char</code>	Alias to a signed 8-bit integer, as defined by the C++ core language.
<code>cl_uchar</code>	Alias to an unsigned 8-bit integer, as defined by the C++ core language.
<code>cl_short</code>	Alias to a signed 16-bit integer, as defined by the C++ core language.
<code>cl_ushort</code>	Alias to an unsigned 16-bit integer, as defined by the C++ core language.
<code>cl_int</code>	Alias to a signed 32-bit integer, as defined by the C++ core language.
<code>cl_uint</code>	Alias to an unsigned 32-bit integer, as defined by the C++ core language.

Scalar data type alias	Description
<code>cl_long</code>	Alias to a signed 64-bit integer, as defined by the C++ core language.
<code>cl_ulong</code>	Alias to an unsigned 64-bit integer, as defined by the C++ core language.
<code>cl_float</code>	Alias to a 32-bit floating-point. The float data type must conform to the IEEE 754 single precision storage format.
<code>cl_double</code>	Alias to a 64-bit floating-point. The double data type must conform to the IEEE 754 double precision storage format.
<code>cl_half</code>	Alias to a 16-bit floating-point. The half data type must conform to the IEEE 754-2008 half precision storage format. Kernels using this type are only supported on devices that have <code>aspect::fp16</code> , as described in <a href="#">Section 5.7</a> .

## C.8. Preprocessor directives and macros

- `SYCL_BACKEND_OPENCL` substitutes to `1` if the OpenCL [SYCL backend](#) is active while building the SYCL application.

### C.8.1. Offline linking with OpenCL C libraries

SYCL supports linking [SYCL kernel functions](#) with OpenCL C libraries during offline compilation or during online compilation by the [SYCL runtime](#) within a SYCL application.

Linking with OpenCL C kernel functions offline is an optional feature and is unspecified. Linking with OpenCL C kernel functions online is performed via the `sycl::make_kernel_bundle` or `sycl::opencl::create_bundle` functions, which both provide a mechanism to create an instance of a SYCL `kernel_bundle` from a `cl_program`.

OpenCL C functions that are linked with, using either offline or online compilation, must be declared as `extern "C"` function declarations. The function parameters of these function declarations must be defined as the OpenCL C interoperability aliases; `pointer` of the `multi_ptr` class template, and scalar data type aliases described in [Table 163](#).

## C.9. SYCL support of non-core OpenCL features

In addition to the OpenCL core features, SYCL also provides support for OpenCL extensions which provide features in OpenCL via `khr` extensions.

Some extensions are natively supported within the SYCL interface, however some can only be used via the OpenCL interoperability interface. The SYCL interface required for native extensions must be available. However if the respective extension is not supported by the executing SYCL `device`, the [SYCL runtime](#) must throw an `exception` with the `errc::feature_not_supported` or `errc::kernel_not_supported` error codes.

The OpenCL backend exposes some `khr` extensions to SYCL applications through the `sycl::aspect` enumerated type. Therefore, applications can query for the existence of these `khr` extensions by calling the `device::has()` or `platform::has()` member functions.

All OpenCL extensions are available through the OpenCL interoperability interface, but some can also be used through core SYCL APIs. [Table 164](#) shows which these are. [Table 164](#) also shows the mapping from each OpenCL extension name to its associated SYCL device `aspect` when one is available.

Table 164. SYCL support for OpenCL 1.2 extensions

SYCL Aspect	OpenCL Extension	Core SYCL API
<code>aspect::atomic64</code>	<code>cl_khr_int64_base_atomics</code>	Yes
<code>aspect::atomic64</code>	<code>cl_khr_int64_extended_atomics</code>	Yes
<code>aspect::fp16</code>	<code>cl_khr_fp16</code>	Yes
-	<code>cl_khr_3d_image_writes</code>	Yes
-	<code>cl_khr_gl_sharing</code>	No
-	<code>cl_apple_gl_sharing</code>	No
-	<code>cl_khr_d3d10_sharing</code>	No
-	<code>cl_khr_d3d11_sharing</code>	No
-	<code>cl_khr_dx9_media_sharing</code>	No

### C.9.1. Half precision floating-point

The half scalar data type: `half` and the half vector data types: `half1`, `half2`, `half3`, `half4`, `half8` and `half16` must be available at compile-time. However a kernel using these types is only supported on devices that have `aspect::fp16`, as described in [Section 5.7](#).

The conversion rules for half precision types follow the same rules as in the OpenCL 1.2 extensions specification [par. 9.5.1](#).

The math functions for half precision types follow the same rules as in the OpenCL 1.2 extensions specification [par. 9.5.2](#), [9.5.3](#), [9.5.4](#), [9.5.5](#). The allowed error in ULP(Unit in the Last Place) is less than 8192, corresponding to [Table 6.9 of the OpenCL 1.2 specification](#).

### C.9.2. Writing to 3D image memory objects

The `unsampled_image_accessor` class in SYCL supports member functions for writing 3D image memory objects, but this functionality is only allowed on a device if the extension `cl_khr_3d_image_writes` is supported on that [device](#).

### C.9.3. Interoperability with OpenGL

Interoperability between SYCL and OpenGL is not directly provided by the SYCL interface, however can be achieved via the SYCL OpenCL interoperability interface.

## C.10. Correspondence of some OpenCL features to SYCL

This section describes the correspondence between some OpenCL features and features in the [core SYCL specification](#) that provide similar functionality. All content in this section is non-normative.

### C.10.1. Work-item functions

The OpenCL 1.2 specification document [ch. 6.12.1 in Table 6.7](#) defines work-item functions that tell various information about the currently executing work-item in an OpenCL kernel. SYCL provides equivalent functionality through the item and group classes that are defined in [Section 4.9.1.4](#), [Section 4.9.1.5](#) and [Section 4.9.1.7](#).

### C.10.2. Vector data load and store functions

The functionality from the OpenCL functions as defined in the OpenCL 1.2 specification document [par.](#)



6.12.7 is available in SYCL through the `vec` class in [Section 4.14.2](#).

### C.10.3. Synchronization functions

In SYCL the OpenCL synchronization functions are available through the `nd_item` class ([Section 4.9.1.5](#)), as they are applied to work-items for local or global address spaces. Please see [Table 97](#).

### C.10.4. `printf` function

The functionality of the `printf` function is covered by the `stream` class ([Section 4.16](#)), which has the capability to print to standard output all of the SYCL classes and primitives, and covers the capabilities defined in the OpenCL 1.2 specification document [par. 6.12.13](#).

## C.11. Precision of built-in math functions

When the SYCL built-in functions defined in sections [Section 4.17.4](#) through [Section 4.17.10](#) are called from a kernel running on the OpenCL backend, their precision is the same as the corresponding OpenCL functions as specified for the OpenCL profile. See ch. 6.12.2 through ch. 6.12.6 of the OpenCL 1.2 specification for the definition of these corresponding OpenCL functions.



# Appendix D: What has changed from previous versions

## D.1. What has changed from SYCL 1.2.1 to SYCL 2020

The SYCL runtime moved from namespace `cl::sycl` provided by `#include <CL/sycl.hpp>` to namespace `sycl` provided by `#include <sycl/sycl.hpp>` as explained in [Section 4.3](#). The old header file is still available for compatibility with SYCL 1.2.1 applications.

The SYCL specification is now based on the core language of C++17, as described in [Section 3.9.1](#). Features of C++17 are now enabled within the specification, such as deduction guides for class template argument deduction.

Naming of lambda expressions passed to kernel invocations is now optional.

Changes to buffers, images and accessors:

- The `image` class has been removed. There are now new classes `unsampled_image` and `sampled_image` which represent sampled and unsampled images. The `sampler` class has been removed and replaced with the new `image_sampler` structure.
- Support for image arrays has been removed.
- The type name `access::target` has been deprecated and replaced with the type `target`.
- The type name `access::mode` has been deprecated and replaced with the type `access_mode`.
- The name of the `accessor` target `target::global_buffer` has been deprecated and replaced with `target::device`.
- Support for the `accessor` target `target::host_buffer` has been deprecated. There is now a new accessor class `host_accessor` which provides equivalent functionality.
- The `buffer` member functions which return an `accessor` of type `target::host_buffer` have been deprecated. A new member function `get_host_access()` has been added which returns a `host_accessor`.
- The `buffer` class has a new variadic overload of the `get_access()` member function which allows construction of an `accessor` with various parameters.
- Support for the `accessor` target `target::local` has been deprecated. There is now a new accessor class `local_accessor` which provides equivalent functionality.
- Support for the `accessor` targets `target::image` and `target::host_image` have been removed. There are now new accessor classes for sampled and unsampled images: `sampled_image_accessor`, `host_sampled_image_accessor`, `unsampled_image_accessor` and `host_unsampled_image_accessor`.
- A new `accessor` target `target::host_task` has been added, which allows access to a `buffer` from a `host task`.
- Support for the `accessor` modes `access_mode::discard_write` and `access_mode::discard_read_write` has been deprecated. Accessors can now be constructed with a property list, and the new property `property::no_init` provides equivalent functionality.
- Support for the `accessor` mode `access_mode::atomic` and the member functions that return an instance of the `atomic` class have been deprecated in favor of using the new `atomic_ref` class instead.
- Support for the `accessor` template parameter `isPlaceholder` has been deprecated, and the value of this parameter no longer has any bearing on whether the accessor is a placeholder. The enumerated type `access::placeholder` is also deprecated. A placeholder accessor can now be constructed by calling the appropriate constructor, without regard to the template parameter.
- The return type of `accessor::is_placeholder()` is no longer `constexpr`.

- The member function `handler::require()` may now be called on any `accessor` with target `target::device`, `target::constant_buffer` or `target::host_task`, regardless of whether it is a placeholder.
- New `accessor` constructors have been added which take a type tag parameter, which allows the class template parameters to be inferred via C++ class template argument deduction (CTAD).
- The `buffer` member function `get_access()` now has a default value for the `target` template parameter, so it is no longer necessary to provide any template parameters in order to get a `access_mode::read_write` accessor.
- The `accessor` template parameters `Dimensions` and `AccessMode` now have default values, so the only required template parameter is `DataT`. Moreover, the default access mode is either `access_mode::read_write` or `access_mode::read`, depending on the constness of the `DataT` type. This makes it possible to declare a read-only accessor by simply using a `const` qualified type.
- Implicit conversions have been added between the two forms of read-only `accessor` (one form has `const DataT` and `access_mode::read` and the other has non-const `DataT` and `access_mode::read`). There is also an implicit conversion from a read-write `accessor` to either of the read-only forms.
- Member functions of `accessor` which return a reference to an element have been changed to return a `const` reference for read-only accessors. The `get_pointer()` member function has also been changed to return a `const` pointer for read-only accessors. The `value_type` and `reference` member types of `accessor` have been changed to be `const` types for read-only accessors.
- The `accessor` class now meets the C++ requirement of `ReversibleContainer`. This includes (but is not limited to) returning `begin` and `end` iterators, specifying a default constructible accessor that can be passed to a kernel but not dereferenced, and making them equality comparable.
- Many of the `accessor` member functions have been marked `noexcept`.
- A `ranged accessor` is no longer allowed to read elements that are outside of its range; attempting to do so produces undefined behavior.
- The semantics of the subscript operator have been changed for a `ranged accessor` which has an offset. Calling `operator[]()` now returns a reference to the first element in the range, rather than a reference to the first element in the underlying buffer.
- The behavior of buffers and accessors with a zero-sized range has been clarified.

Constant memory no longer appears in the SYCL device memory model in SYCL 2020.

The C++ attributes that decorate kernels are now better described, and their position has changed so that they are applied directly to the kernel function. (Previously, they were applied to a device function that the kernel calls, and the implementation needed to propagate the information up to the enclosing kernel.) The old C++ attribute form is no longer included in the SYCL specification.

Changes to the built-in functions specified in [Section 4.17](#):

- The specification no longer uses pseudo "generic type names" to describe these functions, and it now lists the exact synopsis for each function.
- The return type of the integer `abs` and `abs_diff` functions has changed. The return type is now the same as the input type, matching the C++ `std::abs` function.
- The geometric functions specified in [Section 4.17.9](#) now support the `half` data type.
- The `ctz` function was added to [Section 4.17.7](#).
- The specification of `clz` was clarified for the case when the input is zero.

The classes `vector_class`, `string_class`, `function_class`, `mutex_class`, `shared_ptr_class`, `weak_ptr_class`, `hash_class` and `exception_ptr_class` have been removed from the API and the standard classes `std::vector`, `std::string`, `std::function`, `std::mutex`, `std::shared_ptr`, `std::weak_ptr`, `std::hash` and `std::exception_ptr` are used instead.

The specific `sycl::buffer` API taking `std::unique_ptr` has been removed. The behavior is the same as in

SYCL 1.2.1 but with a simplified API. Since there is still the API taking `std::shared_ptr` and there is an implicit conversion from a `std::unique_ptr` prvalue to a `std::shared_ptr`, the API can still be used as before with a `std::unique_ptr` to give away memory ownership.

Offsets to `parallel_for`, `nd_range`, `nd_item` and `item` classes have been deprecated. As such, the parallel iteration spaces all begin at  $(0,0,0)$  and developers are now required to handle any offset arithmetic themselves. The behavior of `nd_item.get_global_linear_id()` and `nd_item.get_local_linear_id()` has been clarified accordingly.

Unified Shared Memory (USM), in [Section 4.8](#), has been added as a pointer-based strategy for data management. It defines several types of allocations with various accessibility rules for host and devices. USM is meant to complement buffers, not replace them.

The `queue` class received a new `property` that requires in-order semantics for a queue where operations are executed in the order in which they are submitted.

The `queue` class received several new member functions to invoke kernels directly on a queue objects instead of inside a command group handler in the `submit` member function.

The `queue` constructor overloads that accept both a `context` and a `device` parameter have been broadened to allow the device to be either a device that is in the context or a `descendent device` of a device that is in the context.

The `program` class has been removed and replaced with a new class `kernel_bundle`, which provides similar functionality in a type-safe and thread-safe way. The `kernel` class has changed, and some member functions have been removed.

Support has been added for `specialization-constants`, which allow a SYCL kernel function to use constant variables whose values aren't known until the kernel is invoked. A SYCL kernel function can now take an optional parameter of type `kernel_handler`, which allows the kernel to read the values of `specialization-constants`.

The constructors for SYCL `context` and `queue` are made `explicit` to prevent ambiguities in the selected constructor resulting from implicit type conversion.

The requirement for C++ standard layout for data shared between host and devices has been relaxed. SYCL now requires data shared between host and devices to be `device copyable` as defined [Section 3.13.1](#).

The concept of a `group` of `work-items` was generalized to include `work-groups` and `sub-groups`. A `work-group` is represented by the `sycl::group` class as in SYCL 1.2.1, and a `sub-group` is represented by the new `sycl::sub_group` class.

The `host_task` member function for the `queue` has been introduced for en-queueing `host tasks` on a `queue` to schedule the SYCL runtime to invoke native C++ functions, conforming to the SYCL memory model. `Host-tasks` also support interoperability with the native SYCL backend objects associated at that point in the DAG using the optional `interop_handle` class.

A library of algorithms based on the C++17 algorithms library was introduced in [Section 4.17.3](#). These algorithms provide a simple way for developers to apply common parallel algorithms using the work-items of a group.

The definition of the `sycl::group` class was modified to support the new group functions in [Section 4.17.2](#). New member types and variables were added to enable generic programming, and member functions were updated to encapsulate all functionality tied to `work-groups` in the `sycl::group` class. See [Table 99](#) for details.

The `barrier` and `mem_fence` member functions of the `nd_item` class have been removed. The `barrier` member function has been replaced by the `group_barrier()` function, which can be used to block work-items

in either `work-groups` or `sub-groups` until all work-items in the group arrive at the barrier. The `mem_fence` member function has been replaced by the `atomic_fence` function, which is more closely aligned with `std::atomic_thread_fence` and offers control over memory ordering and scope.

Changes in the SYCL `vec` class described in [Section 4.14.2](#):

- `operator[]` was added;
- unary `operator+()` and `operator-()` were added;

The device selection now relies on a simpler API based on ranking functions used as `device selectors` described in [Section 4.6.1.1](#).

A new device selector utility has been added to [Section 4.6.1.1](#), the `aspect_selector`, which returns a selector object that only selects devices that have all the requested aspects.

The device query `info::fp_config::correctly_rounded_divide_sqrt` has been deprecated.

A new reduction library consisting of the `reduction` function and `reducer` class was introduced to simplify the expression of variables with `reduction` semantics in SYCL kernels. See [Section 4.9.2](#).

The `atomic` class from SYCL 1.2.1 was deprecated in favor of a new `atomic_ref` interface.

The SYCL exception class hierarchy has been condensed into a single exception type: `exception`. `exception` now derives from `std::exception`. The variety of errors are now provided via error codes, which aligns with the C++ error code mechanism.

The new error code mechanism now also generalizes the previous `get_cl_code` interface to provide a generic interface way for querying backend-specific error codes.

Default asynchronous error handling behavior is now defined, so that asynchronous errors will cause abnormal program termination even if a user-defined asynchronous handler function is not defined. This prevents asynchronous errors from being silently lost during early stages of application development.

Kernel invocation functions, such as `parallel_for`, now take kernel functions by `const` reference. Kernel functions must now have a `const`-qualified `operator()`, and are allowed to be copied zero or more times by an implementation. These clarifications allow implementations to have flexibility for specific devices, and define what users should expect with kernel functors. Specifically, kernel functors can not be marked as `mutable`, and sharing of data between work-items should not be attempted through state stored within a kernel functor.

A new concept called device `aspects` has been added, which tells the set of optional features a device supports. This new mechanism replaces the `has_extension()` function and some uses of `get_info()`.

There is a new [Chapter 6](#) which describes how extensions to the SYCL language can be added by vendors and by the Khronos Group.

A `queue` constructor has been added that takes both a `device` and `context`, to simplify interfacing with libraries.

The `parallel_for` interface has been simplified in some forms to accept a braced initializer list in place of a `range`, and to always take `item` arguments. Kernel invocation functions have also been modified to accept generic lambda expressions. Implicit conversions from one-dimensional `item` and one-dimensional `id` to scalar types have been defined. All of these modifications lead to simpler SYCL code in common use cases.

The behaviour of executing a kernel over a `range` or `nd_range` with index space of zero has been clarified.

Some device-specific queries have been renamed to more clearly be “device-specific kernel” `get_info`

queries (`info::kernel_device_specific`) instead of “work-group” (`get_workgroup_info`) and sub-group (`get_sub_group_info`) queries.

A new math array type `marray` has been defined to begin disambiguation of the multiple possible interpretations of how `sycl::vec` should be interpreted and implemented.

Changes in SYCL address spaces:

- the address space meaning has been significantly improved;
- the generic address space was introduced;
- the constant address space was deprecated;
- behavior of unannotated pointer/reference (raw pointer/reference) is now dependent on the compilation mode. The compiler can either interpret unannotated pointer/reference as addressing the generic address space or to be deduced;
- some ambiguities in the address space deduction were clarified. Notably that deduced type does not affect the user-provided type.

Changes in `multi_ptr` interface:

- addition of `access::address_space::generic_space` to represent the generic address space;
- deprecation of `access::address_space::constant_space`;
- an extra template parameter to allow to select a flavor of the `multi_ptr` interface. There are now 3 different interfaces:
  - interface exposing undecorated types. Returned pointer and reference are not annotated by an address space;
  - interface exposing decorated types. Returned pointer and reference are annotated by an address space;
- deprecation of the 1.2.1 interface;
- deprecation of `constant_ptr`;
- `global_ptr`, `local_ptr` and `private_ptr` alias take the new extra parameter;
- addition of the `address_space_cast` free function to cast undecorated pointer to `multi_ptr`;
- addition of construction/conversion operator for the generic address space;
- removal of the constructor and assignment operator taking an unannotated pointer;
- implicit conversion to a pointer is now deprecated. `get` should be used instead;
- the return type of the member function `get` now depends on the selected interface.
- addition of the member function `get_raw` which returns the underlying pointer as an unannotated pointer;
- addition of the member function `get_decorated` which returns the underlying pointer as an annotated pointer;
- addition of the subscript operator providing random access.

The `cl::sycl::byte` has been deprecated and now the C++17 `std::byte` should be used instead.

A SYCL implementation is no longer required to provide a host device. Instead, an implementation is only required to provide at least one device. Implementations are still allowed to provide devices that are implemented on the host, but it is no longer required. The specification no longer defines any special semantics for a “host device” and APIs specific to the host device have been removed.

The default constructors for the `device` and `platform` classes have been changed to construct a copy of the default device and a copy of the platform containing the default device. Previously, they returned a copy

of the host device and a copy of the platform containing the host device. The default constructor for the `event` class has also been changed to construct an event that comes from a default-constructed `queue`. Previously, it constructed an event that used the host backend.

Explicit copy functions of the handler class have also been introduced to the queue class as shortcuts for the handler ones. This is enabled by the improved placeholder accessors to help reduce code verbosity in certain cases because the shortcut functions implicitly create a command group and call `handler::require`.

Information query descriptors have been changed to structures under namespaces named accordingly. `param_traits` has been removed and the return type of an information query is now contained in the descriptor. The `sycl::info::device::max_work_item_sizes` is now a template that takes a dimension parameter corresponding to the number of dimensions of the work-item size maxima.

Changes to retrieving size information:

- all `get_size()` member functions have been deprecated and replaced with `byte_size()`, which is marked `noexcept`;
- all `get_count()` member functions have been deprecated and replaced with `size()`, which is marked `noexcept`;
- in the `vec` class the functions `byte_size()` and `size()` are now static member functions;
- in the `stream` class `get_size()` has been deprecated in favor of `size()`, whereas `stream::byte_size()` is not available;
- accessors for sampled and unsampled images only define `size()` and not `byte_size()`.

The device descriptors `info::device::max_constant_buffer_size` and `info::device::max_constant_args` are deprecated in SYCL 2020.

The `buffer_allocator` is now templated on the data type and follows the C++ named requirement `Allocator`.

The SYCL `id` and `range` have now unary `+` and `-` operations, prefix `++` and `--` operations, postfix `++` and `--` operations which were forgotten in SYCL 1.2.1.

In SYCL 1.2.1, the `handler::copy()` overload with two `accessor` parameters did not clearly specify which accessor's size determines the amount of memory that is copied. The spec now clarifies that the `src` accessor's size is used.

Any code considered as a `discarded statement` or as a `manifestly constant-evaluated expression or conversion` by the C++ standard is now also accepted in SYCL device function.



# Appendix E: References

International Organization for Standardization (ISO). “Programming Languages — C++”. ISO/IEC 14882:2017, 2017.

International Organization for Standardization (ISO). Accepted resolution to C++ Standard Core Language Defect Report DR2325. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0593r6.html>.

Khronos OpenCL Working Group. *The OpenCL Extension Specification*, Version 1.2.25 (2/13/18). <http://www.khronos.org/registry/cl/specs/opencl-1.2-extensions.pdf>.

Khronos OpenCL Working Group. *The OpenCL Specification*, Version 1.2.19 (11/14/12). <https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf>.

Khronos OpenCL Working Group. *The OpenCL Specification*, Version 2.0.29 (July 21, 2015). <https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf>.

International Organization for Standardization (ISO). " Programming Languages — C++, Langages de programmation — C++ ", International Standard ISO/IEC 14882:2020(E), Sixth edition 2020-12, 2020.

# Appendix F: Optional extensions

Each of the optional extensions in this appendix has been approved by the SYCL working group. These extensions may be promoted to core features in future versions of the SYCL specification, but their design is subject to change.

## F.1. `sycl_khr_default_context`

When a `queue` object is constructed without passing an explicit `context` object, the queue uses the platform's default context. This extension adds a new query function to retrieve this default context from a `platform` object.

### F.1.1. Dependencies

This extension has no dependencies on other extensions.

### F.1.2. Feature test macro

An implementation supporting this extension must predefine the macro `SYCL_KHR_DEFAULT_CONTEXT` to one of the values defined in the table below.

Value	Description
1	Initial version of this extension.

### F.1.3. Extensions to the platform class

This extension adds the following new member functions to the `platform` class.

```
namespace sycl {
class platform {
    context khr_get_default_context() const;
    // ...
};
}
```

#### F.1.3.1. Member functions

`platform::khr_get_default_context`

```
context khr_get_default_context() const
```

*Returns:* A copy of the default context object for this platform. The default context contains all of the `root devices` that are associated with this platform.



# Glossary

## accessor

An accessor is a class which allows a [command](#) to access data managed by a [buffer](#) or [image](#) class or allows a [SYCL kernel function](#) to access local memory on a [device](#). Accessors are also used to express the dependencies among the different [command groups](#). For the full description please refer to [Section 4.7.6](#)

## application scope

The application scope starts with the construction first [SYCL runtime](#) class object and finishes with the destruction of the last one. Application refers to the C++ [SYCL application](#) and not the [SYCL runtime](#).

## aspect

A characteristic of a [device](#) which determines whether it supports some optional feature. Aspects are always boolean, so a [device](#) either has or does not have an aspect.

## asynchronous error

A SYCL asynchronous error is an error occurring after the host API call invoking the error causing action has returned, such that the error cannot be thrown as a typical C++ exception from a host API call. Such errors are typically generated from device kernel invocations which are executed when SYCL task graph dependencies are satisfied, which occur asynchronously from host code execution. For the full description and associated asynchronous error handling mechanisms, please refer to [Section 4.13](#).

## async\_handler

An asynchronous error handler object is a function class instance providing necessary code for handling all the asynchronous errors triggered from the execution of command groups on a queue, within a context or an associated event. For the full description please refer to [Section 4.13.2](#).

## barrier

A barrier may refer to either a [command queue barrier](#) used for host-device coordination, or a [group barrier](#) used to coordinate work-items in a kernel.

## blocking accessor

A blocking accessor is an [accessor](#) which provides immediate access and continues to provide access until it is destroyed. For the full description please refer to [Section 4.7.6](#)

## buffer

The buffer class manages data for the SYCL C++ host application and the SYCL device kernels. The buffer class may acquire ownership of some host pointers passed to its constructors according to the constructor kind.

The buffer class, together with the accessor class, is responsible for tracking memory transfers and guaranteeing data consistency among the different kernels. The [SYCL runtime](#) manages the memory allocations on both the host and the [device](#) within the lifetime of the buffer object. For the full description please refer to [Section 4.7.2](#).

## bundle state

A SYCL bundle state represents the state of a [kernel bundle](#) and therefore its capabilities in the SYCL programming API. Possible states are [input](#), [object](#) or [executable](#).

## command

A request to execute work that is submitted to a [queue](#) such as the invocation of a [SYCL kernel function](#), the invocation of a [host task](#) or an asynchronous copy.

**command group**

In SYCL, the operations required to process data on a [device](#) are represented using a [command group function object](#). Each [command group function object](#) is given a unique [command group handler](#) object to perform all the necessary work required to correctly process data on a [device](#) using a kernel. In this way, the group of commands for transferring and processing data is enqueued as a command group on a [device](#) for execution. A command group is submitted atomically to a SYCL queue.

**command group function object**

A type which is callable with `operator()` that takes a reference to a [command group handler](#), that defines a [command group](#) which can be submitted by a [queue](#). The function object can be a named type, lambda expression or `std::function`.

**command group handler**

The command group handler class provides the interface for the commands that can be executed inside the [command group scope](#). It is provided as a scoped object to all of the data access requests within the command group scope. For the full description please refer to [Section 4.9.4](#).

**command group scope**

The command group scope is the function scope defined by the [command group function object](#). The command group [command group handler](#) object lifetime is restricted to the command group scope. For more details see [Section 4.9.3](#).

**command queue barrier**

The `sycl::queue::wait()` and `sycl::queue::wait_and_throw()` functions block the calling thread until the execution of a [command group function object](#) completes.

**constant memory**

A region of memory that remains constant during the execution of a kernel. The [SYCL runtime](#) allocates and initializes memory objects placed into constant memory.

**context**

A [context](#) represents the runtime data structures and state required by a [SYCL backend](#) API to interact with a group of [devices](#) associated with a [platform](#). The context is defined as the `sycl::context` class, for further details please see [Section 4.6.3](#).

**control flow**

When all [work-items](#) in a [group](#) are executing the same sequence of statements, they are said to be executing under *converged* control flow. Control flow *diverges* when different work-items in a group execute a different sequence of statements, typically as a result of evaluating conditions differently (e.g. in selection statements or loops).

**core SYCL specification**

The text of the SYCL language specification (this document), excluding the text of any backend specifications and excluding the text for any extensions.

**descendent device**

The descendent devices of device *D* include all of the sub-devices of *D*, all of the sub-devices of those devices, etc.

**device**

A SYCL device is an abstraction of a piece of hardware that can execute [SYCL kernels](#).

**device compiler**

A SYCL device compiler is a compiler that produces [device](#) binaries from a valid [SYCL application](#). For the full description please refer to [Chapter 5](#).

**device copyable**

Data that is shared between the host and the devices must generally have a type that abides by the restrictions listed in [Section 3.13.1](#) for a device copyable type.

**device function**

A device function is any function in a [SYCL application](#) that can be run on a [device](#). This includes [SYCL kernel functions](#) and, recursively, functions they call.

**device image**

A device image is a representation of one or more [kernels](#) in an implementation-defined format. A device image could be a compiled version of the kernels in an intermediate language representation which needs to be translated at runtime into a form that can be invoked on a [device](#), it could be a compiled version of the kernels in a native code format that is ready to be invoked without further translation, or it could be a source code representation which needs to be compiled before it can be invoked. Other representations are possible too.

**device selector**

A way to select a device used in various places. This is a callable object taking a [device](#) reference and returning an integer rank. One of the device with the highest non-negative value is selected. See [Section 4.6.1.1](#) for more details.

**discarded statement**

ISO C++ [stmt.if] describes a discarded statement as the branch statement of an `if constexpr` which is not instantiated because of the boolean condition. For more context, see [Section 5.4](#).

**event**

A SYCL object that represents the status of an operation that is being executed by the SYCL runtime.

**executable**

A state which a [kernel bundle](#) can be in, representing [SYCL kernel functions](#) as an executable.

**generic memory**

Generic memory is a virtual memory region which can represent [global memory](#), [local memory](#) and [private memory](#) region.

**global id**

As in OpenCL, a global ID is used to uniquely identify a [work-item](#) and is derived from the number of global [work-items](#) specified when executing a kernel. A global ID is a one, two or three-dimensional value that starts at 0 per dimension.

**global memory**

Global memory is a memory region accessible to all [work-items](#) executing on a [device](#).

**group**

A group of work-items within the index space of a SYCL kernel execution, such as a [work-group](#) or [sub-group](#).

**group barrier**

A coordination mechanism for all [work-items](#) of a group. See the definition of the `group_barrier` function.

**h-item**

A unique identifier representing a single [work-item](#) within the index space of a SYCL kernel hierarchical execution. Can be one, two or three dimensional. In the SYCL interface a [h-item](#) is represented by the `h_item` class (see [Section 4.9.1.6](#)).

**host**

Host is the system that executes the C++ application including the SYCL API.

**host pointer**

A pointer to memory on the host. Cannot be accessed directly from a [device](#).

**host task**

A [command](#) which invokes a native C++ callable, scheduled conforming to SYCL dependency rules.

**host task command**

A type of command that can be used inside a [command group](#) in order to schedule a native C++ function.

**id**

It is a unique identifier of an item in an index space. It can be one, two or three dimensional index space, since the SYCL kernel execution model is an [nd-range](#). It is one of the index space classes. For the full description please refer to [Section 4.9.1.3](#).

**image**

Images in SYCL, like buffers, are abstractions of multidimensional structured arrays. Image can refer to [unsampled\\_image](#) and [sampled\\_image](#). For the full description please refer to [Section 4.7.3](#).

**implementation-defined**

Behavior that is explicitly allowed to vary between conforming implementations of SYCL. A SYCL implementer is required to document the implementation-defined behavior.

**index space classes**

Like in OpenCL, the kernel execution model defines an [nd-range](#) index space. The [SYCL runtime](#) class that defines an [nd-range](#) is the [sycl::nd\\_range](#), which takes as input the sizes of global and local work-items, represented using the [sycl::range](#) class. The kernel library classes for indexing in the defined [nd-range](#) are the following classes:

- [sycl::id](#): The basic index class representing an [id](#);
- [sycl::item](#): The [item](#) index class that contains the [global id](#) and [local id](#);
- [sycl::nd\\_item](#): The [nd-item](#) index class that contains the [global id](#), [local id](#) and the [work-group id](#);
- [sycl::group](#): The [group](#) class that contains the [work-group id](#) and the member functions on a [work-group](#).

**input**

A state which a [kernel bundle](#) can be in, representing [SYCL kernel functions](#) as a source or intermediate representation

**item**

An item id is an interface used to retrieve the [global id](#), [work-group id](#) and [local id](#). For further details see [Section 4.9.1.4](#).

**kernel**

A kernel represents a [SYCL kernel function](#) that has been compiled for a device, including all of the [device functions](#) it calls. A kernel is implicitly created when a [SYCL kernel function](#) is submitted to a device via a [kernel invocation command](#). However, a kernel can also be created manually by pre-compiling a [kernel bundle](#) (see [Section 4.11](#)).

**kernel bundle**

A kernel bundle is a collection of [device images](#) that are associated with the same [context](#) and with a set of [devices](#). Kernel bundles have one of three states: [input](#), [object](#) or [executable](#). Kernel bundles in

the executable state are ready to be invoked on a device, whereas bundles in the other states need to be translated into the executable state before they can be invoked.

**kernel handler**

A representation of a [SYCL kernel function](#) being invoked that is available to the [kernel scope](#).

**kernel invocation command**

A type of command that can be used inside a [command group](#) in order to schedule a [SYCL kernel function](#), includes [single\\_task](#), all variants of [parallel\\_for](#) and [parallel\\_for\\_workgroup](#).

**kernel name**

A kernel name is a class type that is used to assign a name to the kernel function, used to link the host system with the kernel object output by the device compiler. For details on naming kernels please see [Section 5.2](#).

**kernel scope**

The function scope of the [operator\(\)](#) on a [SYCL kernel function](#). Note that any function or member function called from the kernel is also compiled in kernel scope. The kernel scope allows C++ language extensions as well as restrictions to reflect the capabilities of devices. The extensions and restrictions are defined in the SYCL device compiler specification.

**local id**

A unique identifier of a [work-item](#) among other work-items of a [work-group](#).

**local memory**

Local memory is a memory region associated with a [work-group](#) and accessible only by [work-items](#) in that [work-group](#).

**manifestly constant-evaluated expression or conversion**

ISO C++ [expr.const] describes manifestly constant-evaluated expression or conversion like constant expressions, condition of an if constexpr, an immediate invocation, used in template parameters, in constant initialization, etc. This is evaluated at compile-time by the compiler. For more context, see [Section 5.4](#).

**mem-fence**

A memory fence provides control over re-ordering of memory load and store operations when coupled with an atomic operation. See the definition of the [sycl::atomic\\_fence](#) function.

**native backend object**

An opaque object defined by a specific backend that represents a high-level SYCL object on said backend. There is no guarantee of having native backend objects for all SYCL types.

**native-specialization constant**

A [specialization constant](#) in a device image whose value can be used by an online compiler as an immediate value during the compilation.

**nd-item**

A unique identifier representing a single [work-item](#) and [work-group](#) within the index space of a SYCL kernel execution. Can be one, two or three dimensional. In the SYCL interface an [nd-item](#) is represented by the [nd\\_item](#) class (see [Section 4.9.1.5](#)).

**nd-range**

A representation of the index space of a SYCL kernel execution, the distribution of [work-items](#) within into [work-groups](#). Contains a [range](#) specifying the number of global [work-items](#), a [range](#) specifying the number of local [work-items](#) and a [id](#) specifying the global offset. Can be one, two or three dimensional. The minimum size of [range](#) within the [nd-range](#) is 0 per dimension; where any dimension is

set to zero, the index space in all dimensions will be zero. In the SYCL interface an [nd-range](#) is represented by the `nd_range` class (see [Section 4.9.1.2](#)).

### object

A state which a [kernel bundle](#) can be in, representing [SYCL kernel functions](#) as a non-executable object.

### platform

A collection of [devices](#) managed by a single [backend](#).

### private memory

A region of memory private to a [work-item](#). Variables defined in one work-item's private memory are not visible to another work-item. The `sycl::private_memory` class provides access to the work-item's private memory for the hierarchical API as it is described in [Section 4.9.4.2.3](#).

### queue

A SYCL command queue is an object that holds command groups to be executed on a SYCL [device](#). SYCL provides a heterogeneous platform integration using device queue, which is the minimum requirement for a SYCL application to run on a SYCL [device](#). For the full description please refer to [Section 4.6.5](#).

### range

A representation of a number of [work-items](#) or [work-groups](#) within the index space of a SYCL kernel execution. Can be one, two or three dimensional. In the SYCL interface a [range](#) is represented by the `range` class (see [Section 4.9.1.1](#)).

### ranged accessor

A ranged accessor is a host or buffer [accessor](#) that was constructed with a non-zero offset into the data buffer or with an access range smaller than the range of the data buffer, or both. Please refer to [Section 4.7.6.8](#) for more info.

### reduction

An operation that produces a single value by combining multiple values in an unspecified order using a binary operator. If the operator is non-associative or non-commutative, the behavior of a reduction may be non-deterministic.

### root device

A device that is not a sub-device. The function `device::get_devices()` returns a vector of all the root devices.

### rule of five

For a given class, if at least one of the copy constructor, move constructor, copy assignment operator, move assignment operator or destructor is explicitly declared, all of them should be explicitly declared.

### rule of zero

For a given class, if the copy constructor, move constructor, copy assignment operator, move assignment operator and destructor would all be inlined, public and defaulted, none of them should be explicitly declared.

### SMCP

The single-source multiple compiler-passes (SMCP) technique allows a single-source file to be parsed by multiple compilers for building native programs per compilation target. For example, a standard C++ CPU compiler for targeting [host](#) will parse the [SYCL file](#) to create the C++ [SYCL application](#) which offloads parts of the computation to other [devices](#). A SYCL device compiler will parse the same source file and target only SYCL kernels. For the full description please refer to [Section 3.12.1](#). See [SSCP](#) for



another approach.

### **specialization constant**

A constant variable where the value is not known until compilation of the [SYCL kernel function](#).

### **specialization id**

An identifier which represents a reference to a [specialization constant](#) both in the [SYCL application](#) for setting the value prior to the compilation of a [kernel bundle](#) and in a [SYCL kernel function](#) for retrieving the value during invocation.

### **SSCP**

The single-source single compiler-pass (SSCP) technique allows a single-source file to be parsed only once by a single compiler. For example, the SYCL compiler will parse the [SYCL file](#) once. Then, from this single intermediate representation, for each kind of device architecture a compilation flow will generate the binary for each kernel and another compilation flow will generate the [host](#) code of the C++ [SYCL application](#). For the full description please refer to [Section 3.12.2](#). See [SMCP](#) for another approach.

### **string kernel name**

The name of a [SYCL kernel function](#) in string form, this can be the name of a kernel function created via interop or a string form of a [type kernel name](#).

### **sub-group**

The SYCL sub-group (`sycl::sub_group` class) is a representation of a collection of related work-items within a [work-group](#). For further details for the `sycl::sub_group` class see [Section 4.9.1.8](#).

### **sub-group barrier**

A [group barrier](#) for all [work-items](#) in a [sub-group](#).

### **sub-group mem-fence**

A [mem-fence](#) for all [work-items](#) in a [sub-group](#).

### **SYCL application**

A SYCL application is a C++ application which uses the SYCL programming model in order to execute [kernels](#) on [devices](#).

### **SYCL backend**

An implementation of the SYCL programming model using an heterogeneous programming API. A SYCL backend exposes one or multiple SYCL [platforms](#). For example, the OpenCL backend, via the ICD loader, can expose multiple OpenCL [platforms](#).

### **SYCL backend API**

The exposed API for writing SYCL code against a given [SYCL backend](#).

### **SYCL C++ template library**

The template library is a set of C++ templated classes which provide the programming interface to the SYCL developer.

### **SYCL file**

A SYCL C++ source file that contains SYCL API calls.

### **SYCL kernel function**

A type which is callable with `operator()` that takes an [id](#), [item](#), [nd-item](#) or [work-group](#), and an optional [kernel\\_handler](#) as its last parameter. This type can be passed to kernel enqueue member functions of the [command group handler](#). A [SYCL kernel function](#) defines an entry point to a [kernel](#). The function object can be a named [device copyable](#) type or lambda expression.

## SYCL runtime

A SYCL runtime is an implementation of the SYCL API specification. The SYCL runtime manages the different [platforms](#), [devices](#), [contexts](#) as well as memory handling of data between host and [SYCL backend contexts](#) to enable semantically correct execution of SYCL programs.

## type kernel name

The name of a [SYCL kernel function](#) in type form, this can be either a [kernel name](#) provided to a [kernel invocation command](#) or the type of a function object use as a [SYCL kernel function](#).

## USM

Unified Shared Memory (USM) provides a pointer-based alternative to the [buffer](#) programming model. USM enables:

- easier integration into existing code bases by representing allocations as pointers rather than buffers, with full support for pointer arithmetic into allocations;
- fine-grain control over ownership and accessibility of allocations, to optimally choose between performance and programmer convenience;
- a simpler programming model, by automatically migrating some allocations between SYCL [devices](#) and the [host](#).

See [Section 4.8](#)

## work-group

The SYCL work-group ([sycl::group](#) class) is a representation of a collection of related [work-items](#) that execute on a single compute unit. The [work-items](#) in the group execute the same kernel-instance and [share local memory and work-group functions](#). For further details for the [sycl::group](#) class see [Section 4.9.1.7](#).

## work-group barrier

A [group barrier](#) for all [work-items](#) in a [work-group](#).

## work-group mem-fence

A [mem-fence](#) for all [work-items](#) in a [work-group](#).

## work-group id

As in OpenCL, SYCL kernels execute in [work-groups](#). The group ID is the ID of the [work-group](#) that a [work-item](#) is executing within. A group ID is an one, two or three dimensional value that starts at 0 per dimension.

## work-group range

A group range is the size of the [work-group](#) for every dimension.

## work-item

The SYCL work-item is a representation of a [work-item](#) among a collection of parallel executions of a kernel invoked on a [device](#) by a [command](#). A [work-item](#) is executed by one or more processing elements as part of a [work-group](#) executing on a compute unit. A [work-item](#) is distinguished from other [work-items](#) by its [global id](#) or the combination of its [work-group id](#) and its [local id](#) within a [work-group](#).