OpenMP Application Programming Interface

Version 6.0 November 2024

This page intentionally left blank in published version.

Contents

List of Figures

List of Tables

Part I

Definitions

1 Overview of the OpenMP API

The collection of compiler directives, library routines, environment variables, and tool support that this document describes collectively define the specification of the OpenMP Application Program Interface (OpenMP API) for C, C++ and Fortran base programs. This specification provides a model for parallel programming that is portable across architectures from different vendors. Compilers from numerous vendors support the OpenMP API. More information about the OpenMP API can be found at the following web site: **`https://www.openmp.org`**.

The directives, routines, environment variables, and tool support that this document defines allow users to create, to manage, to debug and to analyze parallel programs while permitting portability. The directives extend the C, C++ and Fortran base languages with single program multiple data (SPMD) constructs, tasking constructs, device constructs, work-distribution constructs, and synchronization constructs, and they provide support for sharing, mapping and privatizing data. The functionality to control the runtime environment is provided by routines and environment variables. Compilers that support the OpenMP API often include command line options to enable or to disable interpretation of some or all OpenMP directives.

1.1 Scope

The OpenMP API covers only user-directed parallelization, wherein the programmer explicitly specifies the actions to be taken by the compiler and runtime system in order to execute the program in parallel. OpenMP-compliant implementations are not required to check for data dependences, data conflicts, data races, or deadlocks. Compliant implementations also are not required to check for any code sequences that cause a program to be classified as a non-conforming program. Application developers are responsible for correctly using the OpenMP API to produce a conforming program. The OpenMP API does not cover compiler-generated automatic parallelization.

1.2 Execution Model

A compliant implementation must follow the abstract execution model that the supported base language and OpenMP specification define, as observable from the results of user code in a conforming program. These results do not include output from external monitoring tools or tools that use the OpenMP tool interfaces (i.e., OMPT and OMPD), which may reflect deviations from the execution model such as the unprescribed use of additional native threads, SIMD instruction, alternate loop transformations, or other target devices to facilitate parallel execution of the program.

The OpenMP API includes several directives. Some directives allow customization of base language declarations while other directives specify details of program execution. Such executable directives may be lexically associated with base language code. Each executable directive and any such associated base language code forms a construct. An OpenMP program executes regions, which consist of all code encountered by native threads.

Some regions are implicit but many are explicit regions, which correspond to a specific instance of a construct or routine. Execution is composed of nested regions since a given region may encounter additional constructs and routines. References to regions, particularly explicit regions or nested regions, that correspond to a specific type of construct or routine usually include the name of that construct or routine to identify the type of region that results.

With the OpenMP API, multiple threads execute tasks defined implicitly or explicitly by OpenMP directives and their associated user code, if any. An implementation may use multiple devices for a given execution of an OpenMP program. Concurrent execution of threads may result in different numeric results because of changes in the association of numeric operations.

Each device executes a set of one or more contention groups. Each contention group consists of a set of tasks that an associated set of threads, an OpenMP thread pool, executes. The lifetime of the OpenMP thread pool is the same as that of the contention group. The threads that are associated with each contention group are distinct from threads associated with any other contention group. Threads cannot migrate to execute tasks of a different contention group.

Each OpenMP thread pool has an initial thread, which may be the thread that starts execution of a region that is not nested within any other region, or which may be the thread that starts execution of the structured block associated with a **`target`** or **`teams`** construct. Each initial thread executes sequentially; the code that it encounters is part of an implicit task region, called an initial task region, that is generated by the implicit parallel region that surrounds all code executed by the initial thread. The other threads in the OpenMP thread pool associated with a contention group are unassigned threads. An implicit task is assigned to each of those threads. When a task encounters a **`parallel`** construct, some of the unassigned threads become assigned threads that are assigned to the team of that **`parallel`** region.

The thread that executes the implicit parallel region that surrounds the whole program executes on the host device. An implementation may support other devices besides the host device. If supported, these devices are available to the host device for *offloading* code and data. Each device has its own contention groups.

A task that encounters a **`target`** construct generates a new target task; its region encloses the **`target`** region. The target task is complete after the **`target`** region completes execution. When a target task executes, an initial thread executes the enclosed **`target`** region. The initial thread executes sequentially, as if the **`target`** region is part of an initial task region that an implicit parallel region generates. The initial thread may execute on the requested target device, if it is available. If the target device does not exist or the implementation does not support it, all **`target`** regions associated with that device execute on the host device. Otherwise, the implementation ensures that the **target** region executes as if it were executed in the data environment of the target device unless an **if** clause is present and the **if** clause expression evaluates to *false*.

The **teams** construct creates a league of teams, where each team is an initial team that comprises an initial thread that executes the **teams** region and that executes a distinct contention group from those of initial threads. Each initial thread executes sequentially, as if the code encountered is part of an initial task region that is generated by an implicit parallel region associated with each team. Whether the initial threads concurrently execute the **teams** region is unspecified, and a program that relies on their concurrent execution for the purposes of synchronization may deadlock.

Any thread that encounters a **parallel** construct becomes the primary thread of the new team that consists of itself and zero or more additional unassigned threads that are then assigned to that team as team-worker threads. Those threads remain assigned threads for the lifetime of that team. A set of implicit tasks, one per thread, is generated. The code inside the **parallel** construct defines the code for each implicit task. A different thread in the team is assigned to each implicit task, which is tied, that is, only that assigned thread ever executes it. The task region of the task being executed by the encountering thread is suspended, and each member of the new team executes its implicit task. The primary thread is the parent thread of any thread that executes a task that is bound to the parallel region. An implicit barrier occurs at the end of the **parallel** region. Only the primary thread resumes execution beyond the end of that region, resuming the suspended task region. The other threads again become unassigned threads. A single program can specify any number of **parallel** constructs.

parallel regions may be arbitrarily nested inside each other. If nested parallelism is disabled, or is not supported by the OpenMP implementation, then the new team that is formed by a thread that encounters a **parallel** construct inside a **parallel** region will consist only of the encountering thread. However, if nested parallelism is supported and enabled, then the new team can consist of more than one thread. A **parallel** construct may include a **proc_bind** clause to specify the places to use for the threads in the team within the **parallel** region.

When any team encounters a partitioned worksharing construct, the work inside the construct is divided into work partitions, each of which is executed by one member of the team, instead of the work being executed redundantly by each thread. An implicit barrier occurs at the end of any region that corresponds to a worksharing construct for which the **nowait** clause is not specified. Redundant execution of code by every thread in the team resumes after the end of the worksharing construct. Regions that correspond to team-executed constructs, including all worksharing regions and barrier regions, are executed by the current team such that all threads in the team execute the team-executed regions in the same order.

When a **loop** construct is encountered, the logical iterations of the collapsed loops, which are the affected loops as specified by the **collapse** clause, are executed in the context of its encountering threads, as determined according to its binding region. If the **loop** region binds to a **teams** region, the region is encountered by the set of primary thread that execute the **teams** region. If the **loop** region binds to a **parallel** region, the region is encountered by the team that execute the **parallel** region. Otherwise, the region is encountered by a single thread. If the **loop** region binds to a **`teams`** region, the encountering threads may continue execution after the **`loop`** region without waiting for all iterations to complete; the iterations are guaranteed to complete before the end of the **`teams`** region. Otherwise, all iterations must complete before the encountering threads continue execution after the **`loop`** region. All threads that encounter the **`loop`** construct may participate in the execution of the iterations. Only one thread may execute any given iteration.

When any thread encounters a **`simd`** construct, the iterations of the loop associated with the construct may be executed concurrently using the SIMD lanes that are available to the thread.

When any thread encounters a task-generating construct, one or more explicit tasks are generated. Explicitly generated tasks are scheduled onto threads of the binding thread set of the task, subject to the availability of the threads to execute work. Thus, execution of the new task could be immediate, or deferred until later according to task scheduling constraints and thread availability. Completion of all explicit tasks bound to a given parallel region is guaranteed before the primary thread leaves the implicit barrier at the end of the region. Completion of a subset of all explicit tasks bound to a given parallel region may be specified through the use of task synchronization constructs. Completion of all explicit tasks bound to an implicit parallel region is guaranteed when the associated initial task completes. The initial task on the host device that begins a typical OpenMP program is guaranteed to end by the time that the program exits.

Threads are allowed to suspend the current task region at a task scheduling point in order to execute a different task. Thus, each task consists of a set of one or more subtasks that each correspond to the portion of the task region between any two consecutive task scheduling points that the task encounters. If the task region of a tied task is suspended, the initially assigned thread later resumes execution of the next subtask of the suspended task region. If the task region of an untied task is suspended, any thread in the binding thread set of the task may resume execution of its next subtask.

OpenMP threads are logical execution entities that are mapped to native threads for actual execution. OpenMP does not dictate the details of the implementation of native threads and, instead, specifies requirements on the thread state of OpenMP threads. As long as those requirements are met, a compliant implementation may map the same OpenMP thread differently (i.e., to different native threads) for different portions of its execution (e.g., for the execution of different subtasks). Similarly, while the lifetime of an OpenMP thread and its OpenMP thread pool is identical to that of the associated contention group, OpenMP does not specify the lifetime of any native threads to which it is mapped. Native threads may be created at any time and may be terminated at any time.

The **`cancel`** construct can alter the previously described flow of execution in a region. The effect of the **`cancel`** construct depends on the *cancel-directive-name* that is specified on it. If a task encounters a **`cancel`** construct with a **`taskgroup`** clause, then the explicit task activates cancellation and continues execution at the end of its **`task`** region, which implies completion of that task. Any other task in that **`taskgroup`** that has begun executing completes execution unless it encounters a cancellation point, including one that corresponds to a **`cancellation point`** construct, in which case it continues execution at the end of its explicit **`task`** region, which implies its completion. Other tasks in that **`taskgroup`** region that have not begun execution are aborted, which implies their completion.

If a task encounters a **cancel** construct with any other *cancel-directive-name* clause, it activates cancellation of the innermost enclosing region of the type specified and the thread continues execution at the end of that region. Tasks check if cancellation has been activated for their region at cancellation points and, if so, also resume execution at the end of the canceled region.

If cancellation has been activated, regardless of the *cancel-directive-name* clauses, threads that are waiting inside a barrier other than an implicit barrier at the end of the canceled region exit the barrier and resume execution at the end of the canceled region. This action can occur before the other threads reach that barrier.

OpenMP specifies circumstances that cause error termination. If compile-time error termination is specified, the effect is as if the program encounters an **error** directive on which a **severity** clause specifies a *sev-level* argument of **fatal** and an **at** clause specifies an *action-time* argument of **compilation**. If runtime error termination is specified, the effect is as if the program encounters an **error** directive on which a **severity** clause specifies a *sev-level* argument of **fatal** and an **at** clause specifies an *action-time* argument of **execution**.

A construct that creates a data environment creates it at the time that the construct is encountered. The description of a construct defines whether it creates a data environment. Synchronization constructs and routines are available in the OpenMP API to coordinate tasks and their data accesses. In addition, routines and environment variables are available to control or to query the runtime environment of OpenMP programs. The scope of OpenMP synchronization mechanisms may be limited to the contention group of the encountering task. Except where explicitly specified, any effect of the mechanisms between contention groups is implementation defined. Section 1.3 details the OpenMP memory model, including the effect of these features.

The OpenMP specification makes no guarantee that input or output to the same file is synchronous when executed in parallel. In this case, the programmer is responsible for synchronizing input and output processing with the assistance of synchronization constructs or routines.

Each native thread that enables the execution of a task by an OpenMP thread executes on a hardware thread. A hardware thread executes a stream of instructions defined by a given task region, so that only one OpenMP thread may execute on a hardware thread at a time. A set of consecutive hardware threads may form a progress unit. Hardware threads execute distinct streams of instructions unless they are part of the same progress unit. Threads that execute in the same progress unit may execute from a common stream of instructions, with serialized execution of diverging code paths that occur due to conditional statements. A program that relies on concurrent execution of such diverging code paths for the purposes of synchronization may deadlock.

All concurrency semantics defined by the base language with respect to base language threads apply to OpenMP threads, unless otherwise specified. An OpenMP thread *makes progress* when it performs a flush operation, performs input or output processing, terminates, or makes progress as defined by the base language. OpenMP threads will eventually make progress in the absence of dependence cycles, unless otherwise specified by the base language. A dependence cycle may be implicitly introduced between synchronizing threads where concurrent execution is not guaranteed. Threads may therefore not make progress if the program includes synchronizing threads that descend from different initial teams formed by a **teams** construct or if the program includes synchronizing divergent threads from the same team that execute on the same progress unit. The generation and execution of explicit tasks by threads in the current team does not prevent any of the threads from making progress if executing the explicit tasks as included tasks would ensure that they make progress.

Each device is identified by a device number. The device number for the host device is the value of the total number of non-host devices, while each non-host device has a unique device number that is greater than or equal to zero and less than the device number for the host device. Additionally, the predefined identifier **omp_initial_device** can be used as an alias for the host device and the predefined identifier **omp_invalid_device** can be used to specify an invalid device number. A conforming device number is either a non-negative integer that is less than or equal to the value returned by **omp_get_num_devices** or equal to **omp_initial_device** or **omp_invalid_device**.

A signal handler may only execute directives and routines that have the async-signal-safe property.

1.3 Memory Model

1.3.1 Structure of the OpenMP Memory Model

The OpenMP API provides a relaxed-consistency, shared-memory model. All OpenMP threads have access to a place to store and to retrieve variables, called the memory. A given storage location in the memory may be associated with one or more devices, such that only threads on associated devices have access to it. In addition, each thread is allowed to have its own temporary view of the memory. The temporary view of memory for each thread is not a required part of the OpenMP memory model, but can represent any kind of intervening structure, such as machine registers, cache, or other local storage, between the thread and the memory. The temporary view of memory allows the thread to cache variables and thereby to avoid going to memory for every reference to a variable. Each thread also has access to another type of memory that must not be accessed by other threads, called threadprivate memory.

A directive that accepts data-sharing attribute clauses determines two kinds of access to variables used in the associated structured block of the directive: shared variables and private variables. Each variable referenced in the structured block has an original variable, which is the variable by the same name that exists in the OpenMP program immediately outside the construct. Each reference to a shared variable in the structured block becomes a reference to the original variable. For each private variable referenced in the structured block, a new version of the original variable (of the same type and size) is created in memory for each task or SIMD lane that executes code associated with the directive. Creation of the new version does not alter the value of the original variable. However, attempts to access the original variable from within the region that corresponds to the directive result in unspecified behavior; see Section 7.5.3 for additional details. References to a private variable in the structured block refer to the private version of the original variable for the current task or SIMD lane. The relationship between the value of the value of the original variable and the initial or final value of the private version depends on the exact clause that specifies it. Details of this issue, as well as other issues with privatization, are provided in Chapter 7.

The minimum size at which a memory update may also read and write back adjacent variables that are part of an aggregate variable is implementation defined but is no larger than the base language requires.

A single access to a variable may be implemented with multiple load or store instructions and, thus, is not guaranteed to be an atomic operation with respect to other accesses to the same variable. Accesses to variables smaller than the implementation defined minimum size or to C or C++ bit-fields may be implemented by reading, modifying, and rewriting a larger unit of memory, and may thus interfere with updates of variables or fields in the same unit of memory.

Two memory operations are considered unordered if the order in which they must complete, as seen by their affected threads, is not specified by the memory consistency guarantees listed in Section 1.3.6. If multiple threads write to the same memory unit (defined consistently with the above access considerations) then a data race occurs if the writes are unordered. Similarly, if at least one thread reads from a memory unit and at least one thread writes to that same memory unit then a data race occurs if the read and write are unordered. If a data race occurs then the result of the OpenMP program is unspecified behavior.

A private variable in a task region that subsequently generates an inner nested **`parallel`** region is permitted to be made shared for implicit tasks in the inner **`parallel`** region. A private variable in a task region can also be shared by an explicit task region generated during its execution. However, the programmer must use synchronization that ensures that the lifetime of the variable does not end before completion of the explicit task region sharing it. Any other access by one task to the private variables of another task results in unspecified behavior.

A storage location in memory that is associated with a given device has a device address that may be dereferenced by a thread executing on that device, but it may not be generally accessible from other devices. A different device may obtain a device pointer that refers to this device address. The manner in which an OpenMP program can obtain the referenced device address from a device pointer, outside of mechanisms specified by OpenMP, is implementation defined. Unless otherwise specified, the atomic scope of a storage location is all threads on the current device.

1.3.2 Device Data Environments

When an OpenMP program begins, an implicit **`target_data`** region for each device surrounds the whole program. Each device has a device data environment that is defined by its implicit **`target_data`** region. Any declare target directives and directives that accept data-mapping attribute clauses determine how an original storage block in a data environment is mapped to a corresponding storage block in a device data environment. Additionally, if a variable with static storage duration has original storage that is accessible on a device, and the variable is not a device-local variable, it may be treated as if its storage is mapped with a persistent self map in the implicit **`target_data`** region of the device; whether this happens is implementation defined.

When an original storage block is mapped to a device data environment and a corresponding storage block is not present in the device data environment, a new corresponding storage block (of the same type and size as the original storage block) is created in the device data environment. Conversely, the original storage block becomes the corresponding storage block of the new storage block in the device data environment of the device that performs a mapping operation.

The corresponding storage block in the device data environment may share storage with the original storage block. Writes to the corresponding storage block may alter the value of the original storage block. Section 1.3.6 discusses the impact of this possibility on memory consistency. When a task executes in the context of a device data environment, references to the original storage block refer to the corresponding storage block in the device data environment. If an original storage block is not currently mapped and a corresponding storage block does not exist in the device data environment then accesses to the original storage block result in unspecified behavior unless the **`unified_shared_memory`** clause is specified on a **`requires`** directive for the compilation unit.

The relationship between the value of the original storage block and the initial or final value of the corresponding storage block depends on the *map-type*. Details of this issue, as well as other issues with mapping a variable, are provided in Section 7.9.6.

The original storage block in a data environment and a corresponding storage block in a device data environment may share storage. Without intervening synchronization data races can occur.

If a storage block has a corresponding storage block with which it does not share storage, a write to a storage location designated by the storage block causes the value at the corresponding storage block to become undefined.

1.3.3 Memory Management

The host device, and other devices that an implementation may support, have attached storage resources where variables are stored. These resources can have different traits. A memory space in an OpenMP program represents a set of these storage resources. Memory spaces are defined according to a set of traits, and a single resource may be exposed as multiple memory spaces with different traits or may be part of multiple memory spaces. In any device, at least one memory space is guaranteed to exist.

An OpenMP program can use a memory allocator to allocate memory in which to store variables. This memory will be allocated from the storage resources of the memory space associated with the memory allocator. Memory allocators are also used to deallocate previously allocated memory. When a memory allocator is not used to allocate memory, OpenMP does not prescribe the storage resource for the allocation; the memory for the variables may be allocated in any storage resource.

1.3.4 The Flush Operation

The memory model has relaxed-consistency because the temporary view of memory of a thread is not required to be consistent with memory at all times. A value written to a variable can remain in that temporary view until it is forced to memory at a later time. Likewise, a read from a variable may retrieve the value from that temporary view, unless it is forced to read from memory. OpenMP flush operations are used to enforce consistency between the temporary view of memory of a thread and memory, or between the temporary views of multiple threads.

A flush has an associated thread-set that constrains the threads for which it enforces memory consistency. Consistency is only guaranteed to be enforced between the view of memory of these threads. Unless otherwise specified, the thread-set of a flush only includes all threads on the current device.

If a flush is a strong flush, it enforces consistency between the temporary view of a thread and memory. A strong flush is applied to a set of variable called the flush-set. A strong flush restricts how an implementation may reorder memory operations. Implementations must not reorder the code for a memory operation for a given variable, or the code for a flush for the variable, with respect to a strong flush that refers to the same variable.

If a thread has performed a write to its temporary view of a shared variable since its last strong flush of that variable then, when it executes another strong flush of the variable, the strong flush does not complete until the value of the variable has been written to the variable in memory. If a thread performs multiple writes to the same variable between two strong flushes of that variable, the strong flush ensures that the value of the last write is written to the variable in memory. A strong flush of a variable executed by a thread also causes its temporary view of the variable to be discarded, so that if its next memory operation for that variable is a read, then the thread will read from memory and capture the value in its temporary view. When a thread executes a strong flush, no later memory operation by that thread for a variable in the flush-set of that strong flush is allowed to start until the strong flush completes. The completion of a strong flush executed by a thread is defined as the point at which all writes to the flush-set performed by the thread before the strong flush are visible in memory to all other threads, and at which the temporary view of the flush-set of that thread is discarded.

A strong flush provides a guarantee of consistency between the temporary view of a thread and memory. Therefore, a strong flush can be used to guarantee that a value written to a variable by one thread may be read by a second thread. To accomplish this, the programmer must ensure that the second thread has not written to the variable since its last strong flush of the variable, and that the following sequence of events are completed in this specific order:

1. The value is written to the variable by the first thread;
2. The variable is flushed, with a strong flush, by the first thread;
3. The variable is flushed, with a strong flush, by the second thread; and
4. The value is read from the variable by the second thread.

If a flush is a release flush or acquire flush, it can enforce consistency between the views of memory of two synchronizing threads. A release flush guarantees that any prior operation that writes or reads a shared variable will appear to be completed before any operation that writes or reads the same shared variable and follows an acquire flush with which the release flush synchronizes (see Section 1.3.5 for more details on flush synchronization). A release flush will propagate the values of all shared variables in its temporary view to memory prior to the thread performing any subsequent atomic operation that may establish a synchronization. An acquire flush will discard any value of a shared variable in its temporary view to which the thread has not written since last performing a release flush, and it will load any value of a shared variable propagated by a release flush that synchronizes with it (according to the synchronizes-with relation) into its temporary view so that it may be subsequently read. Therefore, release flushes and acquire flushes may also be used to guarantee that a value written to a variable by one thread may be read by a second thread. To accomplish this, the programmer must ensure that the second thread has not written to the variable since its last acquire flush, and that the following sequence of events happen in this specific order:

1. The value is written to the variable by the first thread;
2. The first thread performs a release flush;
3. The second thread performs an acquire flush; and
4. The value is read from the variable by the second thread.

Note – OpenMP synchronization operations, described in Chapter 17 and in Chapter 28, are recommended for enforcing this order. Synchronization through variables is possible but is not recommended because the proper timing of flushes is difficult.

The flush properties that define whether a flush is a strong flush, a release flush, or an acquire flush are not mutually disjoint. A flush may be a strong flush and a release flush; it may be a strong flush and an acquire flush; it may be a release flush and an acquire flush; or it may be all three.

1.3.5 Flush Synchronization and Happens-Before Order

OpenMP supports thread synchronization with the use of release flushes and acquire flushes. For any such synchronization, a release flush is the source of the synchronization and an acquire flush is the sink of the synchronization, such that the release flush synchronizes with the acquire flush.

A release flush has one or more associated release sequences that define the set of modifications that may be used to establish a synchronization. A release sequence starts with an atomic operation that follows the release flush and modifies a shared variable and additionally includes any read-modify-write atomic operations that read a value taken from some modification in the release sequence. The following rules determine the atomic operation that starts an associated release sequence.

- If a release flush is performed on entry to an atomic operation, that atomic operation starts its release sequence.
- If a release flush is performed in an implicit flush region, an atomic operation that is provided by the implementation and that modifies an internal synchronization variable starts its release sequence.
- If a release flush is performed by an explicit **flush** region, any atomic operation that modifies a shared variable and follows the **flush** region in the program order of its thread starts an associated release sequence.

An acquire flush is associated with one or more prior atomic operations that read a shared variable and that may be used to establish a synchronization. The following rules determine the associated atomic operation that may establish a synchronization.

- If an acquire flush is performed on exit from an atomic operation, that atomic operation is its associated atomic operation.
- If an acquire flush is performed in an implicit flush region, an atomic operation that is provided by the implementation and that reads an internal synchronization variable is its associated atomic operation.
- If an acquire flush is performed by an explicit **flush** region, any atomic operation that reads a shared variable and precedes the **flush** region in the program order of its thread is an associated atomic operation.

The atomic scope of the internal synchronization variable that is used in implicit flush regions is the intersection of the thread-sets of the synchronizing flushes.

A release flush synchronizes with an acquire flush if the following conditions are satisfied:

- An atomic operation associated with the acquire flush reads a value written by a modification from a release sequence associated with the release flush; and
- The thread that performs each flush is in both of their respective thread-sets.

An operation *X* simply happens before an operation *Y*, that is, *X* precedes *Y* in simply happens-before order, if any of the following conditions are satisfied:

1. *X* and *Y* are performed by the same thread, and *X* precedes *Y* in the program order of the thread;
2. *X* synchronizes with *Y* according to the flush synchronization conditions explained above or according to the definition of the synchronizes with relation in the base language, if such a definition exists; or
3. Another operation, *Z*, exists such that *X* simply happens before *Z* and *Z* simply happens before *Y*.

An operation *X* happens before an operation *Y* if any of the following conditions are satisfied:

1. *X* happens before *Y*, as defined in the base language if such a definition exists; or
2. *X* simply happens before *Y*.

A variable with an initial value is treated as if the value is stored to the variable by an operation that happens before all operations that access or modify the variable in the program.

1.3.6 OpenMP Memory Consistency

The following rules guarantee an observable completion order for a given pair of memory operations in race-free programs, as seen by all affected threads. If both memory operations are strong flushes, the affected threads are all threads in both of their respective thread-sets. If exactly one of the memory operations is a strong flush, the affected threads are all threads in its thread-set. Otherwise, the affected threads are all threads.

- If two operations performed by different threads are sequentially consistent atomic operations or they are strong flushes that flush the same variable, then they must be completed as if in some sequential order, seen by all affected threads.
- If two operations performed by the same thread are sequentially consistent atomic operations or they access, modify, or, with a strong flush, flush the same variable, then they must be completed as if in the program order of that thread, as seen by all affected threads.
- If two operations are performed by different threads and one happens before the other, then they must be completed as if in that happens-before order, as seen by all affected threads, if:
  - both operations access or modify the same variable;
  - both operations are strong flushes that flush the same variable; or
  - both operations are sequentially consistent atomic operations.
- Any two atomic operations from different **atomic** regions must be completed as if in the same order as the strong flushes implied in their regions, as seen by all affected threads.

The flush operation can be specified using the **flush** directive, and is also implied at various locations in an OpenMP program; see Section 17.8.6 for details.

Note – Since flushes by themselves cannot prevent data races, explicit flushes are only useful in combination with non-sequentially consistent atomic constructs.

OpenMP programs that:

- Do not use non-sequentially consistent atomic constructs;
- Do not rely on the accuracy of a *false* result from **omp_test_lock** and **omp_test_nest_lock**; and

- Correctly avoid data races as required in Section 1.3.1,

behave as though operations on shared variables were simply interleaved in an order consistent with the order in which they are performed by each thread. The relaxed consistency model is invisible for such programs, and any explicit flushes in such programs are redundant.

1.4 Tool Interfaces

The OpenMP API includes two tool interfaces, OMPT and OMPD, to enable development of high-quality, portable, tools that support monitoring, performance, or correctness analysis and debugging of OpenMP programs developed using any implementation of the OpenMP API. An implementation of the OpenMP API may differ from the abstract execution model described by its specification. The ability of tools that use OMPT or OMPD to observe such differences does not constrain implementations of the OpenMP API in any way.

1.4.1 OMPT

The OMPT interface, which is intended for first-party tools, provides the following:

- A mechanism to initialize a first-party tool;
- Routines that enable a tool to determine the capabilities of an OpenMP implementation;
- Routines that enable a tool to examine OpenMP state information associated with a thread;
- Mechanisms that enable a tool to map implementation-level calling contexts back to their source-level representations;
- A callback interface that enables a tool to receive notification of OpenMP events;
- A tracing interface that enables a tool to trace activity on target devices; and
- A runtime library routine that an OpenMP program can use to control a tool.

OpenMP implementations may differ with respect to the thread states that they support, the mutual exclusion implementations that they employ, and the events for which tool callbacks are invoked. For some events, OpenMP implementations must guarantee that a registered callback will be invoked for each occurrence of the event. For other events, OpenMP implementations are permitted to invoke a registered callback for some or no occurrences of the event; for such events, however, OpenMP implementations are encouraged to invoke tool callbacks on as many occurrences of the event as is practical. Section 32.2.4 specifies the subset of OMPT callbacks that an OpenMP implementation must support for a minimal implementation of the OMPT interface.

With the exception of the **`omp_control_tool`** routine for tool control, all other routines in the OMPT interface are intended for use only by tools. For that reason, OMPT includes a Fortran binding only for **`omp_control_tool`**; all other OMPT functionality is supported with C syntax only.

1.4.2 OMPD

The OMPD interface is intended for third-party tools, which run as separate processes. An OpenMP implementation must provide an OMPD library that can be dynamically loaded and used by a third-party tool. A third-party tool, such as a debugger, uses the OMPD library to access OpenMP state of a program that has begun execution. OMPD defines the following:

- An interface that an OMPD library exports, which a tool can use to access OpenMP state of a program that has begun execution;
- A callback interface that a tool provides to the OMPD library so that the library can use it to access the OpenMP state of a program that has begun execution; and
- A small number of symbols that must be defined by an OpenMP implementation to help the tool find the correct OMPD library to use for that OpenMP implementation and to facilitate notification of events.

Chapter 38, Chapter 39, Chapter 40, Chapter 41, and Chapter 42 describe OMPD in detail.

1.5 OpenMP Compliance

The OpenMP API defines constructs that operate in the context of the base language that is supported by an implementation. If the implementation of the base language does not support a language construct that appears in this document, a compliant implementation is not required to support it, with the exception that for Fortran, the implementation must allow case insensitivity for directive and routine names, and it must allow identifiers of more than six characters. An implementation of the OpenMP API is compliant if and only if it compiles and executes all other conforming programs, and supports the tool interfaces, according to the syntax and semantics laid out in Chapters 1 through 42. All appendices as well as text designated as a note or comment (see Section 1.7) are for information purposes only and are not part of the specification.

All library, intrinsic and built-in procedures provided by the base language must be thread-safe procedures in a compliant implementation. In addition, the implementation of the base language must also be thread-safe. For example, **ALLOCATE** and **DEALLOCATE** statements must be thread-safe in Fortran. Unsynchronized concurrent use of such procedures by different threads must produce correct results (although not necessarily the same as serial execution results, as in the case of random number generation procedures).

Starting with Fortran 90, variables with explicit initialization have the **SAVE** attribute implicitly. This is not the case in Fortran 77. However, a compliant OpenMP Fortran implementation must give such a variable the **SAVE** attribute, regardless of the underlying base language version.

Appendix A lists certain aspects of the OpenMP API that are implementation defined. A compliant implementation must define and document its behavior for each of the items in Appendix A.

1.6 Normative References

- ISO/IEC 9899:1990, *Information Technology - Programming Languages - C*.
  This OpenMP API specification refers to ISO/IEC 9899:1990 as C90.
- ISO/IEC 9899:1999, *Information Technology - Programming Languages - C*.
  This OpenMP API specification refers to ISO/IEC 9899:1999 as C99.
- ISO/IEC 9899:2011, *Information Technology - Programming Languages - C*.
  This OpenMP API specification refers to ISO/IEC 9899:2011 as C11.
- ISO/IEC 9899:2018, *Information Technology - Programming Languages - C*.
  This OpenMP API specification refers to ISO/IEC 9899:2018 as C18.
- ISO/IEC 9899:2024, *Information Technology - Programming Languages - C*.
  This OpenMP API specification refers to ISO/IEC 9899:2024 as C23.
- ISO/IEC 14882:1998, *Information Technology - Programming Languages - C++*.
  This OpenMP API specification refers to ISO/IEC 14882:1998 as C++98.
- ISO/IEC 14882:2011, *Information Technology - Programming Languages - C++*.
  This OpenMP API specification refers to ISO/IEC 14882:2011 as C++11.
- ISO/IEC 14882:2014, *Information Technology - Programming Languages - C++*.
  This OpenMP API specification refers to ISO/IEC 14882:2014 as C++14.
- ISO/IEC 14882:2017, *Information Technology - Programming Languages - C++*.
  This OpenMP API specification refers to ISO/IEC 14882:2017 as C++17.
- ISO/IEC 14882:2020, *Information Technology - Programming Languages - C++*.
  This OpenMP API specification refers to ISO/IEC 14882:2020 as C++20.
- ISO/IEC 14882:2024, *Information Technology - Programming Languages - C++*.
  This OpenMP API specification refers to ISO/IEC 14882:2024 as C++23.
- ISO/IEC 1539:1980, *Information Technology - Programming Languages - Fortran*.
  This OpenMP API specification refers to ISO/IEC 1539:1980 as Fortran 77.
- ISO/IEC 1539:1991, *Information Technology - Programming Languages - Fortran*.
  This OpenMP API specification refers to ISO/IEC 1539:1991 as Fortran 90.
- ISO/IEC 1539-1:1997, *Information Technology - Programming Languages - Fortran*.
  This OpenMP API specification refers to ISO/IEC 1539-1:1997 as Fortran 95.
- ISO/IEC 1539-1:2004, *Information Technology - Programming Languages - Fortran*.
  This OpenMP API specification refers to ISO/IEC 1539-1:2004 as Fortran 2003.
- ISO/IEC 1539-1:2010, *Information Technology - Programming Languages - Fortran*.
  This OpenMP API specification refers to ISO/IEC 1539-1:2010 as Fortran 2008.

- ISO/IEC 1539-1:2018, *Information Technology - Programming Languages - Fortran*. This OpenMP API specification refers to ISO/IEC 1539-1:2018 as Fortran 2018.
- ISO/IEC 1539-1:2023, *Information Technology - Programming Languages - Fortran*. This OpenMP API specification refers to ISO/IEC 1539-1:2023 as Fortran 2023.
- Where this OpenMP API specification refers to C, C++ or Fortran, reference is made to the base language supported by the implementation.

1.7 Organization of this Document

The remainder of this document is structured as normative chapters that define the directives, including their syntax and semantics, the routines and the tool interfaces that comprise the OpenMP API. The document also includes appendices that facilitate maintaining a compliant implementation of the API.

Some sections of this document only apply to programs written in a certain base language. Text that applies only to programs for which the base language is C or C++ is shown as follows:

C / C++

C/C++ specific text...

C / C++

Text that applies only to programs for which the base language is C only is shown as follows:

C

C specific text...

C

Text that applies only to programs for which the base language is C++ only is shown as follows:

C++

C++ specific text...

C++

Text that applies only to programs for which the base language is Fortran is shown as follows:

Fortran

Fortran specific text...

Fortran

Text that applies only to programs for which the base language is Fortran or C++ is shown as follows:

Fortran / C++

Fortran/C++ specific text...

Fortran / C++

Where an entire page consists of base language specific text, a marker is shown at the top of the page. For Fortran-specific text, the marker is:

Fortran (cont.)

For C/C++-specific text, the marker is:

C/C++ (cont.)

Some text is for information only, and is not part of the normative specification. Such text is designated as a note or comment, like this:

Note – Non-normative text...

COMMENT: Non-normative text...

2 Glossary

A | B | C | D | E | F | G | H | I | L | M | N | O | P | R | S | T | U | V | W | Z

A

abstract name

A conceptual abstract name or a numeric abstract name. **128**, 34, 77, 128, 131, 134, 886, 897

accessible device

The host device or any non-host device accessible for execution. 119, 139–141, 360

accessible storage

A storage block that may be accessed by a given thread. 285, 606

acquire flush

A flush that has the acquire flush property. **10**, 11, 12, 92, 101, 496, 499, 501–504

acquire flush property

A flush with the acquire flush property orders memory operations that follow the flush after memory operations performed by a different thread that synchronizes with it. 19, 52, 499

active level

An active parallel region that encloses a given region at some point in the execution of an OpenMP program. The number of active levels is the number of active parallel regions that encloses the given region. 19, 75, 100, 129, 130, 133, 576, 886, 892, 911

active parallel region

A parallel region comprised of implicit tasks that are being executed by a team to which multiple threads are assigned. 19, 105, 115, 116, 132, 216, 217, 571, 576, 577, 579, 580, 885, 888, 915, 916

active target region

A **`target`** region that is executed on a device other than the device that encountered the **`target`** construct. 124

address range

The addresses of a contiguous set of storage locations. 51, 70, 99, 606

address space

A collection of logical, virtual, or physical memory address ranges that contain code, stack, and/or data. Address ranges within an address space need not be contiguous. An address space consists of one or more segments. 20, 52, 80, 95, 109, 145, 146, 359, 606, 699, 700, 820, 831, 836, 838, 839, 841–843, 846, 849, 850, 852, 853, 855, 870, 872, 874

address space context

A tool context that refers to an address space within an OpenMP process. 820

address space handle

A handle that refers to an address space within an OpenMP process. 828, 849–851, 857, 868

affected iteration

A logical iteration of the affected loops of a loop-nest-associated directive. 60, 94, 97, 382

affected loop

A loop from a canonical loop nest, or a **DO CONCURRENT** loop in Fortran, that is affected by a given loop-nest-associated directive. **203**, 4, 20, 62, 67, 68, 108, 113, 154, 203–205, 211, 212, 226, 230, 231, 233, 234, 253, 259, 267, 268, 371, 372, 378–381, 424, 910

affected loop nest

The subset of canonical loop nests of an associated loop sequence that are selected by the **looprange** clause. **207**, 35, 92, 205, 371, 375

aggregate variable

A variable, such as an array or structure, composed of other variables. For Fortran, a variable of character type is considered an aggregate variable. 8, 20, 40, 112, 164, 217, 223, 292, 445, 885

aligned-memory-allocating routine

A memory-management routine that has the aligned-memory-allocating-routine property. **654**, 655, 657, 659

aligned-memory-allocating-routine property

The property that a memory-allocating routine ensures the allocated memory is aligned with respect to an *alignment* argument. **654**, 20, 657, 659

all-constituents property

The property that a clause applies to all leaf constructs that permit it when the clause appears on a compound directive. **159**, 160, 528

all-contention-group-tasks binding property

The binding property that the binding task set is all tasks in the contention group. **534**, 664–671, 673–676

all-data-environments clause

A clause that has the all-data-environments property. 73, 236, 238

all-data-environments property

The property that a data-sharing attribute clause affects any data environment for which it is specified, including minimal data environments. 21, 236, 238, 257

all-device-tasks binding property

The binding property that the binding task set is all tasks on a specified device. **690**

all-device-threads binding property

The binding property that the binding thread set is all threads on the current device. The effect of executing a construct or a routine with this property is not related to any specific region that corresponds to any other construct or routine. **534**, 586, 594, 630–636, 638–644, 646–651, 679, 680, 791, 792

allocator

A memory allocator. 21, 143, 144, 305–312, 315, 316, 358, 463, 545, 547, 555, 558, 638–640, 645, 647, 652–655, 662, 888, 899, 900, 904, 905

allocator structured block

A context-specific structured block that may be associated with an **`allocators`** directive. **187**, 315

allocator trait

A trait of an allocator. 144, 305, 307, 308, 311, 313, 547, 549, 552, 638, 645, 888, 899, 900, 910

all-privatizing property

The property that a clause, when it appears on a combined construct or a composite construct, applies to all constituent constructs to which it applies for which a data-sharing attribute clause may create a private copy of the same list item. **159**, 312, 528

all tasks

All tasks participating in the OpenMP program or in a specified limiting context. 21, 28, 251, 301, 306, 535, 690

all-tasks binding property

The binding property that the binding task set is all tasks. **690**, 689, 690

all threads

All OpenMP threads participating in the OpenMP program. A specific usage of the term may be explicitly limited to a limiting context, such as all threads on a given device or an OpenMP thread pool. 8, 13, 21, 22, 28, 231, 494, 535, 630, 691, 791–793

all-threads binding property

The binding property that the binding thread set is all threads. The effect of executing a construct or a routine with this property is not related to any specific region that corresponds to any other construct or routine. **534**

ancestor thread

For a given thread, its parent thread or one of the ancestor threads of its parent thread. 22, 578, 579, 589, 902, 916

antecedent task

A task that must complete before its dependent tasks can be executed. **507**, 42, 51, 59, 86, 103, 503, 507, 509, 762

argument list

A list that is used as an argument of a directive, clause, or modifier. **158**, 46, 47, 51, 63, 65, 80, 83, 86, 87, 108, 112, 159, 162, 163, 210, 218, 219, 269, 270

array base

The base array of a given array section or array element, if it exists; otherwise, the base pointer of the array section or array element.

COMMENT: For the array section (*p0).x0[k1].p1->p2[k2].x1[k3].x2[4][0:n], where identifiers p*i* have a pointer type declaration and identifiers x*i* have an array type declaration, the array base is: (*p0).x0[k1].p1->p2[k2].x1[k3].x2.

More examples for C/C++:

- The array base for x[i] and for x[i:n] is x, if x is an array or pointer.
- The array base for x[5][i] and for x[5][i:n] is x, if x is a pointer to an array or x is 2-dimensional array.
- The array base for y[5][i] and for y[5][i:n] is y[5], if y is an array of pointers or y is a pointer to a pointer.

Examples for Fortran:

- The array base for x(i) and for x(i:j) is x.

22, 167, 168, 237, 239, 247, 277, 281, 282

array element

A single member of an array as defined by the base language. 23, 241, 247, 259, 269, 270, 276, 281, 286, 295, 296

array item

An array, an array section, or an array element. 529

array section

A designated subset of the elements of an array that is specified using a subscript notation that can select more than one array element. 22–24, 26, 27, 36, 37, 39, 74, 97, 112, 114, 140, 163, 166–168, 221, 236–239, 241, 243, 244, 247, 259, 269, 270, 280, 281, 283, 286, 288, 294, 295, 395, 444, 508, 509, 529, 898, 906, 909, 911, 912, 914

array shaping

A mechanism that reinterprets the region of memory to which an expression that has a type of pointer to *T* as an n-dimensional array of type *T*. 95, 909

assignable OpenMP type instance

An instance of an OpenMP type to which an assignment can be performed. **183**, 183

assigned list item

A list item to which assignment is performed as the result of a data-motion clause. 296, 298

assigned thread

A thread that has been assigned an implicit task of a parallel region. 3, 4, 87, 104, 106, 390, 391, 414, 569

assigning map type

A *map-type* for which the mapping operations may include an assignment operation. **275**

associated device

The associated device of a memory allocator is the device that is specified when the memory allocator is created. If the associated memory space is a predefined memory space, the associated device is the current device. 7, 23

associated loop nest

The associated canonical loop nest, or `DO CONCURRENT` loop in Fortran, of a loop-nest-associated directive. 67, 68, 203, 206, 207, 371, 374

associated loop sequence

The associated canonical loop sequence of a loop-sequence-associated directive. 20, 207, 371

associated memory space

The associated memory space of a memory allocator is the memory space that is specified when the memory allocator is created. 23, 24, 71, 305, 308

assumed-size array

For C/C++, an array section for which the *length* is absent and the size of the dimensions is not known. For Fortran, an assumed-size array in the base language. 24, 71, 114, 166, 168, 198, 212, 213, 222, 236, 238, 275, 280, 281, 286, 287, 535, 899, 915

assumption directive

A directive that provides invariants that specify additional information about the expected properties of the program that can optionally be used for optimization. 24, 362, 365, 904, 906

assumption scope

The scope for which the invariants specified by an assumption directive must hold. 362–369

asynchronous device routine

A routine that has the asynchronous-device routine property. 505, 603, 604, 616, 618, 621

asynchronous-device routine property

The property of a device routine that it performs its operation asynchronously. 24, 604, 615, 617, 620

async signal safe

The guarantee that interruption by signal delivery will not interfere with a set of operations. An async signal safe runtime entry point is safe to call from a signal handler. 24, 744, 777, 786

async-signal-safe entry point

An entry point that has the async-signal-safe property. 786

async-signal-safe property

The property of a routine or entry point that it is async signal safe. **7**, 24, 786, 791–795, 797, 799–801

atomic captured update

An atomic update operation that is specified by an **`atomic`** construct on which the **`capture`** clause is present. 111, 193, 491, 495, 914

atomic conditional update

An atomic update operation that is specified by an **`atomic`** construct on which the **`compare`** clause is present. 34, 35, 191, 491, 492, 495–497, 907

atomic operation

An operation that is specified by an **atomic** construct or is implicitly performed by the OpenMP implementation and that atomically accesses and/or modifies a specific storage location. 8, 11–13, 25, 89, 92, 95, 283, 284, 308, 472, 496, 497, 502, 907

atomic read

An atomic operation that is specified by an **atomic** construct on which the **read** clause is present. 89, 190, 488, 495

atomic scope

The set of threads that may concurrently access or modify a given storage location with atomic operations, where at least one of the operations modifies the storage location. 8, 12, 308, 494

atomic structured block

A context-specific structured block that may be associated with an **atomic** directive. **188**, 30, 89, 111, 114, 188, 193, 494–496, 898

atomic update

An atomic operation that is specified by an **atomic** construct on which the **update** clause is present. 24, 89, 111, 190, 489, 491, 495–497, 914

atomic write

An atomic operation that is specified by an **atomic** construct on which the **write** clause is present. 114, 190, 490, 495

attached pointer

A pointer variable or referring pointer in a device data environment that, as a result of a mapping operation, points to a given data entity that also exists in the device data environment. 85, 284, 287, 288, 296, 463

attach-ineligible

An attribute of a pointer for which pointer attachment may not be performed. 282

automatic storage duration

For C/C++, the lifetime of a variable or object with automatic storage duration, as defined by the base language. For Fortran, the lifetime of a variable, including implied-do, **FORALL**, and **DO CONCURRENT** indices, that is neither a variable that has static storage duration nor a dummy argument without the **VALUE** attribute. For referencing variables, this refers to the lifetime of the referring pointer unless explicitly specified otherwise. 211, 214, 220

available device

An available non-host device or, where explicitly specified, the host device. 139, 141, 319, 634, 652, 690

available non-host device

A non-host device that can be used for the current OpenMP program execution. 26, 139

B

barrier

A point in the execution of a program encountered by a team, beyond which no thread in the team may execute until all threads in the team have reached the barrier and all explicit tasks generated for execution by the team have executed to completion. If cancellation has been requested, threads may proceed to the end of the canceled region even if some threads in the team have not reached the barrier. 4, 6, 26, 50, 58, 273, 385, 402, 404–407, 409, 414, 448, 475–477, 482, 496, 500–502, 521, 590, 689, 704, 733, 763, 764, 902, 917

base address

If a data entity has a base pointer, the address of the first storage location of the implicit array of its base pointer; otherwise, if the data entity has a referenced pointee, the address of the first storage location of its referenced pointee; otherwise, if the data entity has a base variable, the address of the first storage location of its base variable; otherwise, the address of the first storage location of the data entity. 51, 236, 239, 281, 610

base array

For C/C++, a containing array of a given lvalue expression or array section that does not appear in the expression of any of its other containing arrays. For Fortran, a containing array of a given variable or array section that does not appear in the designator of any of its other containing arrays.

> COMMENT: For the array section (*p0).x0[k1].p1->p2[k2].x1[k3].x2[4][0:n], where identifiers p*i* have a pointer type declaration and identifiers x*i* have an array type declaration, the base array is: (*p0).x0[k1].p1->p2[k2].x1[k3].x2.

22, 26, 529

base function

A procedure that is declared and defined in the base language. 41, 54, 92, 113, 322, 328–333, 335, 336, 889

base language

A programming language that serves as the foundation of the OpenMP specification. Section 1.6 lists the current base languages for the OpenMP API. 2, 3, 6, 8, 12, 13, 15, 17, 18, 23–27, 29, 38, 39, 41, 42, 46, 48, 51, 53, 54, 56, 81, 86–88, 93, 94, 98, 100, 101, 109, 148, 151–153, 155, 156, 162–164, 166, 167, 169, 183–185, 189, 195, 196, 201, 203, 215, 221, 239, 240, 242, 247, 249, 259, 261, 264, 278, 281, 293, 294, 308, 309, 311, 315, 316, 331, 335, 337, 362, 411, 495, 516, 533, 535, 564, 885, 904, 905, 909

base language thread

A thread of execution that defines a single flow of control within the program and that may execute concurrently with other base language threads, as specified by the base language. 6, 27

base pointer

For C/C++, an lvalue pointer expression that is used by a given lvalue expression or array section to refer indirectly to its storage, where the lvalue expression or array section is part of the implicit array for that lvalue pointer expression. For Fortran, a data pointer that appears last in the designator for a given variable or array section, where the variable or array section is part of the pointer target for that data pointer.

> COMMENT: For the array section (*p0).x0[k1].p1->p2[k2].x1[k3].x2[4][0:n], where identifiers p*i* have a pointer type declaration and identifiers x*i* have an array type declaration, the base pointer is: (*p0).x0[k1].p1->p2.

22, 26–28, 38, 74, 211, 212, 239, 259, 282–287, 328, 436, 437, 461, 462, 528, 529

base program

A program written in a base language. 2, 80

base referencing variable

For C++, a referencing variable that is used by a given lvalue expression or array section to refer indirectly to its storage, where the lvalue expression or array section is part of the referenced pointee of the referencing variable. For Fortran, a referencing variable that appears last in the designator for a given variable or array section, where the variable or array section is part of the referenced pointee of the referencing variable. 212, 461

base variable

For a given data entity that is a variable or array section, a variable denoted by a base language identifier that is either the data entity or is a containing array or containing structure of the data entity.

> COMMENT:
>
> Examples for C/C++:
>
> - The data entities x, x[i], x[:n], x[i].y[j] and x[i].y[:n], where x and y have array type declarations, all have the base variable x.
> - The lvalue expressions and array sections p[i], p[:n], p[i].y[j] and p[i].y[:n], where p has a pointer type and p[i].y has an array type, has a base pointer p

but does not have a base variable.

Examples for Fortran:

- The data objects x, x(i), x(:n), x(i)%y(j) and x(i)%y(:n), where x and y are arrays, all have the base variable x.
- The data objects p(i), p(:n), p(i)%y(j) and p(i)%y(:n), where p is a pointer and p(i)%y is an array, has a base pointer p but does not have a base variable.
- For the associated pointer p, p is both its base variable and base pointer.

26–28, 217, 276, 287, 436, 437, 462, 463, 528, 529

binding implicit task

The implicit task of the current team assigned to the encountering thread. 28, 57, 124, 389, 652–654

binding-implicit-task binding property

The binding property that the binding task set is the binding implicit task. 652, 653

binding property

A property of a construct or a routine that determines the binding region, binding task set and/or binding thread set. 21, 22, 28, 49, 54, 535

binding region

The enclosing region that determines the execution context and limits the scope of the effects of the bound region is called the binding region. The binding region is not defined for regions for which the binding thread set is all threads or the encountering thread, nor is it defined for regions for which the binding task set is all tasks. 4, 28, 82, 205, 412, 423, 425, 475, 513, 514, 516, 520, 524, 535, 683, 685, 880, 881, 883, 893, 918

binding task set

The set of tasks that are affected by, or provide the context for, the execution of a region. The binding task set for a given region can be all tasks, the current team tasks, all tasks in the contention group, all tasks of the current team that are generated in the region, the binding implicit task, or the generating task. 21, 28, 54, 121, 338, 435, 454, 456, 458, 461, 465, 468, 478, 482, 535, 603, 652, 653, 690, 786, 880–883

binding thread set

The set of threads that are affected by, or provide the context for, the execution of a region. The binding thread set for a given region can be all threads on a specified set of devices, all threads that are executing tasks in a contention group, all primary threads that are executing the initial tasks of an enclosing **`teams`** region, the current team, or the encountering thread. 5, 21, 22, 28, 49, 82, 84, 92, 107, 113, 205, 229, 231, 384, 394, 398, 399, 402, 404–407, 409, 412–414, 420, 423–426, 429, 430, 435, 439, 446, 473, 475, 479, 482, 494–496, 498, 505, 514, 515, 520, 521, 524, 535, 630, 683, 685, 786, 791–793, 893, 901, 902

block-associated directive

A directive for which its associated base language code is a structured block. **153**, 37, 82, 151–155, 186, 315, 337, 369, 384, 394, 402, 405–407, 409, 412, 426, 435, 458, 460, 473, 478, 494, 515

bounds-independent loop

For a structured block sequence, an enclosed canonical loop nest where none of its loops have loop bounds that depend on the execution of a preceding executable statement in the sequence. 202

C

callback

A tool callback. xxvii, 14, 15, 29, 33, 45, 46, 72–74, 77–79, 81, 83, 85, 91, 101, 110, 250, 286, 346, 352, 386, 395, 403, 405–409, 411, 413, 415, 421, 427, 431, 446, 447, 449, 453, 455, 457, 459, 462, 466, 474–478, 480, 497, 500, 509, 513, 515, 516, 522, 590, 603, 604, 607, 609–611, 613–616, 618–621, 664–669, 671–675, 677, 695, 697, 698, 700, 701, 703–707, 720, 725, 730, 731, 737, 744–781, 783–787, 789, 790, 802, 803, 805–808, 810, 812, 816, 817, 821, 822, 826, 833–844, 846, 848, 851, 853, 870, 872, 874, 876, 894–896, 903, 908

callback dispatch

The processing of a registered callback when an associated event occurs, in a manner consistent with the return code provided when a first-party tool registered the callback. 29, 729, 807

callback registration

A process that makes a tool callback available to an OpenMP implementation to enable callback dispatch. 91, 700, 701, 703

canceled taskgroup set

A taskgroup set that has been canceled. **521**, 521

cancellable construct

A construct that has the cancellable property. **519**, 520, 524

cancellable property

The property that a construct may be subject to cancellation. **519**, 29, 384, 407, 416, 417, 478

cancellation

An action that cancels (that is, aborts) a region and causes the execution of implicit tasks or explicit tasks to proceed to the end of the canceled region. **521**, 5, 6, 26, 29, 30, 139, 404, 475, 476, 501, 504, 519–524, 688, 759, 913

cancellation point

A point at which implicit tasks and explicit tasks check if cancellation has been activated. If cancellation has been activated, they perform the cancellation. **520**, 5, 6, 111, 116, 139, 449, 475, 476, 501, 504, 521–524, 741

candidate

A replacement candidate. 324, 329

canonical frame address

An address associated with a procedure frame on a call stack that was the value of the stack pointer immediately prior to calling the procedure for which the frame represents the invocation. 721

canonical loop nest

A loop nest that complies with the rules and restrictions defined in Section 6.4.1. **196**, 20, 23, 29, 30, 54, 66–68, 76, 153, 197, 201–203, 206, 207, 230, 267, 370, 371, 374, 375, 379, 380, 382, 419, 531, 901, 909

canonical loop sequence

A sequence of canonical loop nests that complies with the rules and restrictions defined in Section 6.4.2. **202**, 23, 54, 67, 68, 153, 197, 203, 208, 371, 372, 378, 898, 900

capture structured block

An atomic structured block that may be associated with an **`atomic`** directive that expresses capture semantics. **192**, 192

C/C++-only property

The property that an OpenMP feature is only supported in C/C++. 536, 708–711, 714–732, 734–743, 745–753, 755–757, 759–764, 766–770, 772–777, 780, 782, 784, 786–795, 797, 799–801, 803–814, 819, 820, 822–832

C/C++ pointer property

The property that a routine argument has a pointer type in C/C++ but is an array in Fortran. **535**, 554, 556, 574, 638–642, 644, 664–671, 673–676, 694

child task

A task is a child task of its generating task region. The region of a child task is not part of its generating task region, unless the child task is an included task. 30, 42, 51, 59, 96, 103, 108, 479, 502, 507, 508, 511, 559

chunk

A contiguous non-empty subset of the collapsed iterations of a loop-collapsing construct. 94, 134, 414, 418, 419, 421, 422, 429, 531, 574, 719, 754, 894

class type

For C++, the type of any variable declared with one of the `class`, `struct`, or `union` keywords. 217, 220, 222, 228–231, 244, 249, 254, 258, 271–274, 285, 287, 463

clause

A mechanism to specify customized directive behavior. xxvii, 4–6, 8, 9, 20–22, 24, 25, 31–33, 35, 39–50, 52, 54, 55, 57, 61, 68–71, 73, 76, 77, 79–82, 86, 87, 90–95, 101, 103, 109, 110, 116, 119, 122, 124–127, 129, 132, 143, 148–153, 157–165, 168–171, 174, 179, 181, 182, 203, 204, 206–208, 210–217, 220–231, 233–240, 244, 247–249, 251–254, 256–296, 298–301, 303, 304, 309–319, 321, 322, 324–367, 370–376, 378–380, 382, 383, 385, 387–389, 393–399, 401–407, 409, 414, 418–427, 429, 430, 432–445, 450–459, 461–464, 466, 468–472, 474, 479–502, 504–519, 521–523, 528–531, 534, 535, 561, 568, 570, 583, 586, 590, 599, 600, 604, 607, 608, 610, 645, 646, 652, 653, 655, 678, 715, 716, 741, 748, 760, 761, 783, 888–891, 897–902, 904–907, 909–914, 916–918

clause set

A set of clauses for which restrictions on their use or other properties of their use on a given directive are specified. **160**, 31, 33, 50, 92, 110, 160, 161, 210, 356, 363, 430

clause group

A clause set for which restrictions or properties related to their use on all directives are specified. **157**, 160, 343, 356, 363, 484, 488, 490, 517, 519, 900

clause-list trait

A trait that is defined with properties that match the clauses that may be specified for a given directive. 318, 319, 321

closely nested construct

A construct nested inside another construct with no other construct nested between them. 411, 413, 425, 522, 524

closely nested region

A region nested inside another region with no parallel region nested between them. 84, 257, 404, 425, 522, 524, 915

code block

A contiguous region of memory that contains code of an OpenMP program to be executed on a device. 453

collapsed iteration space

The logical iteration space of the collapsed loops of a loop-collapsing construct. **204**, 264, 267, 401, 415, 418, 421, 422

collapsed iteration

A logical iteration of the collapsed loops of a loop-collapsing construct. 31, 32, 35, 60, 67, 94, 113, 205, 220, 233, 234, 244, 258, 267, 268, 398, 399, 402, 404, 414, 418–423, 429, 502, 516, 531, 753, 754

collapsed logical iteration

A collapsed iteration. 204, 220

collapsed loop

For a loop-collapsing construct, a loop that is affected by the `collapse` clause. **4**, 32, 67, 104, 204, 205, 220, 233, 264, 400, 414, 419, 420, 423, 424, 433, 434, 516, 888, 901

collective step expression

An expression in terms of a step expression and a collector that eliminates recursive calculation in an induction operation. **60**, 32, 244

collector

A binary operator used to eliminate recursion in an induction operation. **60**, 32, 266

collector expression

An OpenMP stylized expression that evaluates to the value of the collective step expression of a collapsed iteration. **244**, 60, 244, 246, 264, 266

combined construct

A construct that is a shortcut for specifying one construct immediately nested inside a leaf construct. **530**, 21, 32, 34, 526, 911, 912

combined directive

A compound directive that is used to form a combined construct. 32, 34, 525

combined-directive name

The name of a combined directive. **525**

combiner

A binary operator used by a reduction operation. **249**, 90, 183, 252, 253

combiner expression

An OpenMP stylized expression that specifies how a reduction combines partial results into a single value. **240**, 90, 240, 241, 248, 251, 260–262, 267, 896

common-field property

The property that a field has a name that is used in more than one OpenMP type, or in more than one OMPD type, or in more than one OMPT type. 726, 727

common-type-callback property

The property that a callback has a type that at least one other callback has. 763, 764, 766–768, 838, 843

compatible context selector

A context selector that matches the OpenMP context in which a directive is encountered. **323**, 323–325, 329

compatible map type

A *map-type* that is consistent with the data-motion attribute of a given data-motion clause. 295, 298

compatible property

The property that a clause, an argument, a modifier, or a clause set does not have the exclusive property. 159

compilation unit

For C/C++, a translation unit. For Fortran, a program unit. 9, 44, 154, 218, 219, 289, 302, 311, 312, 314, 352, 355–357, 361, 368, 463, 608, 645, 646, 655

compile-time error termination

Error termination that is performed during compilation. **6**, 356, 389, 890

complete tile

A tile that has $\prod_k s_k$ logical iterations, where s_k are the list items of the **sizes** clause on the construct. **381**, 84

complex modifier

A modifier that may take at least one argument when it is specified. **158**, 33, 158, 161, 169

complex property

The property that a modifier is a complex modifier. 180, 470

compliant implementation

An implementation of the OpenMP specification that compiles and executes any conforming program as defined by the specification. A compliant implementation may exhibit unspecified behavior when compiling or executing a non-conforming program. **15**, 2, 5, 15, 17, 34, 42, 57, 110, 135, 136, 148, 419, 496, 533, 663, 697, 787, 816, 817, 891

composite construct

A construct that is a shortcut for composing a series or nesting of multiple constructs, but that does not have the semantics of a combined construct. 21, 267, 275, 531, 899, 902

composite directive

A directive that is composed of two (or more) directives but does not have identical semantics to specifying one of the directives immediately nested inside the other. A composite directive either adds semantics not included in the directives from which it is composed or provides an effective nesting of one directive inside the other that would otherwise be non-conforming. If the composite directive adds semantics not included in its constituent directives, the effects of the constituent directives may occur either as a nesting of the directives or as a sequence of the directives. 34, 458, 526, 527

composite-directive name

The directive name of a composite directive. **525**, 526, 527

compound construct

A construct that corresponds to a compound directive. 34, 61, 79, 82, 96, 174, 179, 254, 318, 516, 527–531, 898, 913, 918, 919

compound directive

A combined directive or a composite directive. 20, 32, 34, 35, 64, 160, 525, 528

compound-directive name

The directive name of a compound directive. **525**, 46, 525, 527, 902, 919

compound target construct

A compound construct for which **`target`** is a constituent construct. 276, 277, 529

conceptual abstract name

An abstract name that refers to an implementation defined abstraction that is relevant to the execution model described by this specification. **128**, 19, 77, 85, 128

conditional-update-capture structured block

An update structured block that may be associated with an **`atomic`** directive that expresses an atomic conditional update operation with capture semantics. **192**, 192, 193, 497

conditional-update structured block

An update structured block that may be associated with an `atomic` directive that expresses an atomic conditional update operation. **191**, 191, 192, 497

conforming device number

A device number that may be used in a conforming program. **7**, 141, 305, 321, 322, 452, 547, 592, 599–603, 631, 647, 690

conforming program

An OpenMP program that follows all rules and restrictions of the OpenMP specification. 2, 15, 34, 35, 76, 79, 110, 324, 371, 419

C-only property

The property that an OpenMP feature is only supported in C. 697, 712, 820, 825, 827, 828, 834, 835, 837–849, 851–869, 871–873, 875–877

consistent schedules

The loop schedules of two affected loop nests are consistent if for each assignment of a thread to a collapsed iteration that results from the schedule of one loop nest, the behavior is as if the same thread is assigned to the corresponding collapsed iteration of the other loop nest. **205**, 35, 205, 404

constant property

The property that an expression, including one that is used as the argument of a clause, a modifier or a routine, is a compile-time constant. **161**, 53, 93, 151, 160, 162, 163, 181–183, 204, 206, 207, 270, 304, 309, 311, 313, 317, 321, 322, 343, 344, 350, 354, 357–362, 365–367, 373, 376, 379, 382, 383, 401, 439, 440, 443, 484–492, 517–519, 534, 900

constituent construct

For a given construct, a construct that corresponds to one of the constituent directives of the executable directive. 21, 34, 79, 96, 174, 179, 254, 363, 515, 527–529, 902

constituent directive

For a given directive and its set of leaf directives, a leaf directive in the set or a compound directive that is a shortcut for composing two or more members of that set for which the directive names are consecutively listed. 34, 35, 160, 174, 275, 458, 459, 528, 531, 898

constituent-directive name

The directive name of a constituent directive. **525**, 525, 531, 919

construct

An executable directive, its paired end directive (if any), and the associated structured block (if any), not including the code in any called procedures. That is, the lexical extent of an executable directive. 2–7, 15, 19, 21, 22, 24, 25, 28, 29, 31–37, 40, 42, 43, 45, 46, 48–59, 61, 63, 64, 68, 69, 74–77, 79, 81–84, 86, 87, 90–97, 99–101, 103–107, 110, 111, 113, 114, 116, 117, 120, 122, 124, 125, 132–134, 139, 149, 150, 152, 155, 156, 161, 164, 169, 171, 174, 179, 181–183, 192, 193, 201, 204, 205, 207, 210–214, 216, 217, 219–231, 233, 235–238, 240, 248, 250–254, 257, 259, 264, 267, 268, 273–277, 280–287, 292, 295, 309, 310, 313, 315–318, 328, 332, 334, 338–342, 357–359, 363, 364, 366, 373, 375, 377–382, 384–386, 388, 394–399, 402–413, 416–427, 429–431, 433–437, 439–445, 449–459, 461–466, 468, 469, 472–476, 478–480, 482–506, 508, 509, 511–517, 519–525, 527–531, 561, 583–585, 601–603, 692, 698, 705, 706, 719, 725, 733, 734, 741, 745, 748, 753, 757–761, 770, 772, 783, 828, 829, 880, 881, 889–891, 898–902, 904–907, 909–919

`construct` selector set

A selector set that may match the construct trait set. **321**, 318, 321–323, 329, 330

construct trait set

The trait set that, at a given point in an OpenMP program, consists of all enclosing constructs up to an enclosing **`target`** construct. **318**, 36, 37, 318, 319, 321, 323, 341

containing array

For C/C++, a non-subscripted array (a containing array) to which a series of zero or more array subscript operators and, when specified, dot (i.e., '.') operators are applied to yield a given lvalue expression or array section for which storage is contained by the array. For Fortran, an array (a containing array) without the **POINTER** attribute and without a subscript list to which a series of zero or more array subscript selectors and, when specified, component selectors are applied to yield a given variable or array section for which storage is contained by the array.

> COMMENT: An array is a containing array of itself. For the array section (*p0).x0[k1].p1->p2[k2].x1[k3].x2[4][0:n], where identifiers p*i* have a pointer type declaration and identifiers x*i* have an array type declaration, the containing arrays are: (*p0).x0[k1].p1->p2[k2].x1 and (*p0).x0[k1].p1->p2[k2].x1[k3].x2.

26, 27, 36, 165, 283, 286

containing structure

For C/C++, a structure to which a series of zero or more . (dot) operators and/or array subscript operators are applied to yield a given lvalue expression or array section for which storage is contained by the structure. For Fortran, a structure to which a series of zero or more component selectors and/or array subscript selectors are applied to yield a given variable or array section for which storage is contained by the structure.

> COMMENT: A structure is a containing structure of itself. For C/C++, a structure pointer *p* to which the -> operator applies is equivalent to the application of a . (dot) operator to (**p*) for the purposes of determining containing structures.

For the array section (*p0).x0[k1].p1->p2[k2].x1[k3].x2[4][0:n], where identifiers p*i* have a pointer type declaration and identifiers x*i* have an array type declaration, the containing structures are: *(*p0).x0[k1].p1, (*(*p0).x0[k1].p1).p2[k2] and (*(*p0).x0[k1].p1).p2[k2].x1[k3]

27, 36, 37, 212, 279, 282, 283, 286, 287

contention group

All implicit tasks and their descendent tasks that are generated in an implicit parallel region, *R*, and in all nested regions for which *R* is the innermost enclosing implicit parallel region. 3–6, 21, 28, 64, 81, 94, 100, 116, 117, 130, 134, 141, 301, 306, 360, 387, 393, 453, 473, 494, 534, 535, 571, 584, 585, 601, 602, 663, 891, 899, 907

context-matching construct

A construct that has the context-matching property. 321

context-matching property

The property that a directive adds a trait of the same name to the construct trait set of the current OpenMP context. 37, 337, 384, 394, 399, 416, 417, 460

context selector

The specification of traits that a directive variant or function variant requires in the current OpenMP context in order for that variant to be selected. **320**, 33, 48, 98, 320–325, 328, 329, 331, 335–337, 355, 889, 906

context-specific structured block

Structured blocks that conform to specific syntactic forms and restrictions that are required for certain block-associated directives. **186**, 21, 25, 54, 187, 188

core

A physically indivisible hardware execution unit on a device onto which one or more hardware threads may be mapped via distinct execution contexts. 63, 76, 98, 128, 726

corresponding list item

For a privatization clause, a new list item that derives from an original list item. For a data-mapping attribute clause, a list item in a device data environment that corresponds to an original list item. 68, 69, 231, 238, 239, 273, 280, 282–289, 295, 296, 298, 316, 346, 361, 461, 466, 610, 899

corresponding pointer

For a given pointer variable or a given referring pointer, the corresponding variable or handle that exists in a device data environment. 82, 284, 287, 288

corresponding pointer initialization

For a given data entity that has a base pointer or referring pointer, an assignment to the base pointer or referring pointer such that any lexical reference to the data entity or a subobject of the data entity in a **target** region refers to its corresponding data entity or subobject in the device data environment. 284, 461

corresponding storage

For a given storage block, its corresponding storage block. For a given mapped variable, the corresponding storage of its original storage block. 38, 70, 84, 95, 236, 275, 281, 282, 284–287, 296, 463, 605, 739, 891

corresponding storage block

A storage block that contains the storage of one or more variables in a device data environment that corresponds to mapped variables in an original storage block. 8, 9, 38, 69, 283, 284

C pointer

For C/C++, a base language pointer variable. For Fortran, a variable of type **C_PTR**. 45, 111, 236

current device

The device on which the current task is executing. 8, 10, 21, 23, 45, 57, 72, 102, 145, 319, 435, 451, 535, 577, 580, 583, 630, 647, 654, 655, 683–685, 789, 800

current task

For a given thread, the task corresponding to the task region that it is executing. 38, 49, 57, 280, 305, 332, 478, 479, 568, 570–573, 576, 577, 580, 589, 592, 593, 598, 678

current task region

The region that corresponds to the current task. 5, 104, 399, 427, 446, 475, 479, 520, 521, 860

current team

All threads in the team executing the innermost enclosing parallel region. 28, 82, 94, 104, 106, 117, 214, 399, 402, 403, 405–407, 409, 414, 435, 442, 446, 475, 478, 479, 514, 515, 520, 524, 579, 590, 733

current team tasks

All tasks encountered by the corresponding team. The implicit tasks constituting the parallel region and any descendent tasks encountered during the execution of these implicit tasks are included in this set of tasks. 28, 306

D

data-copying property

The property that a clause copies a list item from one data environment to other data environments. 271, 272

data entity

For C/C++, a data object that is referenced by a given lvalue expression or array section. For Fortran, a data entity as defined by the base language. 25–27, 38–40, 44, 55, 56, 58, 60, 87, 90, 96, 106, 111, 328

data environment

The variables associated with the execution of a given region. 4, 6, 8, 9, 21, 39, 40, 43, 48, 57, 70, 73, 76, 82, 102, 115–117, 121, 124, 125, 210, 236, 257, 273, 280, 295, 426, 429, 436, 445, 454, 456, 461, 466, 603, 799, 904, 914

data-environment attribute

A data-sharing attribute or a data-mapping attribute. 39, 210

data-environment attribute clause

A clause that explicitly determines the data-environment attributes of the list items in its list argument. **210**, 39, 215, 292, 347, 401, 436, 437, 445

data-environment attribute property

The property that a clause is a data-environment clause. 224, 225, 227, 229, 232, 235–238, 252, 255–257, 279, 289, 290, 299, 300, 303, 315

data-environment clause

A clause that is a data-environment attribute clause or otherwise affects the data environment. **210**, 39, 210, 444

data-mapping attribute

The relationship of a data entity in a given device data environment to the version of that entity in the enclosing data environment. **210**, 39, 51, 58, 213, 276, 292, 910

data-mapping attribute clause

A clause that explicitly determines the data-mapping attributes of the list items in its list argument. **210**, 8, 37, 40, 51, 76, 276, 289, 316, 454, 456, 461, 898

data-mapping attribute property

The property that a clause is a data-mapping clause. 279, 289

data-mapping clause

A clause that is a data-mapping attribute clause or otherwise affects the data environment of the target device. **210**, 39, 70, 210

data-mapping construct

A construct that has the data-mapping property. 48, 69, 212, 275, 283, 284, 459

data-mapping property

The property of a construct on which a data-mapping attribute clause may be specified. 40, 454, 456, 458, 460

data-motion attribute

The data-movement relationship between a given device data environment and the version of that data entity in the enclosing data environment. 33, 295

data-motion attribute property

The property that a clause is a data-motion clause. 297, 298

data-motion clause

A clause that specifies data movement between a device set that is specified by the construct on which it appears. 23, 33, 40, 278, 293–296, 298, 466, 906

data race

A condition in which different threads access the same memory location such that the accesses are unordered and at least one of the accesses is a write. Data races produce unspecified behavior. **8**, 2, 8, 9, 13, 14, 40, 225, 227, 231, 251, 259, 273, 284, 296, 308, 402, 420, 496

data-sharing attribute

For a given data entity in a data environment, an attribute that determines the scope in which the entity is visible (i.e., its name provides access to its storage) and/or the lifetime of the entity. A variable that is part of an aggregate variable cannot have a particular data-sharing attribute independent of the other components, except for static data members of C++ classes. **210**, 39, 40, 44, 51, 52, 55, 56, 58, 60, 62–64, 86, 87, 90, 96, 106, 111, 210, 212–214, 222, 224, 276, 292, 454, 456, 458, 461, 466, 528, 888, 910

data-sharing attribute clause

A clause that explicitly determines the data-sharing attributes of the list items in its list argument. **210**, 7, 21, 41, 51, 82, 160, 210, 213, 219, 221–223, 225, 239, 313, 316, 424, 426, 429, 461, 463, 530, 898, 912

data-sharing attribute property

The property that a clause is a data-sharing clause. 224, 225, 227, 229, 232, 235–238, 252, 255–257, 315, 445

data-sharing clause

A clause that is a data-sharing attribute clause. **210**, 41, 210, 212, 213

declaration-associated directive

A declarative directive for which its associated base language code is a procedure declaration. **153**, 152–155, 334, 341, 347, 348, 900

declaration sequence

For C/C++, a sequence of base language declarations, including definitions, that appear in the same scope. The sequence may include other directives that are associated with the declarations. 336, 349, 369

declarative directive

A directive that may only be placed in a declarative context and results in one or more declarations only; it is not associated with the immediate execution of any user code or implementation code. 41, 51, 60, 112, 152, 153, 155, 156, 161, 215, 260, 263, 293, 301, 310, 334, 336, 341, 346, 349, 363, 450, 897

declare target directive

A declarative directive that has the declare-target property. 8, 69, 76, 212, 240, 276, 287, 301, 318, 345–347, 349, 351, 356, 360, 361, 461, 463, 564, 889, 904, 910

declare-target property

The property that a directive applies to procedures and/or variables to ensure that they can be executed or accessed on a device. 41, 346, 349

declare variant directive

A declarative directive that declares a function variant for a given base function. 48, 318, 328–330, 336, 338, 889, 906, 910

default mapper

The mapper that is used for a **`map`** clause for which the *mapper* modifier is not explicitly specified. 86, 278

defined

For variables, the property of having a valid value. For C, for the contents of variables, the property of having a valid value. For C++, for the contents of variables of POD (plain old data) type, the property of having a valid value. For variables of non-POD class type, the property of having been constructed but not subsequently destructed. For Fortran, for the contents of variables, the property of having a valid value. For the allocation or association status of variables, the property of having a valid status.

COMMENT: Programs that rely upon variables that are not defined are non-conforming programs.

42, 109, 131, 146, 916

delimited directive

A directive for which the associated base language code is explicitly delimited by the use of a required paired end directive. **154**, 152, 155, 327, 336, 349, 369

dependence

An ordering relation between two instances of executable code that must be enforced by a compliant implementation. **504**, 42, 47, 103, 181, 435, 504–509, 512, 514, 515, 604, 715, 756, 761, 762

dependence-compatible task

Two tasks between which a task dependence may be established. **507**, 86, 103, 108, 504, 508, 509, 511, 559

dependent task

A task that because of a task dependence cannot be executed until its antecedent tasks have completed. **507**, 22, 100, 103, 448, 458, 480, 502–504, 507–509, 604, 741, 762

depend object

An OpenMP object that supplies user-computed dependences to **`depend`** clauses. **558**, 181, 435, 481, 505, 506, 508, 509, 604, 760, 761, 911

deprecated

For a construct, clause, or other feature, the property that it is normative in the current specification but is considered obsolescent and will be removed in the future. Deprecated features may not be fully specified. In general, a deprecated feature was fully specified in the version of the specification immediately prior to the one in which it is first deprecated. In most cases, a new feature replaces the deprecated feature. Unless otherwise specified, whether any modifications provided by the replacement feature apply to the deprecated feature is implementation defined. 42, 156, 157, 260, 533, 603, 710, 713, 737, 778, 781, 783, 784, 885, 896, 903–905, 907, 909, 911

descendent task

A task that is the child task of a task region or of a region that corresponds to one of its descendent tasks. 37, 38, 42, 430, 448, 502, 521

detachable task

An explicit task that only completes after an associated event variable that represents an *allow-completion* event is fulfilled and execution of the associated structured block has completed. **445**, 426, 437, 502, 503, 538, 590, 910

device

An implementation-defined logical execution engine.

COMMENT: A device could have one or more processors.

3, 4, 7–9, 19, 21–23, 28, 32, 37, 38, 40, 41, 43–46, 48, 53, 56, 59, 71, 75, 76, 79, 83, 84, 98, 100, 102, 103, 109, 115–117, 124, 127, 128, 139–141, 145, 181, 237, 274, 280, 289, 290, 295, 296, 303, 306–308, 318, 319, 321, 323, 332, 345, 346, 359, 360, 436, 450, 453, 455, 457, 461–463, 466, 494, 536, 564, 571, 590, 592, 594–597, 599–603, 605–607, 610–613, 618, 619, 630–634, 645, 647–652, 654, 655, 683, 689, 690, 692, 704–706, 708, 710, 711, 717, 722, 726, 744, 772–776, 778, 779, 785–787, 793, 800, 801, 803–810, 812–814, 819, 822, 826, 833, 836, 842, 846, 850–853, 857, 879, 883, 885, 889, 891, 894, 897, 899, 900, 902, 903, 906, 907, 909, 911–913

device address

An address of an object that may be referenced on a target device. **8**, 8, 45, 62, 111, 235–238, 328, 332, 359, 360, 607, 885, 906, 909

device-affecting construct

A construct that has the device-affecting property. 462, 600, 602, 917

device-affecting property

The property that a device construct can modify the state of the device data environment of a specified target device. 43, 454, 456, 458, 460, 465

device-associated property

The property of a clause that a device must be associated with the construct on which it appears. 235–238

device construct

A construct that has the device property. 2, 43–45, 56, 62, 102, 111, 141, 286, 355, 356, 451, 736, 760, 781, 785, 908, 913

device data environment

The initial data environment associated with a device. **8**, 8, 9, 25, 37–40, 43, 56, 68–72, 84, 87, 111, 124, 210, 235–239, 257, 274, 275, 280–290, 295, 296, 332, 345, 361, 454, 456, 461, 463, 464, 466, 599, 601, 602, 605, 607, 608, 610, 612, 618, 779, 885, 898, 902

device global requirement clause

A *requirement* clause that has the device global requirement property. 355

device global requirement property

The property that a *requirement* clause indicates requirements for the behavior of device constructs that a program requires the implementation to support across all compilation units. 44, 356, 358–362

device-information property

The property of a routine that it provides or modifies information about a specified device that supports use of the device in an OpenMP program. **592**, 44, 592–599, 601

device-information routine

A routine that has the device-information property. **592**, 592

device-local attribute

For a given device, a data-sharing attribute of a data entity that it has static storage duration and is visible only to tasks that execute on that device. **303**, 44, 211, 214

device-local variable

A variable that has the device-local attribute with respect to a given device. **303**, 8, 286, 345, 361, 885

device-memory-information routine

A routine that has the device-memory-information routine property. **604**, 603

device-memory-information routine property

The property of a device memory routine that it enables operations on memory that is associated with the specified devices but does not itself directly operate on that memory. **604**, 44, 604–606

device memory routine

A device routine that has the device memory routine property. **603**, 44, 102, 564, 603, 604, 779, 888, 913

device memory routine property

The property that a device routine operates on or otherwise enables operations on memory that is associated with the specified devices. **603**, 44, 604–606, 608, 609, 611, 613–615, 617, 619, 620

device number

A number that the OpenMP implementation assigns to a device or otherwise may be used in an OpenMP program to refer to a device. **7**, 7, 35, 115, 116, 119, 120, 127, 139–141, 308, 451, 461, 542, 593, 594, 596, 598–602, 610, 612, 689, 692, 773–775, 779, 781, 800, 902

device pointer

An implementation defined handle that refers to a device address and is represented by a C pointer. 8, 63, 111, 235, 236, 328, 332, 359, 604, 606–608, 610–613, 654, 885, 907

device procedure

A procedure that can be executed on a target device, as part of a **`target`** region. 102, 291, 345, 355, 356, 360, 361

device property

The property of a construct that it accepts the **`device`** clause. 43, 346, 349, 454, 456, 458, 460, 465, 468

device region

A region that corresponds to a device construct. 715, 722, 744, 778, 781, 783, 785

device routine

An OpenMP API routine that may require access to one or more specified devices. 24, 44, 141

`device` selector set

A selector set that may match the device trait set. **321**, 321–323

device-specific environment variable

An alternative OpenMP environment variable that controls the behavior of the program only with respect to a particular device or set of devices. **119**, 120, 127, 139, 906

device-tracing callback

A callback that has the device-tracing property. **772**

device-tracing entry point

An entry point that has the device-tracing property. 772, 773

device-tracing property

The property that an entry point or callback is part of the OMPT tracing interface and, so, is used to control the collection of trace records on a device. **772**, 45, 772–777, 780, 782, 784

device trait set

The trait set that consists of traits that define the characteristics of the device that the compiler determines will be the current device during program execution at a given point in the OpenMP program. **319**, 45, 318, 319

device-translating callback

A callback that has the device-translating property. **842**, 843, 844

device-translating property

The property that a callback translates data between the formats used for the device on which the third-party tool and OMPD library run and the device on which the OpenMP program runs. **842**, 46, 843

directive

A base language mechanism to specify OpenMP program behavior. 2, 3, 6–9, 13, 15, 17, 21, 22, 24, 25, 29–31, 33–35, 37, 41, 42, 46–50, 54, 57, 60, 64, 68, 69, 73, 79, 80, 89, 91, 95, 100, 103, 107, 109, 111, 112, 114, 116, 127, 143, 148–157, 159–166, 168, 171, 174, 182, 183, 187, 188, 190–192, 198, 201–208, 210, 211, 213–215, 217–220, 222, 225, 230, 233, 234, 247–249, 254, 257, 260, 261, 263–265, 267–270, 276, 278, 280, 283, 284, 289–294, 300–304, 306, 307, 311–316, 318, 319, 321, 322, 324–328, 335–339, 341–343, 345, 347–357, 359, 362, 363, 368–373, 379, 382, 385, 388, 389, 395, 399, 402, 409–411, 424, 426, 429, 431, 434, 445, 451, 452, 454–458, 461, 463, 465, 466, 469, 470, 474, 481–483, 488, 496, 500–503, 505, 519, 523, 524, 527, 535, 561, 564, 608, 645, 646, 652, 653, 655, 663, 744, 748, 887–890, 896–902, 904–907, 909, 910, 912, 914, 915, 917

directive name

The name of a directive or a corresponding construct. 34, 35, 46, 47, 64, 150, 162, 173, 174, 179, 180, 182, 206, 223, 225–227, 230, 232, 235–238, 252, 255, 256, 258, 262, 263, 265, 266, 269–272, 280, 289–291, 297–300, 303, 309, 310, 313, 316, 325, 326, 330, 331, 333, 339, 340, 344, 350, 353, 354, 357–367, 372, 374, 376, 378, 382, 383, 388, 392, 393, 397, 398, 400–403, 418, 422, 425, 432, 433, 439–445, 450–452, 470, 472, 481, 483–489, 491–493, 506, 507, 511, 512, 517–519, 525, 527

directive-name list

An argument list that consists of directive-name list items. **162**

directive-name list item

A list item that is a directive name. **162**, 46

directive-name separator

Characters used to separate the directive names of leaf constructs in a compound-directive name. A directive-name separator is either white space or, in Fortran, a plus sign (i.e., '+'); a given instance of a compound-directive name must use the same character for all directive-name separators. **525**, 46, 525–527

directive specification

The directive specifier and list of clauses that specify a given directive. **150**, 47, 150, 162

directive-specification list

An argument list that consists of directive-specification list items. **162**

directive-specification list item

A list item that is a directive specification. **162**, 47, 164

directive specifier

The directive name and, where permitted, the directive arguments that are specified for a given directive. **150**, 46

directive variant

A directive specification that can be used in a metadirective. **324**, 37, 92, 324–327, 910

divergent threads

Two threads are divergent if one executes a diverging code path and the other does not due to a conditional statement. 7, 47, 362

diverging code path

For a given pair of threads, the region of a structured block sequence that is executed by only one of the threads. 6, 47

doacross-affected loop

For a worksharing-loop construct in which a stand-alone `ordered` directive is closely nested, a loop that is affected by its `ordered` clause. 48, 207, 371, 514, 900

doacross dependence

A dependence between executable code corresponding to stand-alone `ordered` regions from two doacross iterations: the sink iteration and the source iteration, where the source iteration precedes the sink iteration in the doacross iteration space. The doacross dependence is fulfilled when the executable code from the source iteration has completed. **504**, 47, 98, 512, 514, 715

doacross iteration

A logical iteration of a doacross loop nest. 47, 98, 503, 504, 512, 514

doacross iteration space

The logical iteration space of a doacross loop nest. 47, 512

doacross logical iteration

A doacross iteration. 512

doacross loop nest

The doacross-affected loops of a worksharing-loop construct in which a stand-alone `ordered` construct is closely nested. 47, 512, 514, 912, 913

dynamic context selector

Any context selector that is not a static context selector. 337

dynamic replacement candidate

A replacement candidate that may be selected at runtime to replace a given metadirective. **324**, 324, 325, 329

dynamic storage duration

For C/C++, the lifetime of an object with dynamic storage duration, as defined by the base language. For Fortran, the lifetime of a data object that is dynamically allocated with the `ALLOCATE` statement or some other language mechanism. 211, 214

dynamic trait set

The trait set that consists of traits that define the dynamic properties of an OpenMP program at a given point in its execution. **319**, 111, 318, 320, 321

E

effective context selector

The resulting context selector that must be satisfied for a given function variant to be selected, as determined by the `match` clauses of all `begin declare_variant` directives that delimit a base language code region that encloses the declare variant directive. **336**, 336, 337

effective `map` clause set

The set of all `map` clauses that apply to a data-mapping construct, including any implicit `map` clauses and `map` clauses applied by mappers. **283**, 283, 284

enclosing context

For C/C++, the innermost scope enclosing a directive. For Fortran, the innermost scoping unit enclosing a directive. 48, 73, 82, 96, 213, 214, 252–254, 259, 261, 264, 273, 324, 340, 341, 408, 410, 413, 421, 910

enclosing data environment

For a given directive, the data environment of its enclosing context. 39, 40, 52, 56, 63, 94, 111, 436, 437

encountering device

For a given construct, the device on which the encountering task of the construct executes. 236, 284, 295, 298, 463, 891

encountering task

For a given region, the current task of the encountering thread. 6, 48, 49, 103, 295, 334, 339, 352, 385, 394, 395, 414, 427, 431, 436, 442, 445, 462, 468, 476, 477, 480, 482, 520–522, 535, 579, 587, 588, 670, 673–677, 706, 744, 759, 760, 781, 798, 799, 880, 881

encountering-task binding property

The binding property that the binding thread set is the encountering task. **534**

encountering thread

For a given region, the thread that encounters the corresponding construct, structured block sequence, or routine. 4, 5, 28, 49, 59, 91, 252, 384, 389–391, 394, 403, 423, 425–427, 461, 469, 499, 505, 535, 578, 579, 589, 594, 681, 683, 685–687, 689, 695, 725, 770, 786, 791, 793–795, 798, 800, 902

encountering-thread binding property

The binding property that the binding thread set is the encountering thread. **534**

end-clause property

The property that a clause may appear on an end directive. **150**, 272, 481

end directive

For a given directive, a paired directive that lexically delimits the code associated with that directive. **150**, 35, 42, 49, 150, 152, 153, 155, 156, 160, 187, 188, 192, 327, 336, 337, 349, 350, 474, 904, 905

ending address

The address of the last storage location of a list item or, for a mapped variable, of its original list item. 51, 70, 281

entry point

A runtime entry point. 24, 45, 79, 700, 701, 703–706, 711, 720, 722, 729, 745, 772, 773, 776, 786–814, 894, 895, 903

enumeration

A type or any variable of a type that consists of a specified set of named integer values. For C/C++, an enumeration type is specified with the `enum` specifier. For Fortran, an enumeration type is specified by either (1) a named integer constant that is used as the integer kind of a set of named integer constants that have unique values or (2) a C-interoperable enumeration definition. 49, 536, 539–541, 544, 547, 550, 554, 557–560, 562, 563, 565, 566, 711, 714, 716, 717, 720, 722–725, 727–731, 735, 736, 738–741, 743, 789, 825, 827, 828, 874

environment variable

Unless specifically stated otherwise, an OpenMP environment variable. 2, 6, 118, 119, 127–137, 139–147, 692, 693, 872, 886, 887, 896, 897, 906, 908, 909, 912–915

error termination

A `fatal` action preformed in response to an error. **6**, 33, 93, 389, 900

event

A point of interest in the execution of a thread or a task. 10, 11, 14, 15, 29, 43, 91, 102, 108, 250, 286, 346, 352, 385, 386, 394, 395, 403, 405–411, 413–415, 421, 426, 427, 430, 431, 437, 445–447, 449, 453, 455–457, 459, 462, 466, 474–478, 480, 496, 497, 500, 502, 503, 509, 513, 515, 516, 521, 522, 538, 586, 589, 590, 603, 604, 607–616, 618–621, 664–669, 671–677, 695, 697, 700, 703–705, 710, 726, 728–730, 741, 744, 746, 757–759, 761, 763–765, 767, 771, 772, 776, 778, 781, 783, 784, 786, 789, 790, 796, 805, 806, 808, 812, 813, 816, 878, 880, 881, 883, 894, 902

exception-aborting directive

A directive that has the exception-aborting property. 366, 887

exception-aborting property

For C++, the property of a directive that whether an exception that occurs in its associated region is caught or results in a runtime error termination is implementation defined. 50, 149, 460

exclusive property

The property that a clause, an argument, or a modifier may not be specified when, (respectively), a different clause, argument or modifier is specified. When applied to a clause set, the property applies only to clauses within that set. **160**, 33, 159–161, 232, 266, 313, 343, 381, 405, 426, 429, 484, 488, 519

exclusive scan computation

A scan computation for which the value read does not include the updates performed in the same logical iteration. **270**, 270, 909

executable directive

A directive in an executable context that results in implementation code or prescribes the manner in which any associated user code must execute. 3, 35, 36, 60, 64, 67, 68, 98, 100, 112, 149, 152, 153, 155, 186, 198, 315, 324, 337, 352, 353, 374, 375, 377, 379–381, 384, 394, 399, 402, 405–407, 409, 412, 416, 417, 420, 423, 426, 429, 435, 446, 454, 456, 458, 460, 465, 468, 473, 475, 478, 479, 494, 498, 505, 514, 515, 520, 524

explicit barrier

A barrier that is specified by a **`barrier`** construct. **475**

explicitly associated directive

A declarative directive for which its associated base language declarations are explicitly specified in a variable list or extended list argument. **153**, 152, 153, 155, 215, 301, 310, 346

explicitly determined data-mapping attribute

A data-mapping attribute that is determined due to the presence of a list item on a data-mapping attribute clause. 274

explicitly determined data-sharing attribute

A data-sharing attribute that is determined due to the presence of a list item on a data-sharing attribute clause. **213**, 210, 213, 224

explicit region

A region that corresponds to either a construct of the same name or a library routine call that explicitly appears in the program. **3**, 3, 99, 149, 413, 446, 689, 802

explicit task

A task that is not an implicit task. **5**, 5, 7, 26, 30, 43, 51, 53, 83, 94, 103, 104, 116, 253, 254, 385, 389, 426, 427, 429–431, 447, 475, 503, 524, 586, 689, 719, 756, 798, 864, 910, 913, 916

explicit task region

A region that corresponds to an explicit task. 8, 91, 225, 427, 527, 587, 903

exporting task

A task that permits one of its child tasks to be an antecedent task of a task for which it is a preceding dependence-compatible task. **511**, 108, 427, 437, 508, 511, 559

extended address range

For a given original list item, the address range that starts from the minimum of its starting address and its base address and ends with maximum of its ending address and its base address. **280**, 71, 281

extended list

An argument list that consists of extended list items. **162**, 51

extended list item

A variable list item or the name of a procedure. **162**, 51, 164

extension trait

A trait that is implementation defined. **319**, 318

F

finalized taskgraph record

A taskgraph record in which all information required for a replay execution has been saved. **436**, 71, 436

final task

A task that generates included final tasks when it encounters task-generating constructs on which the **`final`** clause may be specified. **441**, 52, 116, 427, 436, 437, 439, 441, 442, 445, 588, 915

first-party tool

A tool that executes in the address space of the program that it is monitoring. **697**, 14, 29, 78, 144, 695, 697, 699, 903, 911

firstprivate attribute

For a given construct, a data-sharing attribute of a variable that implies the private attribute, and additionally the variable is initialized with the value of the variable that has the same name in the enclosing data environment of the construct. **227**, 52, 211–214, 277, 292, 436, 461, 904, 912

firstprivate variable

A private variable that has the firstprivate attribute with respect to a given construct. 430, 437, 891

flat-memory-copying property

The property that a memory-copying routine copies a unidimensional, contiguous storage block. **612**, 52, 613, 615

flat-memory-copying routine

A routine that has the flat-memory-copying property. **612**, 612, 614, 616

flush

An operation that a thread performs to enforce consistency between its view of memory and the view of memory of any other threads. 6, 10–14, 19, 52, 58, 92, 99, 107, 404, 472, 494, 499–501, 908, 915

flush property

A property that determines the manner in which a flush enforces memory consistency. Any flush has one or more of the following: the strong flush property, the release flush property, and the acquire flush property. 11, 908

flush-set

The set of variables upon which a strong flush operates. **10**, 10

foreign execution context

A context that is instantiated from a foreign runtime environment in order to facilitate execution on a given device. 53, 181, 468, 469, 542, 907

foreign runtime environment

A runtime environment that exists outside the OpenMP runtime with which the OpenMP implementation may interoperate. 53, 62, 86, 468, 471, 539, 542

foreign runtime identifier

A base language string literal or a constant expression of integer OpenMP type that represents a foreign runtime environment. **183**, 469, 471, 891, 902

foreign task

An instance of executable code that is executed in a foreign execution context. 181, 437, 469, 891

Fortran-only property

The property that an OpenMP feature is only supported in Fortran. 534

frame

A storage area on the stack of a thread that is associated with a procedure invocation. A frame includes space for one or more saved registers and often also includes space for saved arguments, local variables, and padding for alignment. 30, 53, 719–721, 744, 798, 824, 864, 865

free-agent thread

An unassigned thread on which an explicit task is scheduled for execution or a primary thread for an explicit parallel region that was a free-agent thread when it encountered the **`parallel`** construct. 53, 100, 107, 116, 132, 142, 143, 389, 390, 448, 588, 589, 734, 890, 897, 902

free property

The property that a modifier can appear in any position in a *modifier-specification-list*. 159

function

A routine or procedure that returns a type that can be the right-hand side of a base language assignment operation. 155, 156, 163, 311, 332, 337, 569–572, 575–579, 581–584, 586–588, 593–599, 601, 604–606, 609, 611, 613–615, 617, 619, 620, 623–628, 631–636, 642–644, 647–651, 653, 656–660, 675, 676, 678, 679, 681, 684, 686, 688–691, 694, 697, 745, 770, 786–795, 797, 799–801, 803–806, 808–814, 834, 835, 837–849, 851–869, 871–873, 875–877

function dispatch

A base function call for which variant substitution may be controlled. 187

function-dispatch structured block

A context-specific structured block that may be associated with a **dispatch** directive. **187**, 187, 188, 318, 331, 333, 337, 338

function variant

A definition of a procedure that may be used as an alternative to the base language definition. 37, 41, 48, 92, 113, 318, 328–336, 338, 340, 468, 906, 910

G

generally-composable property

The property of a loop-transforming construct that it may use directives other than loop-transforming directives in its **apply** clauses. 373, 377, 381

generated loop

A loop that is generated by a loop-transforming construct and is one of the resulting loops that replace the construct. **371**, 55, 59, 77, 107, 197, 203, 205, 371–373, 375, 378, 379, 381, 382, 431, 900

generated loop nest

A canonical loop nest that is generated by a loop-transforming construct. 371, 372

generated loop sequence

A canonical loop sequence that is generated by a loop-transforming construct. 371

generated task

The task that is generated as a result of the generating task encountering a task-generating construct. 5, 124, 213, 426, 427, 429, 430, 434, 439, 440, 442, 444, 468, 469, 479, 480, 482, 508, 509, 511, 756, 760, 761

generating task

For a given region, the task for which execution by a thread generated the region. 28, 54, 55, 124, 338, 427, 454, 456, 458, 461, 465, 468, 503, 603, 861

generating-task binding property

The binding property that the binding task set is the generating task. **603**, 606, 608, 609, 611, 613–615, 617, 619, 620

generating task region

For a given region, the region that corresponds to its generating task. 30, 59, 109, 861

global

A program aspect such as a scope that covers the whole OpenMP program. 57, 115–117, 119, 127, 311, 912

grid loop

The generated loops of a **`tile`** or **`stripe`** construct that iterate over cells of a grid superimposed over the logical iteration space, with spacing determined by the **`sizes`** clause. 77, 379–381, 889, 901

groupprivate attribute

For a given group of tasks, a data-sharing attribute of a data entity that it has static storage duration and is visible only to those tasks. **301**, 55, 211, 214, 301, 303

groupprivate variable

A variable that has the groupprivate attribute with respect to a given group of tasks. **301**, 286, 302, 303, 345, 347, 349, 413, 461

H

handle

An opaque reference that uniquely identifies an abstraction. 20, 37, 45, 55, 74, 75, 79, 83, 91, 95, 103, 113, 181, 287, 305, 306, 547, 630, 635–637, 645–647, 653, 655, 710, 711, 721, 795, 820, 826, 827, 829, 831–834, 841, 849, 850, 852–855, 858–865, 867, 871, 875, 876, 885, 895

handle-comparing property

The property that a routine compares two handle arguments. **865**, 55, 865–867

handle-comparing routine

A routine that has the handle-comparing property. **865**, 865, 895

handle property

The property that a type is used to represent handles. **830**, 56, 820, 829, 831, 832

handle-releasing property

The property that a routine releases a handle. **867**, 55, 867–869

handle-releasing routine

A routine that has the handle-releasing property. **867**, 867

handle type

An OpenMP type, OMPD type, or OMPT type that has the handle property. **830**

happens before

For an event A to happen before an event B, A must precede B in happens-before order. **12**, 13

happens-before order

An asymmetric relation that is consistent with simply happens-before order and, for C/C++, the "happens before" order defined by the base language. 13, 56, 307, 308, 360, 469, 908

hard pause

An instance of a resource-relinquishing routine that specifies that the OpenMP state is not required to persist. **564**, 564

hardware thread

An indivisible hardware execution unit on which only one OpenMP thread can execute at a time. **6**, 6, 37, 88, 128, 131, 534, 596, 726

has-device-addr attribute

For a given device construct, a data-sharing attribute of a data entity that refers to an object in a device data environment that is the same object to which the data entity of the same name in the enclosing data environment of the construct refers. **237**

host address

An address of an object that may be referenced on the host device. 56, 360, 907

host device

The device on which the OpenMP program begins execution. 3–5, 7, 9, 19, 26, 56, 59, 76, 100, 120, 127, 136, 138, 140, 141, 296, 307, 319, 359, 450, 454–457, 462, 463, 466, 564, 582, 585, 594, 598–602, 605, 606, 610, 630, 633, 634, 647, 650, 651, 690, 692, 697, 701, 703–706, 722, 791, 792, 803–805, 814, 828, 849–851, 857, 896, 910

host pointer

A pointer that refers to a host address. 359, 360, 605, 606, 610–612, 907

I

ICV

An internal control variable. **115**, 57, 61, 80, 84, 115, 118–122, 124–130, 132–137, 139–146, 216, 310, 321, 338, 358, 388–391, 394, 397, 404, 415, 419, 426, 429, 436, 437, 443, 451, 453, 454, 456, 461, 466, 501, 504, 520, 521, 537, 563, 568, 570–577, 580–588, 592–595, 599–602, 652–654, 678–680, 682–688, 692, 693, 699, 700, 792, 794, 817, 824, 829, 854, 863, 874–876, 890–892, 894, 896, 897, 902, 903, 906, 908, 914–916

ICV-defaulted clause

A clause that has the ICV-defaulted property. 437

ICV-defaulted property

The property of a clause that if it is not explicitly specified on a directive then the behavior is as if it were specified with an argument that is the value of an ICV. 57, 310, 451

ICV modifying property

The property of a routine or clause that its effect includes modifying the value of an ICV. 452, 568, 572, 573, 575, 582, 584, 592, 599, 601, 683

ICV retrieving property

The property of a routine that its effect includes returning the value of an ICV. 570, 572, 574, 576, 577, 579, 581–584, 586–588, 593, 594, 599, 601, 678–682, 684, 688

ICV scope

A context that contains one copy of a given ICV and defines the extent in which the ICV controls program behavior; the ICV scope may be the OpenMP program (i.e., global), the current device, the binding implicit task, or the data environment of the current task. **115**, 57, 115, 119, 121, 124, 127, 436, 454, 456, 461, 466

idle thread

An unassigned thread that is not currently executing any task. 447, 734

immediately nested construct

A construct is an immediately nested construct of another construct if it is immediately nested within the other construct with no intervening statements or directives. 57, 101, 395, 902

imperfectly nested loop

A nested loop that is not a perfectly nested loop. 910

implementation code

Implicit code that is introduced by the OpenMP implementation. 41, 50, 91, 719

implementation defined

Behavior that must be documented by the implementation and is allowed to vary among different compliant implementations. An implementation is allowed to define it as unspecified behavior. 6, 8, 15, 34, 42, 45, 50, 51, 75, 88, 91, 100, 110, 118, 119, 125, 128–131, 133–137, 139, 141, 142, 145, 146, 148, 149, 157, 204, 214, 217, 235, 237, 300, 304–308, 319, 322, 324, 325, 329, 330, 335, 341, 345, 352, 354, 355, 380–383, 385, 387, 389–392, 394, 397, 399, 405, 408, 415, 419, 420, 430, 437, 453, 463, 469, 471, 496, 533–535, 539, 541, 545, 558, 562, 574–576, 597, 610, 612, 613, 623, 627, 663, 680, 683, 685–687, 693, 695, 697, 701, 703, 704, 719, 726, 730, 733, 764, 779, 788, 793–795, 817, 844, 865, 870, 874, 885–895, 904, 908, 914

`implementation` selector set

A selector set that may match the implementation trait set. **321**, 321, 323

implementation trait set

The trait set that consists of traits that describe the functionality supported by the OpenMP implementation at a given point in the OpenMP program. **319**, 58, 318, 319

implicit array

For C/C++, the set of array elements of non-array type *T* that may be accessed by applying a sequence of [] operators to a given pointer that is either a pointer to type *T* or a pointer to a multidimensional array of elements of type *T*. For Fortran, the set of array elements for a given array pointer.

> COMMENT: For C/C++, the implicit array for pointer p with type *T* (*)[10] consists of all accessible elements p[*i*][*j*], for all *i* and *j*=0,1,...,9.

26, 27, 286

implicit barrier

A barrier that is specified as part of the semantics of a construct other than the **`barrier`** construct. 4–6, 385, 406, 407, 409, 412, 420, 447, 476, 477, 482, 521, 733

implicit flush

A flush that is specified as part of the semantics of a construct or routine other than the **`flush`** construct. 12, 101, 502, 911

implicitly determined data-mapping attribute

A data-mapping attribute that applies to a data entity for which no data-mapping attribute is otherwise determined. **276**, 274, 276, 285, 292, 739

implicitly determined data-sharing attribute

A data-sharing attribute that applies to a data entity for which no data-sharing attribute is otherwise determined. **213**, 96, 210, 213, 214, 222–224, 276, 277, 292, 912

implicit parallel region

An inactive parallel region that is not generated from a **`parallel`** construct. Implicit parallel regions surround the whole OpenMP program, all **`target`** regions, and all **`teams`** regions. 3–5, 37, 58, 59, 61, 95, 132, 301, 389, 395, 425, 446, 447, 582, 585, 600, 602, 689, 828, 917

implicit task

A task generated by an implicit parallel region or generated when a **`parallel`** construct is encountered during execution. 3, 4, 8, 19, 23, 28, 30, 37, 38, 51, 59, 61, 81, 83, 87, 100, 104, 105, 115–117, 124, 125, 214, 227, 252, 253, 270, 273, 384–386, 389–391, 404–415, 420, 421, 501, 503, 524, 682, 719, 744, 758, 794, 798, 828, 862–864

implicit task region

A region that corresponds to an implicit task. 3, 125, 758

importing task

A task that permits a preceding dependence-compatible task to be an antecedent task of one of its child tasks. **511**, 108, 427, 437, 507, 511, 559

inactive parallel region

A parallel region comprised of one implicit task and, thus, is being executed by a team comprised of only its primary thread. 58, 577, 579

inactive target region

A **`target`** region that is executed on the same device that encountered the **`target`** construct. 124

included task

A task for which execution is sequentially included in the generating task region. That is, an included task is an undeferred task and executed by the encountering thread. 7, 30, 52, 59, 91, 426, 439, 441, 454, 456, 459, 461, 466, 468, 479, 482, 603

inclusive scan computation

A scan computation for which the value read includes the updates performed in the same logical iteration. **269**, 269, 909

index-set splitting

The splitting of the logical iteration space into partitions that each are executed by a generated loop. **377**, 901

indirect device invocation

An indirect call to the device version of a procedure on a device other than the host device, through a function pointer (C/C++), a pointer to a member function (C++), a dummy procedure (Fortran), or a procedure pointer (Fortran) that refers to the host version of the procedure. 350, 351

induction

A use of an induction operation. 60, 239

induction attribute

For a given loop-nest-associated construct, a data-sharing attribute of a data entity that implies the private attribute and for which the value is updated according to an induction operation. **258**, 64

induction expression

A collector expression or an inductor expression. **240**, 240

induction identifier

An OpenMP identifier that specifies an inductor OpenMP operation to use in an induction. **239**, 239, 240, 246–249, 259, 263, 264

induction operation

A recurrence operation that expresses the value of a variable as a function, the inductor, applied to its previous value and a step expression. For an induction operation performed in a loop on the induction variable x and a loop-invariant step expression s, $x_i = x_{i-1} \oplus s, i > 0$, where x_i is the value of x at the start of collapsed iteration i, x_0 is the value of x before any tasks enter the loop, and the binary operator $\oplus$ is the inductor. For some inductors, the induction operation can be expressed in a non-recursive closed form as $x_i = x_0 \oplus s_i = x_0 \oplus (s \otimes i)$ where $s_i = s \otimes i$. The expression s_i is the collective step expression of iteration i and the binary operator $\otimes$ is the collector. 32, 59, 60, 64, 98, 111, 239, 243, 258, 266, 898

induction variable

A variable for which an induction operation determines its values. 60, 243, 264

inductor

A binary operator used by an induction operation. 60, 243

inductor expression

An OpenMP stylized expression that specifies how an induction operation determines a new value of an induction variable from its previous value and a step expression. **243**, 60, 243, 244, 246, 248, 258, 264, 265

informational directive

A directive that is neither declarative nor executable, but otherwise conveys user code properties to the compiler. **352**, 112, 152, 355, 363, 368, 369

initialization phase

The portion of an affected iteration that includes all statements that initialize private variables prior to the input phase and scan phase of a scan computation. **267**, 267, 268, 270, 899

initializer

An OpenMP operation that uses an initializer expression. **249**, 61, 90, 244, 245, 249, 252

initializer expression

An OpenMP stylized expression that determines the initializer for the private copies of list items in a reduction clause. **241**, 61, 90, 242–244, 248, 251, 261, 263, 267, 345

initial task

An implicit task associated with an implicit parallel region. 4, 5, 28, 61, 95, 124, 125, 253, 389, 394, 395, 413, 421, 446, 447, 453, 462, 503, 679, 705, 706, 719, 758, 785, 792, 798, 883

initial task region

A region that corresponds to an initial task. 3, 115, 116, 501, 503, 571, 577, 580

initial team

The team that comprises an initial thread executing an implicit parallel region. 4, 7, 105, 116, 394, 420, 422, 581, 829

initial thread

The thread that executes an implicit parallel region. 3, 4, 61, 84, 87, 106, 132, 133, 135, 216, 394, 395, 412, 420, 425, 446, 501, 503, 742, 886, 888

innermost-leaf property

The property that a clause applies to the innermost leaf construct that permits it when it appears on a compound construct. **159**, 180, 225, 232, 235, 269, 270, 272, 445, 488–492, 506, 517, 518, 528

input map type

The map type specified in a **`map`** clause specified on a construct to which map-type decay is applied to determine an output map type. **275**, 70, 82, 109, 275, 276

input phase

The portion of a logical iteration that contains all computations that update a list item for which a scan computation is performed. **267**, 60, 111, 267, 269, 270

input place partition

The place partition that is used to determine the *place-partition-var* and *place-assignment-var* ICVs and the place assignments of the implicit tasks of a **`parallel`** region. **389**, 389–391, 393

intent(in) property

The property that a routine argument is an **`intent(in)`** dummy argument in Fortran. In C/C++, the memory pointed to by the argument is not written by the runtime but must be readable. **535**, 596, 597, 604–606, 609, 611, 613, 614, 616, 617, 623–628, 631–636, 638–642, 644, 646, 648–652, 683, 685, 686, 692, 698, 726, 734, 748–752, 755, 759, 760, 762, 765, 766, 769, 770, 772, 774, 777, 780, 782, 786, 835, 837, 839, 840, 842, 844, 845, 854

intent(out) property

The property that a routine argument is an `intent(out)` dummy argument in Fortran. In C/C++, the memory pointed to by the argument is not read by the runtime but must be writeable. **535**, 623–625, 638, 640, 642, 684, 686, 787, 788, 847, 853, 870, 872, 873, 876

internal control variable

A conceptual variable that specifies runtime behavior of a set of threads or tasks in an OpenMP program. **115**, 56, 885

interoperability object

An OpenMP object of **`interop`** OpenMP type, which is an opaque type. These objects represent information that supports interaction with foreign runtime environments. **539**, 62, 181, 328, 334, 339, 468–471, 539, 543, 622, 629, 892, 902, 907

interoperability property

A property associated with an interoperability object. **468**, 62, 541, 622–625, 627, 628

interoperability-property-retrieving property

The property that a routine retrieves an interoperability property from an interoperability object. **622**, 62, 623–625

interoperability-property-retrieving routine

A routine that has the interoperability-property-retrieving property. **622**, 622, 624–626

interoperability routine

A routine that has the interoperability-routine property. **622**, 468, 541, 543, 622, 629

interoperability-routine property

The property that a routine provides a mechanism to inspect the properties associated with an interoperability object. **622**, 62, 623–628

intervening code

For two consecutive affected loops of a loop-nest-associated construct, user code that appears inside the loop body of the outer affected loop but outside the loop body of the inner affected loop. **198**, 84, 198, 204, 205, 434

is-device-ptr attribute

For a given device construct, a data-sharing attribute of a variable that implies the private attribute, and additionally the variable is initialized with a device address that corresponds to the device pointer variable of the same name in the enclosing data environment of the construct. **235**

ISO C binding property

The property of a routine that its Fortran version has the **`BIND(C)`** attribute. 63, 554, 556, 603–606, 608, 609, 611, 613–615, 617, 619, 620, 635, 640, 642, 643, 656–661

ISO C property

The property that a routine argument has the **`BIND(C)`** attribute in Fortran. If any argument of a routine has the ISO C property then the routine has the ISO C binding property. **535**, 63, 554, 604–609, 611, 613, 614, 616, 617, 619, 620, 640, 642, 656–661, 770, 774, 777

iteration count

The number of times that the loop body of a given loop is executed. **203**, 203–205, 264, 379, 383, 888

iterator

A programming mechanism to specify a set of values. 169, 170, 196, 204, 286, 400, 906, 916

iterator specifier

A tuple that specifies an *iterator-identifier* and its associated iterator value set. **169**, 63, 162, 169

iterator-specifier list

An argument list that consists of iterator-specifier list items. **162**

iterator-specifier list item

A list item that is an iterator specifier. **162**, 63

iterator value set

The set of values that correspond to a given instance of an *iterator* modifier. **170**, 63, 169–171

L

last-level cache

The last cache in a memory hierarchy that is used by a set of cores. 128

lastprivate attribute

For a given construct, a data-sharing attribute of a variable that implies the private attribute, and additionally, the final value of the variable may be assigned to the variable that has the same name in the enclosing data environment of the construct. **230**, 64, 211

lastprivate variable

A private variable that has the lastprivate attribute with respect to a given construct. 909

leaf construct

For a given construct, a construct that corresponds to one of the leaf directives of the executable directive. 20, 32, 46, 61, 82, 174, 318, 516, 528–531, 918

leaf directive

For a given directive, the directive itself if it is not a compound directive, or a directive from which the compound directive is composed that is not itself a compound directive. 35, 64, 527

leaf-directive name

The directive name of a leaf directive. **525**, 525, 527, 919

league

The set of teams formed by a `teams` construct, each of which is associated with a different contention group. **4**, 105, 116, 253, 394, 395, 421–423, 581, 725, 758

lexicographic order

The total order of two logical iteration vectors $\omega_a = (i_1, \ldots, i_n)$ and $\omega_b = (j_1, \ldots, j_n)$, denoted by $\omega_a \leq_{\text{lex}} \omega_b$, where either $\omega_a = \omega_b$ or $\exists m \in \{1, \ldots, n\}$ such that $i_m < j_m$ and $i_k = j_k$ for all $k \in \{1, \ldots, m-1\}$. 380, 381

linear attribute

For a given loop-nest-associated construct, a data-sharing attribute of a variable that is equivalent to an induction attribute for which the induction operation is a linear recurrence, where the binary operator $\oplus$ is $+$ and the step expression s is a loop-invariant integer expression. **232**, 64

linear variable

A private variable that has the linear attribute with respect to a given construct. **232**

list

A comma-separated set. 22, 39, 40, 64, 85, 158, 162, 163, 345, 349, 387, 444, 700, 886

list item

A member of a list. 21, 23, 33, 37, 39, 40, 46, 47, 49, 51, 61, 63, 65, 68–71, 73, 76, 80, 82, 83, 86, 87, 98, 109, 112, 141, 158–160, 162–165, 168–170, 210–212, 214, 217–222, 225–231, 233–239, 241, 243–245, 247–250, 252–254, 256–259, 267–270, 272–276, 279–291, 294–296, 300–303, 311–313, 315, 328, 332, 333, 338, 339, 345–349, 363, 364, 372–374, 378–380, 401, 421, 424, 430, 436, 437, 444, 445, 454, 456, 459, 461–464, 466, 499, 500, 507–509, 521, 522, 528–531, 534, 875, 888, 897, 899, 900, 904, 905, 910, 916

local static variable

A variable with static storage duration that for C/C++ has block scope and for Fortran is declared in the specification part of a procedure or **BLOCK** construct. 305, 309

locator list

An argument list that consists of locator list items. **162**, 160, 295, 437

locator list item

A list item that refers to storage locations in memory and is one of the items specifically identified in Section 5.2.1. **163**, 65, 162–164, 181, 435, 437, 505, 506, 508, 510

lock

An OpenMP variable that is used in lock routines to enforce mutual exclusion. 65, 66, 74, 75, 80, 97, 109, 110, 449, 496, 501, 504, 558, 561, 663–668, 670–676, 734, 742, 769, 788, 795, 893, 913

lock-acquiring property

The property that a routine may acquire a lock by putting it into the locked state. **670**, 65, 663, 670, 671

lock-acquiring routine

A routine that has the lock-acquiring property. **670**, 449, 663, 670, 675, 765–768

lock-destroying property

The property that a routine destroys a lock by putting it into the uninitialized state. **667**, 65, 668, 669

lock-destroying routine

A routine that has the lock-destroying property. **667**, 668, 669, 767

locked state

The lock state that indicates the lock has been set by some task. **663**, 65, 66, 673

lock-initializing property

The property that a routine initializes a lock by putting it into the unlocked state. **664**, 65, 664–667

lock-initializing routine

A routine that has the lock-initializing property. **664**, 664–667, 765, 766

lock property

The property that a routine operates on locks. **663**, 66

lock-releasing property

The property that a routine may unset a lock by returning it to the unlocked state. **672**, 66, 663, 673, 674

lock-releasing routine

A routine that has the lock-releasing property. **672**, 449, 663, 672, 673, 767, 768

lock routine

A routine that has the lock property. **663**, 65, 535, 663, 893

lock state

The state of a lock that determines if it can be set. **663**, 65, 109, 110, 663, 672–674

lock-testing property

The property that a routine that may set a lock by putting it into the locked state does not suspend execution of the task that executes the routine if it cannot set the lock. **675**, 66, 675, 676

lock-testing routine

A routine that has the lock-testing property. **675**, 675, 766–768

logical iteration

An instance of the executed loop body of a canonical loop nest, or a **DO CONCURRENT** loop in Fortran, denoted by a number in the logical iteration space of the loops that indicates an order in which the logical iteration would be executed relative to the other logical iterations in a sequential execution. 4, 20, 32, 33, 47, 50, 59, 61, 66, 67, 92, 94, 99, 107, 111, 204, 205, 253, 370, 371, 375, 377–382, 401, 429–433, 534, 719, 754, 889, 890, 905, 907, 910, 912, 916

logical iteration space

For a canonical loop nest, or a **DO CONCURRENT** loop in Fortran, the sequence $0,\ldots,N-1$ where N is the number of distinct logical iterations. **204**, 32, 47, 55, 59, 66, 107, 204, 374, 377–380, 534

logical iteration vector

An n-tuple $(i_1,\ldots,i_n)$ that identifies a logical iteration of a canonical loop nest, where n is the loop nest depth and i_k is the logical iteration number of the k^{th} loop, from outermost to innermost. 64, 66, 88, 205, 380, 381, 905

logical iteration vector space

The set of logical iteration vectors that each correspond to a logical iteration of a canonical loop nest. 205, 379, 381

loop body

A structured block that encompasses the executable statements that are iteratively executed by a loop statement. **197**, 62, 63, 66, 378, 434

loop-collapsing construct

A loop-nest-associated construct for which some number of outer loops of the associated loop nest may be collapsed loops. 31, 32, 205, 219, 220, 233, 398

loop-iteration variable

For a loop of a canonical loop nest, *var* as defined in Section 6.4.1. A C++ range-based `for`-statement has no loop-iteration variable. 67, 171, 196, 200–205, 211–213, 230, 233, 371, 424, 434, 512, 513, 529, 531, 916

loop-iteration vector

An n-tuple $(i_1, \ldots, i_n)$ that identifies a logical iteration of the affected loops of a loop-nest-associated directive, where n is the number of affected loops and i_k is the value of the loop-iteration variable of the k^{th} affected loop, from outermost to innermost. 67, 203, 204, 512, 513

loop-iteration vector space

The set of loop-iteration vectors that each corresponds to a logical iteration of the affected loops of a loop-nest-associated directive. **204**, 203, 204

loop-nest-associated construct

A loop-nest-associated directive and its associated loop nest. 60, 62, 64, 67, 92, 94, 97, 113, 154, 205, 234, 259, 372, 373, 380, 381, 404, 512, 531

loop-nest-associated directive

An executable directive for which the associated user code must be a canonical loop nest. **153**, 20, 23, 67, 152, 153, 198, 203, 211, 212, 233, 258, 371, 372, 375, 377, 379–381, 399, 416, 417, 420, 423, 429, 516

loop nest depth

For a canonical loop nest, the maximal number of loops, including the outermost loop, that can be affected by a loop-nest-associated directive. 66, 203, 206, 374

loop schedule

The manner in which the collapsed iterations of affected loops are to be distributed among a set of threads that cooperatively execute the affected loops. **205**, 35, 92, 94, 205, 398, 404, 414, 420, 423, 905

loop-sequence-associated construct

A loop-sequence-associated directive and its associated canonical loop sequence. 68, 207

loop-sequence-associated directive

An executable directive for which the associated user code must be a canonical loop sequence. **153**, 23, 67, 152, 371, 374

loop sequence length

For a canonical loop sequence, the number of consecutive canonical loop nests regardless of their nesting into blocks. 203, 208

loop-sequence-transforming construct

A loop-sequence-associated construct with the loop-transforming property. **371**

loop-transforming construct

A loop-transforming directive and its associated loop nest or associated canonical loop sequence. **371**, 54, 76, 108, 197, 203, 205, 370–374, 378, 431, 900, 901, 904, 907

loop-transforming directive

A directive with the loop-transforming property. 54, 68, 108, 371, 373, 374, 379

loop-transforming property

The property that a construct is replaced by the loops that result from applying the transformation as defined by its directive to its affected loops. 68, 369, 374, 375, 377, 379–381

loosely structured block

For Fortran, a block of zero or more executable constructs (including OpenMP constructs), where the first executable construct (if any) is not a Fortran **BLOCK** construct, with a single entry at the top and a single exit at the bottom. 99, 153

M

map-entering clause

A **map** clause that, if it appears on a map-entering construct, specifies that the reference counts of corresponding list items are increased and, as a result, those list items may enter the device data environment. **275**, 68, 283, 285, 361, 455

map-entering construct

A construct that has the map-entering property. 68, 274, 281, 283, 284, 287, 527, 564

map-entering map type

A *map-type* that specifies the clause on which it is specified is a map-entering clause. **275**, 275

map-entering property

A property of a construct that it may include mapping operations that allocate storage on the target device and that result in assignment to the corresponding list item from the original list item. 68, 275, 454, 458, 460

map-exiting clause

A **map** clause that, if it appears on a map-exiting construct, specifies that the reference counts of corresponding list items are decreased and, as a result, those list items may exit the device data environment. **275**, 69, 457

map-exiting construct

A construct that has the map-exiting property. 69, 274, 284, 527

map-exiting map type

A *map-type* that specifies the clause on which it is specified is a map-exiting clause. **275**, 275

map-exiting property

A property of a construct that it may include mapping operations that release storage on the target device and that result in assignment from the corresponding list item to the original list item. 69, 275, 456, 458, 460

mappable storage block

A storage block, derived from the list items of **map** clauses specified on a data-mapping construct, for which a corresponding storage block in a device data environment is created, removed, or otherwise referenced by the construct. 283, 284, 287, 296

mappable type

A type that is valid for a mapped variable. If a type is composed from other types (such as the type of an array element or a structure element) and any of the other types are not mappable types then the type is not a mappable type.

For C, the type must be a complete type.

For C++, the type must be a complete type; in addition, for class types:

- All member functions accessed in any **target** region must appear in a declare target directive.

For Fortran, no restrictions on the type except that for derived types:

- All type-bound procedures accessed in any **target** region must appear in a **declare_target** directive.

  COMMENT: Pointer types are mappable types but the memory block to which the pointer refers is not mapped.

69, 287, 290, 291, 296

mapped address range

For a given original list item, the address range that starts from its starting address and ends with its ending address. **280**, 71, 281

mapped variable

An original variable in a data environment with a corresponding variable in a device data environment. The original and corresponding variables may share storage. 38, 49, 69, 70, 82, 98, 464, 564

mapper

An operation that defines how variables of given type are to be mapped or updated with respect to a device data environment. 41, 48, 111, 183, 274–276, 278, 281–283, 287, 293–296, 298, 299

mapper identifier

An OpenMP identifier that specifies the name of a user-defined mapper. **278**, 278, 295

mapping operation

An operation that establishes or removes a correspondence between a variable in one data environment and another variable in a device data environment. 9, 23, 25, 69, 70, 95, 275, 283, 284, 286, 361, 564, 734, 739, 899, 900

map type

A categorization of a data-mapping clause that determines whether the mapping operations that result from that clause include assignments between the original storage and corresponding storage of its list items. 61, 82, 109, 283, 284

map-type decay

A process applied to input map type, according to an underlying map type, that results in an output map type. **275**, 61, 82, 275, 281, 459

map-type-modifying property

The property that a modifier that combines with a *map-type* to determine details of a mapping operation. 280, 282

matchable candidate

A mapped variable for which corresponding storage was created in a device data environment. **280**, 71, 281

matched candidate

A matchable candidate that, due to a matching mapped address range or extended address range, may determine the lower bound and length to use for a given assumed-size array that is a list item in a `map` clause. **281**, 236, 281, 287, 904

matching taskgraph record

A finalized taskgraph record that has a matching value for the scalar expression that identifies a `taskgraph` region. **436**, 92, 435–439

memory

A storage resource for storing and retrieving variables that are accessible by threads. **7**, 6–11, 13, 19, 20, 32, 44, 52, 63, 65, 71–73, 76, 89, 92, 99, 101, 105, 107, 114, 116, 143, 164, 165, 231, 303–308, 359, 360, 484–487, 494, 499, 509, 544, 555, 561, 603, 607, 608, 612, 618, 619, 630, 639, 643, 646, 647, 654, 655, 661, 662, 720, 774, 778, 779, 799, 821, 826, 833–837, 839, 840, 846, 853, 872, 874, 876, 885, 899, 900, 902, 903, 907–910, 913, 915

memory-allocating routine

A memory-management routine that has the memory-allocating-routine property. **654**, 20, 72, 89, 114, 654, 655, 662

memory-allocating-routine property

The property that a memory-management routine allocates memory. **654**, 71, 656–660

memory allocator

An OpenMP object that fulfills requests to allocate and to deallocate memory for program variables from the storage resources of its associated memory space. **9**, 9, 21, 23, 24, 71, 72, 116, 287, 305–313, 358, 463, 549, 646, 647, 652–655, 662, 888, 899, 903, 910

memory-allocator-retrieving property

The property that a memory-management routine retrieves a memory allocator handle. **647**, 71, 647–651

memory-allocator-retrieving routine

A memory-management routine that has the memory-allocator-retrieving property. **647**, 647–652

memory-copying property

The property that a routine copies memory from the device data environment of one device to the device data environment of another device. **612**, 71, 613–615, 617

memory-copying routine

A routine that has the memory-copying property. **612**, 52, 89, 448, 612, 613

memory-management routine

A routine that has the memory-management-routine property. **630**, 20, 71–73, 630, 635–637

memory-management-routine property

The property that a routine manages memory on the current device. **630**, 72, 631–636, 638–644, 646–653, 656–661

memory part

A storage block that resides on a single storage resource within a memory space. 72

memory partition

A definition of how a memory allocator divides the allocated memory into memory parts and the storage resources on which it allocates those memory parts. 72, 307, 553, 555, 556, 639, 641–644

memory partitioner

An OpenMP object that represents mechanisms to create and to destroy memory partitions. 72, 306, 307, 547, 553–555, 637–644

memory-partitioning property

The property that a memory-management routine creates or destroys or otherwise affects memory partitions or memory partitioners. **637**, 72, 638–643

memory-partitioning routine

A memory-management routine that has the memory-partitioning property. **637**

memory-reading callback

A callback that has the memory-reading property. **837**, 837, 838

memory-reading property

The property that a callback reads memory from an OpenMP program. **837**, 72, 838

memory-reallocating routine

A memory-management routine that has the memory-reallocating-routine property. **654**, 655, 660

memory-reallocating-routine property

The property that a memory-allocating routine deallocates memory in addition to allocating it. 72, 660

memory-setting property

The property that a routine fills memory in a device data environment with a specified value. **618**, 73, 619, 620

memory-setting routine

A routine that has the memory-setting property. **618**, 448, 618–621

memory space

A representation of storage resources from which memory can be allocated or deallocated. More than one memory space may exist. **630**, 9, 23, 24, 72, 73, 102, 144, 287, 304, 307, 317, 555, 630, 635–637, 643, 645, 647, 888, 903, 910

memory-space-retrieving property

The property that a memory-management routine retrieves a memory space handle. **630**, 73, 631–634

memory-space-retrieving routine

A memory-management routine that has the memory-space-retrieving property. **630**, 630–634

mergeable task

A task that may be a merged task if it is an undeferred task. **440**, 102, 427, 440, 468, 479

merged task

A task with a minimal data environment. 73, 428, 440, 449, 459, 719, 781, 882

metadirective

A directive that conditionally resolves to another directive. **324**, 47, 48, 92, 152, 324–327, 363, 889, 904, 905, 907, 910

minimal data environment

A data environment of a task that, inclusive of ICVs, is the same as that of its enclosing context, with the exception of list items in all-data-environments clauses that are specified on the task-generating construct that generated the task. 21, 73, 236, 238

modifier

A mechanism to customize clause behavior for its specified arguments. xxvii, 22, 33, 35, 41, 50, 53, 63, 70, 76, 80, 81, 86, 87, 91, 92, 97, 109, 110, 126, 158–163, 169, 171, 174, 181, 215, 224, 230, 231, 233, 249, 251, 268, 275, 276, 278, 280–282, 286, 287, 294–296, 300, 316, 317, 331–333, 342, 343, 348, 414, 419, 421, 435–437, 459, 468, 470, 471, 505, 513, 528, 529, 739, 888, 890, 891, 898–902, 904–907, 909, 911

mutex-acquiring callback

A callback that has the mutex-acquiring property. **765**

mutex-acquiring property

The property of a callback that it is dispatched when attempting to acquire mutually-exclusive access for a mutual-exclusion construct or when initializing or attempting to acquire a lock. **765**, 73, 766

mutex-execution callback

A callback that has the mutex-execution property. **767**

mutex-execution property

The property of a callback that it is dispatched when mutually-exclusive access is acquired or released for a mutual-exclusion construct or when a lock is acquired, released, or destroyed. **767**, 74, 767, 768

mutual-exclusion construct

A construct that has the mutual-exclusion property. 74, 765–768

mutual-exclusion property

The property that a construct provides mutual-exclusion semantics. 74, 473, 494, 514, 515

mutually exclusive tasks

Tasks that may be executed in any order, but not at the same time. 448, 508

N

named-handle property

The property that a handle is an integer kind in Fortran that is distinguished by the name of the handle. 538, 553, 558–560

named parameter list item

A parameter list item that is the name of a parameter of a procedure. **163**, 162, 163, 299, 300

named pointer

For C/C++, the base pointer of a given lvalue expression or array section, or the base pointer of one of its named pointers. For Fortran, the base pointer of a given variable or array section, or the base pointer of one of its named pointers.

> COMMENT: For the array section (*p0).x0[k1].p1->p2[k2].x1[k3].x2[4][0:n], where identifiers p*i* have a pointer type declaration and identifiers x*i* have an array type declaration, the named pointers are: p0, (*p0).x0[k1].p1, and (*p0).x0[k1].p1->p2.

74, 165

name-list trait

A trait that is defined with properties that match the names that identify particular instances of the trait that are effective at a given point in an OpenMP program. 318, 319, 321, 322

native thread

An execution entity upon which an OpenMP thread may be implemented. 3, 5, 6, 75, 80, 81, 88, 107, 117, 135, 136, 385, 395, 398, 719, 733, 734, 742, 745, 747, 777, 786, 817, 829, 836, 855–857, 867, 878

native thread context

A tool context that refers to a native thread. 822, 836, 837, 839, 841

native thread handle

A handle that refers to a native thread. 828, 854–857, 867, 869

native thread identifier

An identifier for a native thread defined by a native thread implementation. 138, 822, 829, 830, 841, 851, 855, 856

native trace format

A format for implementation defined trace records that may be device-specific. 75, 704–706, 812, 814

native trace record

A trace record in a native trace format. 706, 726, 727, 812–814

nestable lock

A lock that can be acquired (i.e., set) multiple times by the same task before being released (i.e., unset). **663**, 75, 504, 560, 663, 664, 672, 734, 769, 795

nestable lock property

The property that a routine operates on nestable locks. **663**, 75, 665, 667, 669, 671, 674, 676

nestable lock routine

A routine that has the nestable lock property. **663**, 560

nested construct

A construct (lexically) enclosed by another construct. 210

nested parallelism

A condition in which more than one level of parallelism is active at a point in the execution of an OpenMP program. 4, 908

nested region

A region (dynamically) enclosed by another region. That is, a region generated from the execution of another region or one of its nested regions. 3, 37, 76, 84, 369, 404

new list item

An instance of a list item created for the data environment of the construct on which a privatization clause or a data-mapping attribute clause specified. **219**, 37, 87, 111, 219–221, 226–228, 230, 233, 235, 236, 258, 267, 283–285, 916

NUMA domain

A device partition in which the closest memory to all cores is the same memory and is at a similar distance from the cores. 128

non-negative property

The property that an expression, including one that is used as the argument of a clause, a modifier or a routine, has a value that is greater than or equal to zero. **161**, 119, 130, 131, 133, 140, 142–144, 160, 163, 204, 305, 322, 378, 384, 394, 443, 541, 575, 582, 600, 636, 680, 695, 771, 793, 794, 892, 893

non-conforming program

An OpenMP program that is not a conforming program. 2, 34, 42, 110, 214, 217, 429, 448, 505, 663, 900

non-host declare target directive

A declare target directive that does not specify a **device_type** clause with **host**. 345

non-host device

A device that is not the host device. 7, 19, 26, 100, 117, 119, 120, 127, 139, 140, 329, 359, 362, 385, 425, 450, 464, 594, 690, 692, 850, 851, 857, 889, 896

non-null pointer

A pointer that is not NULL. 622, 698, 700, 704, 745, 746, 813

non-null value

A value that is not NULL. 655, 731, 797, 798, 818, 836, 837, 839, 871

non-property trait

A trait that is specified without additional properties. 318, 319, 323

nonrectangular-compatible property

The property that the transformation defined by a loop-transforming construct is compatible with non-rectangular loops and therefore will not yield a non-conforming canonical loop nest due to their presence. 371, 372, 375

non-rectangular loop

For a loop nest, a loop for which a loop bound references the iteration variable of a surrounding loop in the loop nest. 76, 200, 202, 205, 207, 234, 259, 372, 376, 380, 381, 420, 423, 433, 909

non-sequentially consistent atomic construct

An **atomic** construct for which the **seq_cst** clause is not specified 13

NULL

A null pointer. For C/C++, the value **NULL** or the value **nullptr**. For Fortran, the disassociated pointer for variables that have the **POINTER** attribute or the value **C_NULL_PTR** for variables of type **C_PTR**. 76, 145, 332, 590, 597, 605–609, 611, 612, 618, 627, 628, 654, 655, 661, 684, 686, 687, 695, 698, 700, 704, 744, 757, 758, 763, 764, 771, 773, 774, 779, 781, 787, 789, 790, 795–799, 818, 836, 837, 839, 844, 872, 894

numeric abstract name

An abstract name that refers to a quantity associated with a conceptual abstract name. **128**, 19, 85, 128–130, 134, 897

O

offsetting loop

The outer generated loops of a **stripe** construct that determine the offsets within the grid cells used for each execution of the grid loops. **379**, 379, 380, 889

OMPD

An interface that helps a third-party tool inspect the OpenMP state of a program that has begun execution. **816**, 2, 14, 15, 77, 108, 116, 146, 184, 185, 816–818, 820, 822, 824, 827–829, 833, 836, 841, 845–849, 855, 878

OMPD callback

A callback that has the OMPD property. 184, 185, 823, 826, 827, 831, 833, 836, 837, 839, 841

OMPD library

A dynamically loadable library that implements the OMPD interface. **816**, 15, 46, 816–823, 826, 829–831, 833–839, 841–851, 853, 867, 870, 872, 874, 876

OMPD property

The property that a callback, routine or type is included in OMPD and its namespace, which implies it has the **ompd_** prefix. 77, 78, 819, 820, 822–832, 834, 835, 837–849, 851–869, 871–873, 875–877

OMPD routine

A routine that has the OMPD property. 826, 827, 831, 845–850, 855, 856, 858–862, 875–877

OMPD type

A type that has the OMPD property. **184**, 33, 56, 81, 83, 184, 185, 819–824, 826–837, 839, 841–844

OMPT

An interface that helps a first-party tool monitor the execution of an OpenMP program. **697**, 2, 14, 45, 78, 98, 144, 146, 185, 476, 565, 690, 697–701, 703–706, 722, 725, 727, 733, 744–746, 772, 786, 787, 802, 803, 812, 813, 877, 894, 903

OMPT active

An OMPT interface state in which the OpenMP implementation is prepared to accept runtime calls from a first-party tool and will dispatch any registered callbacks and in which a first-party tool can invoke runtime entry points if not otherwise restricted. 695, 700, 707

OMPT callback

A callback that has the OMPT property. 185, 703, 711, 713, 744, 787, 802

OMPT inactive

An OMPT interface state in which the OpenMP implementation will not make any callbacks and in which a first-party tool cannot invoke runtime entry points. 695, 699, 700, 745

OMPT interface state

A state that indicates the permitted interactions between a first-party tool and the OpenMP implementation. 78, 695, 699, 700, 707, 745

OMPT pending

An OMPT interface state in which the OpenMP implementation can only call functions to initialize a first-party tool and in which a first-party tool cannot invoke runtime entry points. 699, 700

OMPT property

The property that a callback, runtime entry point or type is included in OMPT and its namespace, which implies it has the `ompt_` prefix. 78, 79, 697, 698, 708, 710–712, 714–732, 734–743, 745–753, 755–757, 759–770, 772–777, 780, 782, 784, 786–797, 799–801, 803–814

OMPT-tool finalizer

An implementation of the **`finalize`** callback. **707**, 446, 698, 746

OMPT-tool initializer

An implementation of the **initialize** callback. **697**, 446, 698, 700, 703, 745

OMPT type

A type that has the OMPT property. xxvii, **184**, 33, 56, 81, 83, 185, 415, 697, 698, 700, 703, 705–708, 710, 711, 713–731, 733, 735–738, 740–743, 745–751, 753, 754, 756–776, 778, 779, 781, 783–785, 787–796, 798–814, 824, 830, 864, 870, 877, 894, 896, 903, 905, 908

once-for-all-constituents property

The property that a clause applies once for all constituent constructs to which it applies when it appears on a compound construct. **159**, 205, 206, 528

opaque property

The property that an OpenMP type is opaque, which implies that objects of that type may only be accessed, modified and destroyed through OpenMP directives, routines, callbacks and entry points. Further, an object of an opaque type can be copied without affecting, or copying, its underlying state. Destruction of an OpenMP object, which by definition has an opaque type, destroys the state to which all copies of the object refer. All handles have opaque types. 79, 538, 539, 553, 558–560, 623–628, 710, 717, 772, 776, 811–813, 840, 849–853, 857, 858, 860, 863, 865–873, 875–877

opaque type

A type that has the opaque property. 62, 79, 80, 538, 539, 553, 558–560

OpenMP Additional Definitions document

A document that exists outside of the OpenMP specification and defines additional values that may be used in a conforming program. The OpenMP Additional Definitions document is available via `https://www.openmp.org/specifications/`. 79, 140, 319, 469, 539, 541

OpenMP API routine

A runtime library routine that is defined by the OpenMP implementation and that can be called from user code via the OpenMP API. 45, 80, 93, 115, 127, 240, 359, 360, 367, 533, 586, 630, 688, 694, 892

OpenMP architecture

The architecture on which a region executes. 80, 699

OpenMP context

The execution context of an OpenMP program as represented by a set of traits, including active constructs, execution devices, OpenMP functionality supported by the implementation and any available dynamic values. **318**, 33, 37, 98, 183, 318, 320, 321, 323–325, 328–331, 335, 337, 341, 355, 541, 889, 906

OpenMP environment variable

A variable that is part of the runtime environment in which an OpenMP program executes and that a user may set to control the behavior of the program, typically through the initialization of an ICV. **127**, 45, 50, 115, 120, 127, 872, 914

OpenMP identifier

An identifier that has a specialized purpose for use in OpenMP programs, as defined by this specification. **183**, 60, 70, 86, 90, 93, 159, 164, 183, 185, 241–244

OpenMP lock variable

A lock. 663

OpenMP object

Any object of an opaque type that allows programmers to save, to manipulate and to use state related to the OpenMP API. 42, 62, 71, 72, 79, 505, 773, 776, 803, 811, 813

OpenMP operation

When used as a list item, a special expression that returns an object of a specified OpenMP types. Otherwise, an operation that is applied to a list item according to the semantics of a directive, clause, or modifier. **165**, 60, 61, 80, 90, 162, 165, 183, 333, 406, 499

OpenMP operation list

An argument list that consists of OpenMP operation list items. **162**, 165

OpenMP operation list item

A list item that is an OpenMP operation. **162**, 80

OpenMP process

A collection of one or more native threads and address spaces. An OpenMP process may contain native threads and address spaces for multiple OpenMP architectures. At least one native thread in an OpenMP process is mapped to an OpenMP thread. An OpenMP process may be live or a core file. 20, 80, 819, 820, 829, 836, 845, 846, 849, 850

OpenMP program

A program that consists of a base program that is annotated with OpenMP directives or that calls OpenMP API routines. 3, 5–9, 13, 14, 19, 21, 22, 26, 32, 35, 36, 44–46, 48, 55–58, 62, 72, 75, 76, 78–80, 91, 93, 108, 110, 115, 117, 127, 138, 148, 149, 164, 183, 214, 217, 222, 233, 251, 289, 293, 294, 304, 305, 318–320, 325, 360, 370, 395, 404, 443, 463, 464, 472, 473, 497, 499, 505, 582, 585, 592, 600, 602, 612, 663, 678, 688, 690, 691, 694, 695, 697, 699, 700, 703, 720, 721, 744, 771, 789, 796, 801, 802, 808, 816–818, 821, 826, 829, 835, 837, 839, 842–844, 878, 885, 915, 917

OpenMP property

The property that a routine, callback or type is in the OpenMP namespace, which implies it has the **omp_** prefix. 81, 536–542, 544, 545, 547, 548, 550, 552–554, 556–558, 560, 562, 563, 565, 566, 573, 574, 623–628, 631–636, 638–642, 644, 646, 648–652, 656–661, 664–671, 673–676, 694

OpenMP stylized expression

A base language expression that is subject to restrictions that enable its use within an OpenMP implementation. 32, 33, 60, 61, 159, 185, 240

OpenMP thread

A logical execution entity with a stack and associated thread-specific memory subject to the semantics and constraints of this specification and may be implemented upon a native thread. 5–7, 22, 56, 75, 80, 84, 105–107, 132, 134, 136, 568, 777, 851, 854–858, 860, 863, 871, 878, 890

OpenMP thread pool

The set of all threads that may execute a task of a contention group and, thus, are ever available to be assigned to a team that executes implicit tasks of the contention group, 3, 5, 22, 93, 94, 106, 442, 448

OpenMP type

A type that has the OpenMP property or a type that is an OMPD type or an OMPT type. **183**, 23, 33, 53, 56, 62, 79, 80, 82, 83, 159, 162, 163, 165, 181–185, 204, 334, 376, 469, 509, 519, 533, 534, 536, 538, 539, 541, 543–545, 547, 549, 552–556, 558–567, 622, 771, 892, 905, 907

optional property

The property that a clause, a modifier or an argument is optional and thus may be omitted. If any argument of a routine has the optional property then the routine has the overloaded property. 81, 157–159, 163, 206, 270, 325, 326, 334, 341, 343, 344, 346, 350, 357–362, 365–367, 372, 382, 383, 393, 418, 422, 439, 440, 473, 481, 483–492, 498, 511, 517, 518, 535, 616, 617, 620, 623–625

order-concurrent-nestable construct

A construct that has the **order-concurrent**-nestable property. 398, 917

order-concurrent-nestable property

The property that a construct or routine generates a region that may be a strictly nested region of a region that was generated by a construct on which an **order** clause with an *ordering* argument of **concurrent** is specified. 81, 374, 375, 377, 379–381, 384, 399, 423, 494

order-concurrent-nestable routine

A routine that has the **order-concurrent**-nestable property. 398, 917

original list item

The instance of a list item in the data environment of the enclosing context. 37, 49, 51, 69, 70, 82, 98, 215, 219–221, 225, 227–231, 233, 235–237, 242, 247, 248, 250–254, 256–259, 267, 268, 271, 272, 280, 283–285, 288, 289, 295, 296, 298, 346, 361, 418, 420, 422, 444, 466, 899, 916

original list-item updating clause

A clause that has the original list-item updating property 522

original list-item updating property

The property that a clause includes an effect of updating the value of the original list item when the region for which it is specified is completed. 82, 229, 252, 255, 257

original pointer

An original list item that corresponds to a corresponding pointer. 284

original storage

The storage of a given mapped variable. 8, 70, 95, 285, 286, 739

original storage block

A storage block that contains the storage of one or more mapped variables in a data environment. 8, 9, 38, 283

original variable

For a variable that is referenced in the structured block that is associated with a block-associated directive that accepts data-sharing attribute clauses, the variable by the same name that exists immediately outside the construct. **7**, 7

orphaned construct

A construct that gives rise to a region for which the binding thread set is the current team, but is not nested within another construct that gives rise to the binding region. 515

outermost-leaf property

The property that a clause applies to the outermost leaf construct that permits it when it appears on a compound construct. **159**, 237, 271, 481, 483, 528

output map type

The map type that results when map-type decay is applied to an input map type. **275**, 61, 70, 109, 275, 281

overlapping type name

An OpenMP type for which its name has the overlapping type-name property. 754

overlapping type-name property

The property that an OpenMP type name is used for both an ordinary OpenMP type (possibly an OMPD type or an OMPT type) and for a callback in the same name space; which type is intended should be apparent from the context in this document. 82, 717, 722, 735, 743, 752, 753, 762, 765, 766

overloaded property

The property that a routine has an overloaded C++ interface. 81, 83, 655–661

overloaded routine

A routine that has the overloaded property. 655, 661

P

parallel handle

A handle that refers to a parallel region. **831**, 828, 858–860, 866, 868

parallelism-generating construct

A construct that has the parallelism-generating property. 231, 367, 371, 526

parallelism-generating property

The property that a construct enables parallel execution by generating one or more teams, explicit tasks, or SIMD instructions. 83, 384, 394, 399, 426, 429, 454, 456, 458, 460, 465

parallel region

A region that has a set of associated implicit tasks and an associated team of threads that execute those tasks. 4, 5, 19, 23, 31, 38, 53, 59, 83, 85, 100, 103–105, 114, 116, 125, 132, 136, 273, 389, 402, 404–407, 409, 414, 423–426, 429, 475–478, 502, 527, 536, 568–570, 715, 722, 744, 758, 763, 764, 796–798, 827, 831, 854, 858–860, 863, 866, 894, 914, 916

parameter list

An argument list that consists of parameter list items. **162**

parameter list item

A list item that identifies one or more parameters of a procedure. **162**, 74, 83, 162, 163, 534

parent device

For a given `target` region, the device on which the corresponding `target` construct was encountered. 257, 359, 451, 461

parent thread

The thread that encountered the `parallel` construct and generated a `parallel` region is the parent thread of each thread that executes a task region that binds to that `parallel` region. The primary thread of a **parallel** region is the same thread as its parent thread with respect to any resources associated with an OpenMP thread. The thread that encounters a **target** or **teams** construct is not the parent thread of the initial thread of the corresponding **target** or **teams** region. **4**, 22, 83, 84

partial tile

A tile that is not a complete tile. **381**, 381

partitioned construct

A construct that has the partitioned property. **404**, 84, 526

partitioned property

The property of a construct that it is a work-distribution construct for which any encountered user code in the corresponding region, excluding code from nested regions that are not closely nested regions, is executed by only one thread from its binding thread set. 84, 405, 407, 409, 412, 416, 417, 420, 423

partitioned worksharing construct

A construct that is both a partitioned construct and a worksharing construct. **4**, 84

partitioned worksharing region

A region that corresponds to a partitioned worksharing construct. 917

perfectly nested loop

A loop that has no intervening code between it and the body of its surrounding loop. The outermost loop of a loop nest is always perfectly nested. **198**, 57, 268, 376, 379–381, 514, 916

persistent self map

A self map for which the corresponding storage remains present in the device data environment, as if it has an infinite reference count. **360**, 8, 885

place

An unordered set of processors on a device. **130**, 4, 61, 84, 85, 106, 116, 117, 128, 131–133, 389–393, 679–682, 792–794, 886, 890, 897

place-assignment group

A logical group of places and positions from the *place-assignment-var* ICV that is used to define a set of assignments of threads to places according to a given thread affinity policy. **390**, 390, 391

place-count abstract name

A numeric abstract name that refers to a quantity associated with a place-list abstract name. **128**

place list

The ordered list that describes all OpenMP places available to the execution environment. 85, 131, 394, 679, 792, 886, 897

place-list abstract name

A conceptual abstract name that refers to a set of hardware abstractions of a given category that may be used to specify each place in a place list. **128**, 85, 128, 131

place number

A number that uniquely identifies a place in the place list, with zero identifying the first place in the place list, and each consecutive whole number identifying the next place in the place list. **390**, 390, 681, 682, 793, 794

place partition

An ordered list that corresponds to a contiguous interval in the place list. It describes the places currently available to the execution environment for a given parallel region. 61, 106, 117, 391, 392

pointer association query

A query to the association status of a pointer via comparison to zero in C/C++ or by calling the **ASSOCIATED** intrinsic with one argument in Fortran. 463

pointer attachment

The process of making a pointer variable an attached pointer. **284**, 25, 285

pointer property

The property that a routine or callback either returns a pointer type in C/C++ and is an assumed-size array in Fortran or has an argument that has such a type. **535**, 596, 597, 614, 616, 617, 620, 625–628, 631, 632, 636, 644, 648, 649, 680, 682–686, 698, 714, 726, 734, 745–753, 755–757, 759–762, 765, 766, 769, 770, 772, 774–777, 780, 782, 784, 786–788, 790–793, 795–797, 799, 800, 803–814, 834, 835, 837, 839–842, 844–846, 849–854, 856–873, 875–877

pointer-to-pointer property

The property that a routine or callback either returns a pointer-to-pointer type in C/C++ or has an argument that has such a type. **535**, 775, 782, 787, 788, 796, 797, 799, 800, 834, 840, 847, 849–851, 853, 854, 856–863, 870, 873, 876

positive property

The property that an expression, including one that is used as the argument of a clause, a modifier or a routine, has a value that is greater than zero. **161**, 129–131, 133–135, 160, 162, 206, 207, 300, 305, 306, 309, 313, 373, 374, 376, 383, 388, 393, 397, 401, 418, 422, 432, 433, 452, 546, 547, 568, 583, 584, 602, 605, 614, 617, 631, 632, 645, 648, 649, 734, 805, 886, 887, 889–893

post-modified property

The property of a clause that its modifiers must appear after its arguments. **158**, 159, 161, 223, 232, 291, 300

preceding dependence-compatible task

For a given task, a dependence-compatible task that may be its antecedent task. **507**, 51, 59, 507, 508

predecessor task

For a given task, an antecedent task of that task, or any predecessor task of any of its antecedent tasks. **507**, 86, 455, 457, 462, 466, 479, 508

predefined default mapper

The default mapper that is used if no default mapper that is a user-defined mapper is visible for the type of a given list item. **278**, 238, 278, 281, 282, 288, 295, 296

predefined identifier

Unless otherwise specified, an OpenMP identifier that is defined for use in arbitrary base language expressions. **183**, 7, 183, 378, 533, 534, 692, 693, 708, 847, 892

predetermined data-sharing attribute

A data-sharing attribute that applies regardless of the clauses that are specified on a given construct, unless explicitly specified otherwise. **211**, 210–213, 222, 224, 276, 292, 461, 528, 915

preference specification

The specification of a set of preferences for interoperating with a foreign runtime environment. **470**, 86, 162, 471, 891

preference specification list

An argument list that consists of preference specification list items. **162**

preference specification list item

A list item that is a preference specification. **162**, 86, 470

pre-modified property

The property of a clause that its modifiers must appear before its arguments. **158**, 161

preprocessed code

For C/C++, a sequence of preprocessing tokens that result from the first six phases of translation, as defined by the base language. 337, 906

present storage

A storage block that exists in a given device data environment. 282–287

primary thread

An assigned thread that has thread number 0. A primary thread may be an initial thread or the thread that encounters a `parallel` construct, forms a team, generates a set of implicit tasks, and then executes one of those tasks as thread number 0. **4**, 4, 5, 28, 53, 59, 84, 87, 105, 106, 216, 271, 272, 384, 385, 390, 392, 403, 405, 503, 569, 796, 916

private attribute

For a given construct, a data-sharing attribute of a data entity that its lifetime is limited to that of the corresponding region and it is visible only to a single task generated by the construct or to a single SIMD lane used by the construct. **219**, 7, 8, 21, 52, 60–63, 87, 90, 111, 160, 210–212, 214, 221, 228, 231, 236, 238, 241, 242, 247, 252–254, 256, 257, 267, 268, 273, 313, 371, 404, 521, 528, 915

private-only variable

A variable that has a private attribute and no other data-sharing attribute with respect to a given construct. **226**, 437

private variable

A variable that has a private attribute with respect to a given construct. **7**, 7, 8, 52, 60, 64, 87, 90, 220, 222, 267, 268, 270, 273, 410, 413, 418, 421, 422, 898

privatization clause

The clause that may result in private variables that are new list items. **210**, 37, 76, 87, 222, 236

privatization property

The property that a clause privatizes list items. 225, 227, 229, 232, 235, 236, 252, 255–257, 445

privatized list item

A list item that appears in the argument list of a privatization clause, resulting in one or more private new list items. **219**, 219–222, 225, 226, 235, 253

procedure

A function (for C/C++ and Fortran) or subroutine (for Fortran). 15, 26, 30, 35, 41, 45, 51, 53, 54, 59, 65, 74, 83, 91, 95, 100, 107, 108, 112, 146, 149, 154, 161–164, 184, 188, 214, 225, 226, 228, 233, 234, 240, 261, 264, 277, 281, 282, 294, 296, 300, 318, 319, 322, 329, 330, 335, 336, 341–345, 347–351, 402, 412, 413, 448, 450, 461, 464, 533, 534, 555, 556, 638, 639, 641–644, 697, 698, 700, 707, 719, 721, 731, 798, 806, 821, 826, 827, 831, 836, 841, 889, 906, 910

procedure property

The property that a routine argument has a function pointer type in C/C++ and a procedure type in Fortran. **535**, 638, 808

processor

An implementation defined hardware unit on which one or more threads can execute. 43, 84, 117, 131, 136, 595, 680, 791–794, 803, 885, 886, 914

product order

The partial order of two logical iteration vectors $\omega_a = (i_1, \ldots, i_n)$ and $\omega_b = (j_1, \ldots, j_n)$, denoted by $\omega_a \leq_{\text{product}} \omega_b$, where $i_k \leq j_k$ for all $k \in \{1, \ldots, n\}$. 381

program order

An ordering of operations performed by the same thread as determined by the execution sequence of operations specified by the base language.

> COMMENT: For versions of C and C++ that include base language support for threading, program order corresponds to the *sequenced-before* relation between operations performed by the same thread.

12, 13, 88, 98

progress group

A group of consecutive threads in a team that may execute on the same progress unit. **393**, 393

progress unit

An implementation defined set of consecutive hardware threads on which native threads may execute a common stream of instructions. **6**, 6, 7, 88, 393, 534, 596

property

A characteristic of an OpenMP feature. xxvii, 20–22, 24, 28–31, 33, 35, 37, 39–41, 43–46, 49, 50, 52–55, 57, 61–63, 65, 66, 68–79, 81–97, 101, 103–107, 109, 110, 112–114, 159, 160, 169, 173, 179–182, 205–207, 215, 223–227, 229, 230, 232, 235–238, 251, 252, 255–258, 260, 262, 263, 265, 266, 269–272, 274, 278–280, 289–291, 293, 297–301, 303, 309, 310, 312, 313, 315, 316, 318–321, 323, 325–327, 330, 331, 333, 334, 336–341, 343, 344, 346, 349, 350, 352–355, 357–369, 372, 374–384, 388, 392–394, 397–403, 405–409, 412, 416–418, 420, 422, 423, 425, 426, 429, 432–435, 438–446, 450–452, 454, 456, 458, 460, 465, 468–470, 472, 473, 475, 478, 479, 481–494, 498, 504–507, 511, 512, 514, 515, 517–520, 524, 528, 535–545, 547, 548, 550, 552–554, 556–560, 562, 563, 565, 566, 568–579, 581–584, 586–590, 592–602, 604–609, 611, 613–617, 619, 620, 622–628, 631–636, 638–644, 646–653, 656–661, 664–671, 673–676, 678–686, 688–692, 694, 697, 698, 708–712, 714–732, 734–743, 745–753, 755–757, 759–770, 772–777, 780, 782, 784, 786–797, 799–801, 803–814, 819, 820, 822–832, 834, 835, 837–873, 875–877, 892, 899

pure property

The property that a directive has no observable side effects or state, yielding the same result every time it is encountered. 149, 215, 260, 263, 266, 293, 301, 310, 325, 327, 334, 341, 346, 352, 368, 369, 374, 375, 377, 379–381, 399, 897, 904

R

raw-memory-allocating routine

A memory-allocating routine that has the raw-memory-allocating-routine property. **654**, 654–657

raw-memory-allocating-routine property

The property that a memory-allocating routine returns a pointer to uninitialized memory. **654**, 89, 656, 657

read-modify-write

An atomic operation that reads and writes to a given storage location.

COMMENT: Any atomic update is a read-modify-write operation.

11, 89

read structured block

An atomic structured block that may be associated with an **`atomic`** directive that expresses an atomic read operation. **189**, 190, 192, 497

rectangular-memory-copying property

The property of a memory-copying routine that the memory that it copies forms a rectangular subvolume. **612**, 89, 614, 617

rectangular-memory-copying routine

A routine with the rectangular-memory-copying property. **612**, 612, 615, 618, 735, 779, 893

reduction

A use of a reduction operation. 33, 90, 104, 183, 239–242, 244, 245, 249–251, 253, 256, 430, 898, 904, 907, 909, 912, 914

reduction attribute

For a given construct, a data-sharing attribute of a data entity that implies the private attribute and for which a partial result is computed in the context of a reduction computation. **249**, 90

reduction clause

A reduction-scoping clause or a reduction-participating clause. **239**, 61, 219, 222, 239–241, 247–251, 253, 256, 257, 260, 261

reduction expression

A combiner expression or an initializer expression. **240**, 240

reduction identifier

An OpenMP identifier that specifies a combiner OpenMP operation to use in a reduction. **239**, 183, 239, 240, 244, 245, 247–249, 251, 260, 261, 430, 899

reduction operation

An operation that applies a combiner and an associated initializer to a set of values. 32, 89, 94, 111, 239

reduction-participating clause

A clause that defines the participants in a reduction. **239**, 90, 239, 251, 252, 256

reduction-participating property

The property that a clause is a reduction-participating clause. 252, 256

reduction-scoping clause

A clause that defines the region in which a reduction is computed. **239**, 90, 239, 250–253, 256, 257, 430

reduction-scoping property

The property that a clause is a reduction-scoping clause. 252, 255

reduction variable

A private variable that has the reduction attribute with respect to a given construct. **249**, 249

referenced pointee

For a given referencing variable, the referenced data object to which the referring pointer points. 26, 27, 91, 237, 238, 279, 282, 283, 296

referencing variable

For C++, a data entity that is a reference. For Fortran, a data entity that is an allocatable variable or a data pointer. 25, 27, 90, 91, 112, 210, 212, 237, 238, 279, 282, 283, 289, 296

referring pointer

If a given referencing variable is a Fortran data pointer, the pointer object that is pointer associated with the referenced pointee; otherwise, an associated implementation defined handle through which the referenced pointee is made accessible. 25, 37, 38, 90, 210, 212, 238, 279, 282–284, 289, 461

region

All code encountered during a specific instance of the execution of a given construct, structured block sequence or routine. A region includes any code in called procedures as well as any implementation code. The generation of a task at the point where a task-generating construct is encountered is a part of the region of the encountering thread. However, an explicit task region that corresponds to a task-generating construct is not part of the region of the encountering thread unless it is an included task region. The point where a **`target`** or **`teams`** directive is encountered is a part of the region of the encountering thread, but the region that corresponds to the **`target`** or **`teams`** directive is not.

A region may also be thought of as the dynamic or runtime extent of a construct or of a routine. During the execution of an OpenMP program, a construct may give rise to many regions. 3–8, 12, 13, 19, 21, 22, 26, 28, 30, 31, 38, 39, 42, 45, 47–51, 54, 55, 58, 59, 61, 69, 71, 76, 79, 81–84, 87, 90–93, 95–97, 99, 101–107, 109, 113–117, 122, 124, 128–130, 133, 136, 149, 155, 193, 194, 198, 205, 210, 214, 216, 217, 220, 221, 228, 231, 237, 239, 240, 248, 250–254, 256, 257, 271, 281, 283, 284, 286, 288, 296, 306–308, 311, 313, 316, 328, 338, 340, 345, 358, 359, 366, 369, 384, 385, 388, 389, 391, 394–396, 398–400, 402–410, 412, 413, 420, 421, 423–427, 429, 430, 433, 435–437, 439, 445–451, 454, 456, 458, 459, 461–466, 468, 472–480, 494–505, 513–516, 519–525, 535, 564, 568, 569, 571, 576, 577, 580–583, 585, 588, 590, 592–594, 596–603, 618, 630, 645, 646, 652, 653, 655, 664–676, 678, 683, 685–687, 689, 690, 693–695, 703, 704, 706, 715, 719, 725, 733, 734, 736, 742, 744, 749–751, 753, 758, 763–768, 778, 781, 785, 788, 795, 796, 800, 850, 859, 878–881, 883, 885, 887–889, 891, 893, 894, 898, 900–903, 906, 910, 912, 913, 915–918

region endpoint

An event that indicates the beginning or end of a region that may be of interest to a tool. 703, 704, 729

region-invariant property

The property that an expression, including one that is used as the argument of a clause, a modifier or a routine, has a value that is invariant for the associated region. **161**, 160, 232, 258, 300, 384, 394, 418, 422

registered callback

A callback for which callback registration has been performed. 14, 29, 78, 701, 703, 894

release flush

A flush that has the release flush property. **10**, 11, 12, 92, 101, 496, 499, 501–504

release flush property

A flush with the release flush property orders memory operations that precede the flush before memory operations performed by a different thread with which it synchronizes. 52, 92, 499

release sequence

A set of modifying atomic operations that are associated with a release flush that may establish a synchronizes-with relation between the release flush and an acquire flush. 11, 12, 502

repeatable property

The property that a clause or modifier may appear more than once in a given context with which it is associated. 159, 180

replacement candidate

A directive variant or function variant that may be selected to replace a metadirective or base function. **324**, 30, 48, 324, 325, 328, 329, 331, 335, 889

replayable construct

A task-generating construct that an implementation must record into a taskgraph record, if one is recorded. **435**, 92, 94, 103, 215, 435–437, 441

replay execution

An execution of a given **`taskgraph`** region that entails executing replayable constructs that are saved in a matching taskgraph record. **436**, 52, 94, 103, 215, 435–437, 891, 898

reproducible schedule

A loop schedule for the affected loop nest of a given loop-nest-associated construct that does not change between different executions of the construct that have the same binding thread set and have the same number of logical iterations. **404**, 205, 398, 414, 420, 423, 905

required property

The property that a clause, a modifier, an argument, or at least one member of a clause set is required and, thus, may not be omitted. **160**, 157, 159–161, 181, 251, 252, 255, 256, 258, 262, 265, 266, 325, 330, 331, 356, 363, 374, 378, 458, 465, 468, 505, 511, 512, 519, 535

reservation type

A thread-reservation type. 142

reserved locator

An OpenMP identifier that represents system storage that is not necessarily bound to any base language storage item. **164**, 163, 164, 506, 508, 509, 906

reserved thread

A thread in an OpenMP thread pool that must have a particular thread-reservation type when executing a task. 141

resource-relinquishing property

The property that a routine relinquishes some (or all) resources that the OpenMP program is currently using. **688**, 93, 689, 690

resource-relinquishing routine

A routine that has the resource-relinquishing property. **688**, 56, 98, 563, 564, 688, 689

reverse-offload region

A region that is associated with a **`target`** construct that specifies a **`device`** clause with the **`ancestor`** *device-modifier*. 345, 911

routine

Unless specifically stated otherwise, an OpenMP API routine. xxvii, 2, 3, 6, 7, 14, 15, 17, 21, 22, 24, 28, 30, 35, 44, 49, 52, 53, 55, 57, 58, 61–63, 65, 66, 71–73, 75–79, 81, 83, 85, 86, 88, 89, 91, 93, 97, 105, 110, 112, 115, 120–122, 129, 139, 147, 216, 306, 398, 462, 463, 533–535, 537, 555, 556, 561, 563–565, 567–590, 592–612, 614–616, 618, 620–626, 628–676, 678–694, 698, 701, 744, 745, 754, 760, 769, 787, 798, 817, 826, 833, 845–867, 870–872, 874–877, 892–894, 901–903, 907–911, 913, 915–917

runtime entry point

A function interface provided by an OpenMP runtime for use by a tool. A runtime entry point is typically not associated with a global function symbol. **701**, 24, 49, 78, 93, 697, 704, 705, 745, 786

runtime error termination

An error termination that is performed during execution. **6**, 50, 149, 283, 285, 296, 389, 450, 451, 600, 602, 603, 689, 887

S

`safesync`-compatible expression

An expression that is **`omp_curr_progress_width`**, a constant expression, or an expression for which all operands are **`safesync`**-compatible expressions. 93, 393

saved data environment

For a given replayable construct that is recorded in a taskgraph record, an associated enclosing data environment that is also saved in the record for possible use in a replay execution of the construct. **436**, 103, 215, 435, 437

scalar variable

For C/C++, a scalar-variable, as defined by the base language. For Fortran, a scalar variable with enum, enumeration, assumed, or intrinsic type, excluding character type, as defined by the base language. 185, 189, 195, 200, 211, 214, 223, 231, 277, 292, 778, 888, 912

scan computation

A computation performed in the logical iterations of a loop nest that yields a set of values that are a running total, as defined by a reduction operation, over an input set of values. **267**, 50, 59–61, 94, 111, 253, 254, 267

scan phase

The portion of an affected iteration that includes all statements that read the result of a scan computation. **267**, 60, 267–270

schedulable task

A member of the schedulable task set of a thread. 448, 449

schedulable task set

If the thread is a structured thread, the set of tasks bound to the current team. If the thread is an unassigned thread, any explicit task in the contention group associated with the current OpenMP thread pool. 94, 447, 448

schedule specification

The specification of a loop schedule for a given loop-nest-associated construct, which includes but is not limited to the schedule type and chunk size. **404**, 94, 205, 404

schedule-specification clause

A clause that has the schedule-specification property. **404**

schedule-specification property

The property of a clause that it defines, in part or in full, the schedule specification of a given loop-nest-associated construct. 94, 397, 418, 422

schedule type

The part of a schedule specification that identifies the method by which the collapsed iterations are distributed to threads. 94, 117, 125, 134, 415, 419, 537, 573, 574, 892

scope handle

A handle that refers to an OpenMP scope. **827**, 875–877

segment

A portion of an address space associated with a set of address ranges. 20, 826

selector set

Unless specifically stated otherwise, a trait selector set. 36, 45, 58, 102, 111, 322

self map

A mapping operation for which the corresponding storage is the same as its original storage. **284**, 84, 283, 285, 361, 900

semantic requirement set

A logical set of semantic properties maintained by a task that is updated by directives in the scope of the task region. **328**, 332, 334, 338, 339, 482

separated construct

A construct for which its associated structured block is split into multiple structured block sequences by a separating directive. **154**, 95, 154, 155, 267, 268

separating directive

A directive that splits a structured block that is associated with a construct, the separated construct, into multiple structured block sequences. **154**, 95, 152, 154, 155, 266, 268, 408

sequentially consistent atomic operation

An atomic operation that is specified by an **`atomic`** construct for which the **`seq_cst`** clause is specified. 13, 914

sequential part

All code encountered during the execution of an initial task region that is not part of a **`parallel`** region that corresponds to a **`parallel`** construct or a **`task`** region corresponding to a **`task`** construct. Instead, it is enclosed by an implicit parallel region.

> COMMENT: Executable statements in called procedures may be in both a sequential part and any number of explicit **`parallel`** regions at different points in the program execution.

95, 216, 683, 685

shape-operator

For C/C++, an array shaping operator that reinterprets a pointer expression as an array with one or more specified dimensions. **165**, 165, 295, 444, 509, 909

shared attribute

For a given construct, a data-sharing attribute of a data entity that its lifetime is not limited to that of the corresponding region and, if the data entity is a variable, it is visible to all tasks generated by the construct in addition to being visible in the enclosing context of the construct if declared outside the construct. **225**, 8, 96, 210–214, 225, 252–254, 259, 427, 430, 454, 456, 461, 466, 888

shared variable

A variable that has the shared attribute with respect to a given construct. **7**, 7, 10–12, 14, 488–491

sharing task

A task for which the implicitly determined data-sharing attribute is shared unless explicitly specified otherwise. **213**, 96, 458

sharing-task property

The property that a task-generating construct generates sharing tasks. 458

sibling task

Two tasks are each a sibling task of the other if they are child tasks of the same task region. 96, 507, 508

signal

A software interrupt delivered to a thread. 24, 96, 817

signal handler

A function called asynchronously when a signal is delivered to a thread. 7, 24, 720, 786, 817

SIMD

Single Instruction, Multiple Data, a lock-step parallelization paradigm. 233, 318, 341, 342, 402, 888, 889, 914

SIMD chunk

A set of iterations executed concurrently, each by a SIMD lane, by a single thread by means of SIMD instructions. **399**, 97, 342, 399, 401, 912

SIMD construct

A **`simd`** construct or a compound construct for which the **`simd`** construct is a constituent construct. 419

SIMD instruction

A single machine instruction that can operate on multiple data elements. 3, 83, 96, 97, 300, 399

SIMDizable construct

A construct that has the SIMDizable property. 399, 917

SIMDizable property

The property that a construct may be encountered during execution of a **`simd`** region. 97, 374, 375, 377, 379–381, 399, 423, 494, 515, 516

SIMD lane

A software or hardware mechanism capable of processing one data element from a SIMD instruction. 5, 7, 87, 96, 219–221, 226, 233, 234, 250–253, 258, 399

SIMD loop

A loop that includes at least one SIMD chunk. 299, 341, 342

SIMD-partitionable construct

A construct that has the SIMD-partitionable property. 526

SIMD-partitionable property

The property of a loop-nest-associated construct that it partitions the set of affected iterations such that each partition can be divided into SIMD chunks. 97, 416, 417, 420, 429

simple lock

A lock that cannot be set if it is already owned by the task trying to set it. **663**, 97, 559, 663, 670

simple lock property

The property that a routine operates on simple locks. **663**, 97, 664, 666, 668, 670, 673, 675

simple lock routine

A routine that has the simple lock property. **663**, 559

simple modifier

A modifier that can never take an argument when it is specified. **158**, 158, 160, 161

simply contiguous array section

An array section that can be determined to have contiguous storage at compile time. In Fortran, this determination may result from the specification of the **`CONTIGUOUS`** attribute on the declaration of the array. 214, 888

simply happens before

For an event *A* to simply happen before an event *B*, *A* must precede *B* in simply happens-before order. **12**, 12, 13

simply happens-before order

An ordering relation that is consistent with program order and the synchronizes-with relation. **12**, 56, 97

sink iteration

A doacross iteration for which executable code, because of a doacross dependence, cannot execute until executable code from the source iteration has completed. **512**, 47

socket

The physical location to which a single chip of one or more cores of a device is attached. 128

soft pause

An instance of a resource-relinquishing routine that specifies that the OpenMP state is required to persist. **564**, 564

source iteration

A doacross iteration for which executable code must complete execution before executable code from another doacross iteration can execute due to a doacross dependence. **512**, 47, 98

stand-alone directive

A unassociated directive that is also an executable directive. **153**, 155, 156

standard trace format

A format for OMPT trace records. **704**, 710, 728, 812, 894

starting address

The address of the first storage location of a list item or, for a mapped variable of its original list item. 51, 70, 281

static context selector

The context selector for which traits in the OpenMP context can be fully determined at compile time. 48, 324, 326, 329

static storage duration

For C/C++, the lifetime of an object with static storage duration, as defined by the base language. For Fortran, the lifetime of a variable with a **SAVE** attribute, implicit or explicit, a common block object or a variable declared in a module. 8, 25, 44, 55, 65, 106, 211, 214, 215, 218, 224, 242, 274, 282, 287, 290, 291, 298, 302, 305, 309, 311, 345, 360, 361, 436, 437, 461, 885

step expression

A loop-invariant expression used by an induction operation. 32, 60, 64, 171, 243, 244, 248, 264

storage block

The physical storage that corresponds to an address range in memory. 9, 19, 38, 52, 69, 72, 82, 87, 99, 112, 463, 891

storage location

A storage block in memory. 7–9, 19, 25, 26, 49, 65, 89, 98, 188, 193–195, 233, 236, 237, 256, 259, 281, 308, 360, 401, 435, 494–497, 499, 500, 508, 509, 607, 715, 888, 891

strictly nested region

A region nested inside another region with no other explicit region nested between them. 81, 105, 395, 396, 398, 421, 425, 582, 585, 600, 602, 901, 917

strictly structured block

A single Fortran **`BLOCK`** construct, with a single entry at the top and a single exit at the bottom. 99, 153, 411

string literal

For C/C++, a string literal. For Fortran, a character literal constant. 53, 140, 469, 471

striping

The reordering of logical iterations of a loop that follows a grid while skipping logical iterations in-between. **379**, 901

strong flush

A flush that has the strong flush property. **10**, 10, 11, 13, 53, 496, 499

strong flush property

A flush with the strong flush property flushes a set of variables from the temporary view of the memory of the current thread to the memory. 52, 99, 499

structure

A structure is a variable that contains one or more variables that may have different types. This includes variables that have a **`struct`** type in C/C++, variables that have a **`class`** type in C++, and variables that have a derived type and are not arrays in Fortran. 36, 99, 212, 214, 238, 276, 278, 280, 282, 283, 287, 288, 296, 298, 299, 315, 462, 545, 698, 700, 707, 715, 718, 719, 725, 727, 728, 731, 734, 744–746, 754, 761, 798, 812, 819, 820, 823, 824, 831, 888, 909, 912

structured block

For C/C++, an executable statement, possibly compound, with a single entry at the top and a single exit at the bottom, or an OpenMP construct. For Fortran, a strictly structured block or a loosely structured block. **186**, 3, 7, 29, 35, 37, 43, 67, 82, 95, 109, 110, 132, 153–155, 186, 187, 198, 202, 236–239, 271, 273, 342, 371, 382, 384, 385, 395, 402, 405–407, 409, 410, 412–414, 421, 426, 427, 435, 439, 447, 458, 459, 474, 479, 502, 503, 516, 590, 705, 725, 741, 744, 754, 767, 768, 882, 890

structured block sequence

For C/C++, a sequence of zero or more executable statements (including constructs) that together have a single entry at the top and a single exit at the bottom. For Fortran, a block of zero or more executable constructs (including OpenMP constructs) with a single entry at the top and a single exit at the bottom. 29, 47, 49, 91, 95, 101, 154, 186, 198, 202, 230, 231, 267–270, 407–409, 890

structured parallelism

Parallel execution through the implicit tasks of (possibly nested) parallel regions by the set of structured threads in a contention group. 142, 143

structured thread

A thread that is assigned to a team and is not a free-agent thread. 94, 100, 107, 117, 142, 387, 897

subroutine

A procedure for which a call cannot be used as the right-hand side of a base language assignment operation. 554, 556, 568, 572–575, 582, 584, 589, 592, 599, 601, 608, 638–641, 646, 652, 661, 664–671, 673, 674, 680, 682, 683, 685, 692, 711, 722, 746–753, 755–757, 759–764, 766–769, 772–777, 780, 782, 784, 801, 807

subsidiary directive

A directive that is not an executable directive and that appears only as part of a construct. 152, 156, 266–268, 408, 429, 434, 435, 901

subtask

A portion of a task region between two consecutive task scheduling points in which a thread cannot switch from executing one task to executing another task. **5**, 5, 448, 449

successor task

For a given task, a dependent task of that task, or any successor task of a dependent task of that task. **507**, 100

supported active levels

An implementation defined maximum number of active levels of parallelism. 575, 576, 885

supported device

The host device or any non-host device supported by the implementation, including any device-related requirements specified by the **`requires`** directive. 119, 139–141, 450

synchronization construct

A construct that orders the completion of code executed by different threads. **472**, 2, 6, 522, 760

synchronization hint

An indicator of the expected dynamic behavior or suggested implementation of a synchronization mechanism. **561**, 472, 561, 562, 663, 893, 911

synchronizes with

For an event *A* to synchronize with an event *B*, a synchronizes-with relation must exist from *A* to *B*. **12**, 11, 12, 19, 502–504

synchronizes-with relation

An asymmetric relation that relates a release flush to an acquire flush, or, for C/C++, any pair of events *A* and *B* such that *A* "synchronizes with" *B* according to the base language, and establishes memory consistency between their respective executing threads. **10**, 92, 98, 101

synchronizing-region callback

A callback that has the synchronizing-region property. **763**, 764

synchronizing-region property

The property that a callback indicates the beginning or end of a synchronization-related region. **763**, 101, 763, 764

synchronizing threads

Two threads are synchronizing if the completion of a structured block sequence by one of the threads requires that it first observes a modification by the other thread, including the modification to an internal synchronization variable that an implementation performs for implicit flush synchronization as described in Section 1.3.5. 6, 7, 101, 362, 393

T

`target`-consistent clause

A clause for which all expressions that are specified on it are **`target`**-consistent expressions. 396

`target`-consistent expression

An expression that has the **`target`**-consistent property. 101, 396

`target`-consistent property

The property of an expression that its evaluation results in the same value when used on an immediately nested construct of a **`target`** construct as when specified on that **`target`** construct. 101, 179, 397, 452

target device

A device with respect to which the current device performs an operation, as specified by a device construct or device memory routine. **451**, 3, 4, 14, 40, 43, 45, 69, 102, 115, 116, 236, 237, 239, 257, 275, 283–286, 295, 298, 319, 361, 450, 451, 453, 454, 456, 462, 466, 592, 593, 603, 604, 607, 608, 610, 611, 697, 701, 704–706, 721, 722, 772, 773, 778, 779, 781, 785, 800, 803–805, 807, 814, 894, 899, 909

`target_device` selector set

A selector set that may match the target device trait set. **321**, 321–323, 906

target device trait set

The trait set that consists of traits that define the characteristics of a device that the implementation supports. **319**, 102, 318, 319, 321, 323, 897

target memory space

A memory space that is associated with at least one device that is not the current device when it is created. **630**, 307, 645, 647

target task

A mergeable untied task that is generated by a device construct or a call to a device memory routine and that coordinates activity between the current device and the target device. 3, 257, 286, 454–457, 461, 462, 465, 466, 501, 503, 603, 604, 613, 619, 719, 756, 760, 778, 781, 785, 798

target variant

A version of a device procedure that can only be executed as part of a **`target`** region. 318

task

A specific instance of executable code and its data environment that the OpenMP implementation can schedule for execution by a team. 3–9, 21, 22, 28, 30, 38, 42, 44, 50–52, 54, 55, 57, 59, 60, 62, 65, 66, 73–75, 81, 83, 86, 87, 91, 93–97, 100, 102–104, 106–110, 115–117, 124, 132, 134, 181, 216, 219–221, 225–227, 250, 251, 253, 256–258, 281, 284–287, 301, 305, 306, 328, 384, 386, 387, 390, 393, 395, 403, 405, 406, 408–410, 413, 414, 421, 426–430, 432–436, 439–445, 447–449, 453, 458, 459, 468, 469, 473–476, 478–480, 482, 494, 496, 497, 502–504, 507, 509, 513, 515, 516, 521, 522, 524, 531, 534, 559, 571, 574, 584–586, 590, 601, 602, 663–673, 719, 720, 722, 733, 740–742, 744, 755–760, 762, 786, 798, 799, 827, 831, 832, 860–862, 864–866, 882, 890, 891, 901, 902, 907, 914–916

task completion

A condition that is satisfied when a thread reaches the end of the executable code that is associated with the task and any *allow-completion* event that is created for the task has been fulfilled. 104, 426

task dependence

A dependence between two dependence-compatible tasks: the dependent task and an antecedent task. The task dependence is fulfilled when the antecedent task has completed. **504**, 42, 103, 108, 448, 505, 507, 509, 511, 559, 586, 604, 715, 716, 902, 907, 914

task-generating construct

A construct that has the task-generating property. **5**, 52, 54, 73, 91, 92, 96, 103, 124, 132, 211, 213, 214, 427, 435, 437, 441, 458, 508, 509, 527, 898, 901, 909, 917

task-generating property

The property that a construct generates one or more explicit tasks that are child tasks of the encountering task. 103, 426, 429, 454, 456, 458, 460, 465

taskgraph-altering clause

A clause that has the taskgraph-altering property. 435–437

taskgraph-altering property

The property of a clause that if it appears on a replayable construct, it affects the resulting number of tasks or the resulting task dependences in a replay execution of a taskgraph record. 103, 432, 433, 507

taskgraph record

For a given **`taskgraph`** construct that is encountered on a given device, a data structure that contains a sequence of recorded replayable constructs, with their respective saved data environments, that are encountered while executing the corresponding **`taskgraph`** region. **435**, 52, 92, 94, 103, 435–438, 891

taskgroup set

A set of tasks that are logically grouped by a **`taskgroup`** region, such that a task is a member of the taskgroup set if and only if its **`task`** region is nested in the **`taskgroup`** region and it binds to the same parallel region as the **`taskgroup`** region. 29, 103, 478, 521

task handle

A handle that refers to a task region. 828, 860–863, 866, 869

task-inherited clause

A clause that has the task-inherited property. **434**

task-inherited property

The property of a clause that if it appears on a **`task_iteration`** directive, it will be inherited by the tasks that are generated by a task-generating construct. 103, 444, 507

taskloop-affected loop

A collapsed loop of a **taskloop** construct. 171, 431, 434

task priority

A hint for the task execution order of tasks generated by a construct. **443**, 143, 443, 912, 913

task reduction

A reduction that is performed over a set of tasks that may include explicit tasks. **256**, 253, 256, 909

task region

A region consisting of all code encountered during the execution of a task. 4–6, 8, 38, 42, 83, 96, 100, 107, 110, 216, 227, 384, 385, 394, 448, 449, 454, 456, 458, 466, 501, 521, 588, 672, 715, 719, 722, 756, 798, 860, 864, 882

task scheduling point

A point during the execution of the current task region at which the task can be suspended to be resumed later; or the point of task completion, after which the executing thread may switch to a different task. **447**, 5, 100, 216, 250, 385, 427, 446–449, 475, 476, 478, 479, 495, 500, 501, 612, 618, 741, 757, 914

task synchronization construct

A **taskwait**, a **taskgroup**, or a **barrier** construct. 5, 426, 448

team

A set of one or more assigned threads assigned to execute the set of implicit tasks of a parallel region. **4**, 3, 4, 7, 19, 26, 38, 59, 61, 64, 81, 83, 87, 88, 100, 102, 105, 106, 109, 110, 113, 114, 116, 125, 132, 133, 216, 234, 253, 254, 259, 270, 271, 273, 362, 384, 385, 389–395, 397, 402–410, 414, 415, 418–423, 425, 453, 473, 475, 476, 495, 502, 503, 516, 523, 569, 570, 581, 583, 599, 600, 725, 733, 749, 758, 785, 796, 797, 829, 854, 858–860, 863, 887, 890, 891, 906, 907, 915–917

team-executed construct

A construct that has the team-executed property. **4**

team-executed property

The property that a construct gives rise to a team-executed region. 104, 405–407, 409, 416, 417, 423, 475

team-executed region

A region that is executed by all or none of the threads in the current team. **4**, 104, 917

team-generating construct

A construct that has the team-generating property. 917

team-generating property

The property that a construct generates a parallel region. 105, 384, 394, 460

team number

A number that the OpenMP implementation assigns to an initial team. If the initial team is not part of a league formed by a **`teams`** construct then the team number is zero; otherwise, the team number is a non-negative integer less than the number of initial teams in the league. 105, 117, 422, 583, 758

`teams`-nestable construct

A construct that has the **`teams`**-nestable property. 396, 917

`teams`-nestable property

The property that a construct or routine generates a region that may be a strictly nested region of a **`teams`** region. 105, 374, 375, 377, 379–381, 384, 420, 423, 581, 582

`teams`-nestable routine

A routine that has the **`teams`**-nestable property. 396, 917

team-worker thread

A thread that is assigned to a team but is not the primary thread. It executes one of the implicit tasks that is generated when the team is formed for an active parallel region. **4**, 113, 132

temporary view

The state of memory that is accessible to a particular thread. **7**, 7, 10, 11, 499

third-party tool

A tool that executes as a separate process from the process that it is monitoring and potentially controlling. **816**, 15, 46, 77, 116, 816–818, 820–823, 826, 829–831, 833, 835, 836, 841, 843, 845, 846, 851, 878, 911

thread

Unless specifically stated otherwise, an OpenMP thread. 3–8, 10–15, 19, 22, 23, 25, 26, 28, 35, 38, 40, 47, 49, 50, 52–54, 61, 62, 67, 71, 81, 83, 84, 87, 88, 92–94, 96, 99–102, 104–107, 109, 110, 113, 115–117, 119, 128–130, 134–136, 138, 142, 143, 149, 205, 215–217, 227, 229, 234, 250–252, 254, 259, 270, 271, 273, 286, 305–308, 346, 352, 353, 360, 366, 384–395, 402–415, 418–421, 423–427, 429–431, 435, 439, 442, 446–449, 453, 455, 457, 462, 466, 472–478, 480, 482, 494–497, 499–504, 509, 513–516, 520–524, 534, 561, 568–573, 579, 584, 585, 590, 601, 602, 607–613, 618, 619, 664–669, 671–678, 681, 692, 695, 697, 701, 706, 715, 725, 733, 734, 742, 747, 749, 754, 758, 765, 769, 781, 786, 791, 793, 795–799, 802, 812, 813, 821, 830–833, 836, 837, 839, 841, 845, 854, 855, 858–862, 864, 871, 878, 886–888, 890, 891, 900, 901, 907, 911, 914–917

thread affinity

A binding of threads to places within the current place partition. **389**, 84, 115, 116, 132, 133, 136–138, 216, 389–392, 678, 686, 687, 886, 890, 909, 913

thread-exclusive construct

A construct that has the thread-exclusive property. 917

thread-exclusive property

The property that a construct when encountered by multiple threads in the current team is executed by only one thread at a time. 106, 473, 515

thread-limiting construct

A construct that has the thread-limiting property. 149

thread-limiting property

For C++, the property that a construct limits the threads that can catch an exception thrown in the corresponding region to the thread that threw the exception. 106, 384, 394, 402, 405–407, 426, 460, 473, 515

thread number

For an assigned thread, a non-negative number assigned by the OpenMP implementation. For threads within the same team, zero identifies the primary thread and subsequent consecutive numbers identify any worker threads of the team. For an unassigned thread, the thread number is the value **`omp_unassigned_thread`**. **384**, 87, 106, 117, 216, 384, 390, 393, 403, 418, 569, 578, 758, 798, 854, 916

thread-pool-worker thread

A thread in an OpenMP thread pool that is not the initial thread. 742

threadprivate attribute

For a given OpenMP thread, a data-sharing attribute of a data entity that it has static storage duration, or thread storage duration for C/C++, and is visible only to tasks that are executed by the thread. **215**, 107, 211, 214, 217, 219, 271–274, 915

threadprivate memory

The set of threadprivate variables associated with each thread. **7**, 217, 448, 888

threadprivate variable

A variable that has the threadprivate attribute with respect to a given OpenMP thread. **215**, 106, 215–219, 270, 271, 398, 413, 462

thread-reservation type

A categorization of a thread as either a structured thread or a free-agent thread. **141**, 92, 93

thread-safe procedure

A procedure that performs the intended function even when executed concurrently (by multiple native threads). 15

thread-selecting construct

A construct that has the thread-selecting property. 526, 527

thread-selecting property

The property that a construct selects a subset of threads that can execute the corresponding region from the binding thread set of the region. 107, 402, 405

thread-set

The set of threads for which a flush may enforce memory consistency. **10**, 10, 12, 13, 494, 499, 501

thread state

The state associated with a thread, which may be represented by an enumeration type that describes the current OpenMP activity of a thread. Only one of the enumeration values can apply to a thread at any time. 5, 14, 697, 700, 701, 733, 788, 795, 870, 871, 894

tied task

A task that, when its task region is suspended, can be resumed only by the same thread that was executing it before suspension. That is, the task is tied to that thread. **5**, 4, 384, 439, 448

tile

For a **`tile`** directive, the logical iteration space of the tile loops. **381**, 33, 84, 107, 381, 383

tile loop

The inner generated loops of a **`tile`** construct that iterate over the logical iterations that correspond to a tile. **380**, 107, 380, 381, 383, 889, 901

tool

Code that can observe and/or modify the execution of an application. 2, 14, 15, 17, 52, 91, 93, 105, 108, 117, 144–146, 453, 459, 565–567, 614–616, 618, 620, 621, 689, 694, 695, 697–701, 703–706, 715, 720, 722, 726, 731, 733, 744–751, 753, 754, 756–779, 781, 783, 785–796, 798–814, 833–855, 858–860, 865, 867–870, 872–874, 876, 877, 894

tool callback

A procedure that a tool provides to an OpenMP implementation to invoke when an associated event occurs. 14, 29, 476, 513, 531, 705, 744, 808, 894

tool context

An opaque reference provided by a tool to an OMPD library. A tool context uniquely identifies an abstraction. 20, 75, 108, 834, 840

tool defined

Behavior that must be documented by the tool implementation, and is allowed to vary among different compliant tools. 566, 695, 771

trace record

A data structure in which to store information associated with an occurrence of an event. 45, 75, 98, 110, 184, 185, 704–706, 710, 725, 726, 728, 744, 761, 773, 775–779, 781, 783–785, 803, 805, 806, 808, 810–814, 894, 896

trait

An aspect of an OpenMP implementation or the execution of an OpenMP program. 9, 21, 31, 37, 45, 48, 51, 58, 75, 76, 79, 98, 102, 108, 111, 139, 140, 144, 304–309, 313, 316, 318–323, 337, 355, 546, 547, 555, 638, 645, 654, 655, 889, 897, 899, 900, 906, 910

trait selector

A member of a trait selector set. **320**, 318, 321–325, 330, 337

trait selector set

A set of traits that are specified to match the trait set at a given point in an OpenMP program. **320**, 95, 108, 322

trait set

A grouping of related traits. **318**, 36, 45, 48, 58, 102, 108, 318, 321, 323

transformation-affected loop

For a loop-transforming construct, an affected loop that is replaced according to the semantics of the constituent loop-transforming directive. 205, 369–371, 375–383

transparent task

A task for which child tasks are visible to external dependence-compatible tasks for the purposes of establishing task dependences. Unless otherwise specified, a transparent task is both an importing task and an exporting task. **511**, 108, 437

type-name list

An argument list that consists of type-name list items. **162**, 169, 260, 261, 293

type-name list item

A list item that is the name of a type. **163**, 108, 162–164, 263, 264

U

ultimate property

The property that a clause or an argument must be the lexically last clause or argument to appear on the directive. For a modifier, the property that it must be the lexically last modifier to appear on a pre-modified clause or that it must be the lexically first modifier to appear on a post-modified clause. **161**, 159, 161, 207, 251, 252, 255, 256, 258, 300, 326, 397, 418, 422

unassigned thread

A thread that is not currently assigned to any team. **3**, 3, 4, 53, 57, 94, 106, 442, 448, 569, 734

unassociated directive

A directive that is not directly associated with any base language code. **152**, 98, 152–155, 260, 263, 293, 327, 352, 355, 368, 369, 434, 446, 454, 456, 465, 468, 475, 479, 498, 505, 514, 520, 524

undeferred task

A task for which execution is not deferred with respect to its generating task region. That is, its generating task region is suspended until execution of the structured block associated with the undeferred task is completed. **427**, 59, 73, 109, 427, 430, 437, 440, 503

undefined

For variables, the property of not being defined; that is, the variable does not have a valid value. 9, 147, 522, 742, 790, 793, 794, 796, 798–800

underlying map type

The map type that determines which output map type results from an input map type. **275**, 70, 275

unified address space

An address space that is used by all devices. **359**

uninitialized state

The lock state that indicates the lock must be initialized before it can be set. 65, 639, 641, 664, 668, 670, 675

union

A union is a type that defines one or more fields that overlap in memory, so only one of the fields can be used at any given time. For C/C++, implemented using union types. For Fortran, implemented using derived types. 109, 708, 710, 714

unique property

The property that a clause, a modifier, or an argument may appear at most once in a given context with which it is associated. For a clause set, each member of the clause set may appear at most once in the given context. **160**, 159–161, 169, 173, 179–182, 205–207, 223, 225–227, 230, 232, 235–238, 252, 255, 256, 258, 262, 263, 265, 266, 269–272, 278–280, 289–291, 297–300, 303, 309, 310, 313, 316, 325, 326, 330, 331, 333, 339, 340, 343, 344, 350, 353, 354, 356–367, 372, 374, 376, 378, 382, 383, 388, 392, 393, 397, 398, 400–403, 418, 422, 424, 425, 432, 433, 438–445, 450–452, 470, 472, 481, 483–493, 504, 506, 507, 510–512, 517–519

unit of work

In constructs that use units of work, one or more executable statements that will be executed by a single thread and are part of the same structured block. A structured block can consist of one or more units of work; the number of units of work into which a structured block is split is allowed to vary among different compliant implementations. 110, 409, 410, 412, 413, 753

unlocked state

The lock state that indicates the lock can be set by any task. **663**, 65, 66, 663, 664, 668, 670, 672–674

unsigned property

The property that a routine or callback either returns an unsigned type in C/C++ or has an argument that has such a type. 698, 749, 757, 765, 782, 784, 805

unspecified behavior

A behavior or result that is not specified by the OpenMP specification or not known prior to the compilation or execution of an OpenMP program. Unspecified behavior may result from:

- Issues that this specification documents as having unspecified behavior;
- A non-conforming program; or
- A conforming program exhibiting an implementation defined behavior.

7–9, 34, 40, 57, 110, 149, 218, 222, 228, 237, 243, 247, 294, 303, 306, 313, 359, 362, 443, 444, 461, 463, 477, 510, 522, 561, 592–594, 596–603, 607, 610, 611, 622, 629, 645, 646, 655, 662, 663, 682, 683, 685–687, 693, 802

untied task

A task that, when its task region is suspended, can be resumed by any thread in the team. That is, the task is not tied to any thread. **5**, 102, 217, 427, 439, 448, 914

untraced-argument property

The property of an argument of a callback that it is omitted from the corresponding trace record of the callback. 746, 749, 755, 769, 770, 777, 780, 784

update-capture structured block

An atomic structured block that may be associated with an **atomic** directive that expresses an atomic captured update operation. **192**, 192, 193, 497

update structured block

An atomic structured block that may be associated with an **atomic** directive that expresses an atomic update operation. **190**, 34, 35, 190–192

update value

The update value of a new list item used for a scan computation is, for a given logical iteration, the value of the new list item on completion of its input phase. **267**, 111, 267

use-device-addr attribute

For a given device construct, a data-sharing attribute of a data entity that refers to an object in a device data environment that corresponds to the data entity of the same name in the enclosing data environment of the construct if such an object exists, and otherwise refers to the entity in the enclosing data environment. **238**

use-device-ptr attribute

For a given device construct, a data-sharing attribute of a C pointer variable that implies the private attribute, and additionally the variable is initialized to be a device pointer that refers to the device address that corresponds to the value of a C pointer of the same name in the enclosing data environment of the construct. **236**

user-defined cancellation point

A cancellation point that is specified by a **cancellation point** construct. **524**, 524

user-defined induction

An induction operation that is defined by a **declare_induction** directive. **263**, 264–266, 898

user-defined mapper

A mapper that is defined by a **declare_mapper** directive. **293**, 70, 86, 183, 281, 294–296, 904

user-defined reduction

A reduction operation that is defined by a **declare_reduction** directive. **260**, 260, 262, 263, 523, 914

`user` selector set

A selector set that may match traits in the dynamic trait set. **321**, 321–323

utility directive

A directive that facilitates interactions with the compiler and/or supports code readability. A utility directive is an informational directive except when specified to be an executable directive. **352**, 112, 152, 352, 353, 369

V

value property

The property that a routine parameter does not have a pointer type in C/C++ and has the **VALUE** attribute in Fortran. **535**, 554, 604–609, 611, 613, 614, 616, 617, 619, 620, 656–661, 734, 770, 774, 777

variable

A referencing variable or a named data storage block, for which the value can be defined and redefined during the execution of a program; for C/C++, this includes `const`-qualified types when explicitly permitted.

COMMENT: An array element or structure element is a variable that is part of an aggregate variable.

7–13, 15, 20, 25–27, 31, 36–42, 44, 52, 53, 55, 60, 62–65, 70, 71, 74, 82, 87, 90, 96, 98, 99, 101, 107, 109, 111, 112, 115, 153–155, 163, 164, 169, 181–185, 187, 189, 199, 201, 205, 210–225, 227–231, 234, 238, 240–244, 248, 254, 258–261, 264, 270–278, 281–283, 286–292, 294, 301–305, 308, 309, 311, 312, 315, 316, 322, 325, 329, 331, 340, 341, 345–350, 360, 361, 371, 388, 394, 403, 404, 414, 418, 422, 424, 427, 430, 436, 437, 441, 445, 450, 454, 456, 459, 461, 463, 464, 466, 499, 500, 511, 513, 528–530, 663, 704, 742, 778, 788–790, 795–800, 817, 818, 830, 834, 836, 851, 885, 888, 899, 904, 906, 909, 910, 912, 915

variable list

An argument list that consists of variable list items. **162**, 51, 249, 313

variable list item

For C/C++, a list item that is a variable or an array section; for Fortran, a list item that is a named item specifically identified in Section 5.2.1. **163**, 51, 112, 162–164, 435, 437

variant-generating directive

A declarative directive that has the variant-generating property. 325

variant-generating property

The property that a declarative directive generates a variant of a procedure. 112, 341, 346, 349

variant substitution

The replacement of a call to a base function by a call to a function variant. 54, 329, 338, 906

W

wait identifier

A unique handle associated with each data object (for example, a lock) that the OpenMP runtime uses to enforce mutual exclusion and potentially to cause a thread to wait actively or passively. **742**, 742, 795

white space

A non-empty sequence of space and/or horizontal tab characters. 46, 127, 134, 135, 137, 150, 155–158, 172, 173, 526, 896

work distribution

The manner in which execution of a region that corresponds to a work-distribution construct is assigned to threads. 206

work-distribution construct

A construct that has the work-distribution property. **404**, 2, 84, 113, 114, 227, 229, 231, 254, 404, 405, 423, 752

work-distribution property

The property that a construct is cooperatively executed by threads in the binding thread set of the corresponding region. 113, 405–407, 409, 412, 416, 417, 420, 423

work-distribution region

A region that corresponds to a work-distribution construct. 229, 231, 404, 405

worker thread

Unless specifically stated otherwise, a team-worker thread. 106, 385

worksharing construct

A construct that has the worksharing property. **404**, 4, 84, 113, 114, 228, 229, 234, 252–254, 259, 407, 414, 423, 477, 523, 527, 528, 733

worksharing-loop construct

A construct that has the worksharing-loop property. **414**, 47, 48, 114, 134, 254, 259, 414–419, 514–516, 521, 523, 526, 753, 890, 899, 905, 910, 912, 916, 918

worksharing-loop property

The property of a worksharing construct that it is a loop-nest-associated construct that distributes the collapsed iterations of the affected loops among the threads in the team. 113, 416, 417, 529

worksharing-loop region

A region that corresponds to a worksharing-loop construct. **414**, 117, 125, 414, 514, 516, 918

worksharing property

The property of a construct that it is a work-distribution construct that is executed by the team of the innermost enclosing parallel region and includes, by default, an implicit barrier. 113, 405–407, 409, 416, 417, 423

worksharing region

A region that corresponds to a worksharing construct. **404**, 4, 228, 229, 252, 404, 476, 501, 753, 907, 917

write-capture structured block

An atomic structured block that may be associated with an **`atomic`** directive that expresses an atomic write operation with capture semantics. **192**, 193

write structured block

An atomic structured block that may be associated with an **`atomic`** directive that expresses an atomic write operation. **190**, 190, 192, 497

Z

zeroed-memory-allocating routine

A memory-allocating routine that has the zeroed-memory-allocating-routine property. **654**, 654, 658, 659

zeroed-memory-allocating-routine property

The property that a memory-allocating routine returns a pointer to memory that has been set to zero. **654**, 114, 658, 659

zero-length array section

An array section that does not include any elements of the array. 167, 247, 280, 509

zero-offset assumed-size array

An assumed-size array for which the lower bound is zero. 236, 277, 282

3 Internal Control Variables

An OpenMP implementation must act as if internal control variables (ICVs) control the behavior of an OpenMP program. These ICVs store information such as the number of threads to use for future **`parallel`** regions. One copy exists of each ICV per instance of its ICV scope. Possible ICV scopes are: global; device; implicit task; and data environment. If an ICV scope is global then one copy of the ICV exists for the whole OpenMP program. If an ICV scope is device then a distinct copy of the ICV exists for each device. If an ICV scope is implicit task then a distinct copy of the ICV exists for each implicit task. If an ICV scope is data environment then a distinct copy of the ICV exists for the data environment of each task, unless otherwise specified. The ICVs are given values at various times (described below) during the execution of the program. They are initialized by the implementation itself and may be given values through OpenMP environment variables and through calls to OpenMP API routines. The program can retrieve the values of these ICVs only through routines.

For purposes of exposition, this document refers to the ICVs by certain names, but an implementation is not required to use these names or to offer any way to access the variables other than through the ways shown in Section 3.2.

3.1 ICV Descriptions

Section 3.1 shows the ICV scope and description of each ICV.

TABLE 3.1: ICV Scopes and Descriptions

ICV	Scope	Description
active-levels-var	data environment	Number of nested active parallel regions such that all active parallel regions are enclosed by the outermost initial task region on the device
affinity-format-var	device	Controls the thread affinity format when displaying thread affinity
available-devices-var	global	Controls target device availability and the device number assignment

ICV	Scope	Description
bind-var	data environment	Controls the binding of threads to places; when binding is requested, indicates that the execution environment is advised not to move threads between places; can also provide default thread affinity policies
cancel-var	global	Controls the desired behavior of the **cancel** construct and cancellation points
debug-var	global	Controls whether an OpenMP implementation will collect information that an OMPD library can access to satisfy requests from a third-party tool
def-allocator-var	implicit task	Controls the memory allocator used by memory allocation routines, directives and clauses that do not specify one explicitly
default-device-var	data environment	Controls the default target device
device-num-var	device	Device number of a given device
display-affinity-var	global	Controls the display of thread affinity
dyn-var	data environment	Enables dynamic adjustment of the number of threads used for encountered **parallel** regions
explicit-task-var	data environment	Boolean that is *true* if a given task is an explicit task, otherwise *false*
final-task-var	data environment	Boolean that is *true* if a given task is a final task, otherwise *false*
free-agent-thread-limit-var	data environment	Controls the maximum number of free-agent threads that may execute tasks in the contention group in parallel
free-agent-var	data environment	Boolean that is *true* if a free-agent thread is currently executing a given task, otherwise *false*
league-size-var	data environment	Number of initial teams in a league
levels-var	data environment	Number of nested parallel regions such that all parallel regions are enclosed by the outermost initial task region on the device
max-active-levels-var	data environment	Controls the maximum number of nested active parallel regions when the innermost active parallel region is generated by a given task
max-task-priority-var	global	Controls the maximum value that can be specified in the **priority** clause
nteams-var	device	Controls the number of teams requested for encountered **teams** regions

ICV	Scope	Description
nthreads-var	data environment	Controls the number of threads requested for encountered **parallel** regions
num-devices-var	global	Number of available non-host devices
num-procs-var	device	The number of processors available on the device
place-assignment-var	implicit task	Controls the places to which threads are bound
place-partition-var	implicit task	Controls the place partition available for encountered **parallel** regions
run-sched-var	data environment	Controls the schedule used for worksharing-loop regions that specify the **runtime** schedule type
stacksize-var	device	Controls the stack size for threads that the OpenMP implementation creates
structured-thread-limit-var	data environment	Controls the maximum number of structured threads that may execute tasks in the contention group in parallel
target-offload-var	global	Controls the offloading behavior
team-generator-var	data environment	Generator type of current team that refers to a construct name or the OpenMP program
team-num-var	data environment	Team number of a given thread
team-size-var	data environment	Size of the current team
teams-thread-limit-var	device	Controls the maximum number of threads that may execute tasks in parallel in each contention group that a **teams** construct creates
thread-limit-var	data environment	Controls the maximum number of threads that may execute tasks in the contention group in parallel
thread-num-var	data environment	Thread number of an implicit task within its current team
tool-libraries-var	global	List of absolute paths to tool libraries
tool-var	global	Indicates that a tool will be registered
tool-verbose-init-var	global	Controls whether an OpenMP implementation will verbosely log the registration of a tool
wait-policy-var	device	Controls the desired behavior of waiting native threads

3.2 ICV Initialization

Section 3.2 shows the ICVs, associated environment variables, and initial values.

TABLE 3.2: ICV Initial Values

ICV	Environment Variable	Initial Value
active-levels-var	(none)	*0 (zero)*
affinity-format-var	**OMP_AFFINITY_FORMAT**	implementation defined
available-devices-var	**OMP_AVAILABLE_DEVICES**	*See below*
bind-var	**OMP_PROC_BIND**	implementation defined
cancel-var	**OMP_CANCELLATION**	*false*
debug-var	**OMP_DEBUG**	*disabled*
def-allocator-var	**OMP_ALLOCATOR**	implementation defined
default-device-var	**OMP_DEFAULT_DEVICE**	*See below*
device-num-var	(none)	*0 (zero)*
display-affinity-var	**OMP_DISPLAY_AFFINITY**	*false*
dyn-var	**OMP_DYNAMIC**	implementation defined
explicit-task-var	(none)	*false*
final-task-var	(none)	*false*
free-agent-thread-limit-var	**OMP_THREAD_LIMIT**, **OMP_THREADS_RESERVE**	*See below*
free-agent-var	(none)	*false*
league-size-var	(none)	*1 (one)*
levels-var	(none)	*0 (zero)*
max-active-levels-var	**OMP_MAX_ACTIVE_LEVELS**, **OMP_NUM_THREADS**, **OMP_PROC_BIND**	implementation defined
max-task-priority-var	**OMP_MAX_TASK_PRIORITY**	*0 (zero)*
nteams-var	**OMP_NUM_TEAMS**	*0 (zero)*
nthreads-var	**OMP_NUM_THREADS**	implementation defined
num-devices-var	(none)	implementation defined
num-procs-var	(none)	implementation defined
place-assignment-var	(none)	implementation defined
place-partition-var	**OMP_PLACES**	implementation defined
run-sched-var	**OMP_SCHEDULE**	implementation defined
stacksize-var	**OMP_STACKSIZE**	implementation defined

ICV	Environment Variable	Initial Value
structured-thread-limit-var	**OMP_THREAD_LIMIT**, **OMP_THREADS_RESERVE**	*See below*
target-offload-var	**OMP_TARGET_OFFLOAD**	*default*
team-generator-var	(none)	*0 (zero)*
team-num-var	(none)	*0 (zero)*
team-size-var	(none)	*1 (one)*
teams-thread-limit-var	**OMP_TEAMS_THREAD_LIMIT**	*0 (zero)*
thread-limit-var	**OMP_THREAD_LIMIT**	implementation defined
thread-num-var	(none)	*0 (zero)*
tool-libraries-var	**OMP_TOOL_LIBRARIES**	*empty string*
tool-var	**OMP_TOOL**	*enabled*
tool-verbose-init-var	**OMP_TOOL_VERBOSE_INIT**	*disabled*
wait-policy-var	**OMP_WAIT_POLICY**	implementation defined

If an ICV has an associated environment variable and that ICV neither has global ICV scope nor is *default-device-var* then the ICV has a set of associated device-specific environment variables that extend the associated environment variable with the following syntax:

*<ENVIRONMENT VARIABLE>*_**ALL**

or

*<ENVIRONMENT VARIABLE>*_**DEV***[_<device>]*

where *<ENVIRONMENT VARIABLE>* is the associated environment variable and *<device>* is the device number as specified in the **device** clause (see Section 15.2); the semantic and precedence is described in Chapter 4.

Semantics

- The initial value of *available-devices-var* is the set of all accessible devices that are also supported devices.
- The initial value of *dyn-var* is implementation defined if the implementation supports dynamic adjustment of the number of threads; otherwise, the initial value is *false*.
- The initial value of *free-agent-thread-limit-var* is one less than the initial value of *thread-limit-var*.
- The initial value of *structured-thread-limit-var* is the initial value of *thread-limit-var*.
- If *target-offload-var* is **mandatory** and the number of available non-host devices is zero then *default-device-var* is initialized to **omp_invalid_device**. Otherwise, the initial value is an implementation defined non-negative integer that is less than or, if *target-offload-var* is not **mandatory**, equal to the value returned by **omp_get_initial_device**.

- The value of the *nthreads-var* ICV is a list.
- The value of the *bind-var* ICV is a list.

The host device and non-host device ICVs are initialized before any construct or routine executes. After the initial values are assigned, the values of any OpenMP environment variables that were set by the user are read and the associated ICVs are modified accordingly. If no device number is specified on the device-specific environment variable then the value is applied to all non-host devices.

Cross References

- **OMP_AFFINITY_FORMAT**, see Section 4.3.5
- **OMP_ALLOCATOR**, see Section 4.4.1
- **OMP_AVAILABLE_DEVICES**, see Section 4.3.7
- **OMP_CANCELLATION**, see Section 4.3.6
- **OMP_DEBUG**, see Section 4.6.1
- **OMP_DEFAULT_DEVICE**, see Section 4.3.8
- **OMP_DISPLAY_AFFINITY**, see Section 4.3.4
- **OMP_DYNAMIC**, see Section 4.1.2
- **OMP_MAX_ACTIVE_LEVELS**, see Section 4.1.5
- **OMP_MAX_TASK_PRIORITY**, see Section 4.3.11
- **OMP_NUM_TEAMS**, see Section 4.2.1
- **OMP_NUM_THREADS**, see Section 4.1.3
- **OMP_PLACES**, see Section 4.1.6
- **OMP_PROC_BIND**, see Section 4.1.7
- **OMP_SCHEDULE**, see Section 4.3.1
- **OMP_STACKSIZE**, see Section 4.3.2
- **OMP_TARGET_OFFLOAD**, see Section 4.3.9
- **OMP_TEAMS_THREAD_LIMIT**, see Section 4.2.2
- **OMP_THREAD_LIMIT**, see Section 4.1.4
- **OMP_TOOL**, see Section 4.5.1
- **OMP_TOOL_LIBRARIES**, see Section 4.5.2
- **OMP_WAIT_POLICY**, see Section 4.3.3

3.3 Modifying and Retrieving ICV Values

Section 3.3 shows methods for modifying and retrieving the ICV values. If *(none)* is listed for an ICV, the OpenMP API does not support its modification or retrieval. Calls to routines retrieve or modify ICVs with data environment ICV scope in the data environment of their binding task set.

TABLE 3.3: Ways to Modify and to Retrieve ICV Values

ICV	Ways to Modify Value	Ways to Retrieve Value
active-levels-var	(none)	**omp_get_active_level**
affinity-format-var	**omp_set_affinity_format**	
		omp_get_affinity_format
available-devices-var	(none)	(none)
bind-var	(none)	**omp_get_proc_bind**
cancel-var	(none)	**omp_get_cancellation**
debug-var	(none)	(none)
def-allocator-var	**omp_set_default_allocator**	
		omp_get_default_allocator
default-device-var	**omp_set_default_device**	
		omp_get_default_device
device-num-var	(none)	**omp_get_device_num**
display-affinity-var	(none)	(none)
dyn-var	**omp_set_dynamic**	**omp_get_dynamic**
explicit-task-var	(none)	**omp_in_explicit_task**
final-task-var	(none)	**omp_in_final**
free-agent-thread-limit-var	(none)	(none)
free-agent-var	(none)	**omp_is_free_agent**
league-size-var	(none)	**omp_get_num_teams**
levels-var	(none)	**omp_get_level**
max-active-levels-var	**omp_set_max_active_levels**	
		omp_get_max_active_levels
max-task-priority-var	(none)	**omp_get_max_task_priority**
nteams-var	**omp_set_device_num_teams**	
		omp_get_device_num_teams
	omp_set_num_teams	**omp_get_max_teams**
nthreads-var	**omp_set_num_threads**	**omp_get_max_threads**
num-devices-var	(none)	**omp_get_num_devices**
num-procs-var	(none)	**omp_get_num_procs**
place-assignment-var	(none)	(none)

ICV	Ways to Modify Value	Ways to Retrieve Value
place-partition-var	(none)	**omp_get_partition_num_places**, **omp_get_partition_place_nums**, **omp_get_place_num_procs**, **omp_get_place_proc_ids**
run-sched-var	**omp_set_schedule**	**omp_get_schedule**
stacksize-var	(none)	(none)
structured-thread-limit-var	(none)	(none)
target-offload-var	(none)	(none)
team-generator-var	(none)	(none)
team-num-var	(none)	**omp_get_team_num**
team-size-var	(none)	**omp_get_num_threads**
teams-thread-limit-var	**omp_set_device_teams_thread_limit**	**omp_get_device_teams_thread_limit**
	omp_set_teams_thread_limit	**omp_get_teams_thread_limit**
thread-limit-var	**thread_limit**	**omp_get_thread_limit**
thread-num-var	(none)	**omp_get_thread_num**
tool-libraries-var	(none)	(none)
tool-var	(none)	(none)
tool-verbose-init-var	(none)	(none)
wait-policy-var	(none)	(none)

Semantics

- The value of the *bind-var* ICV is a list. The **omp_get_proc_bind** routine retrieves the value of the first element of this list.
- The value of the *nthreads-var* ICV is a list. The **omp_set_num_threads** routine sets the value of the first element of this list, and the **omp_get_max_threads** routine retrieves the value of the first element of this list.
- Detailed values in the *place-partition-var* ICV are retrieved using the listed routines.
- The **thread_limit** clause sets the *thread-limit-var* ICV for the region of the construct on which it appears.

Cross References

- **omp_get_active_level** Routine, see Section 21.17
- **omp_get_affinity_format** Routine, see Section 29.9
- **omp_get_cancellation** Routine, see Section 30.1

- **omp_get_default_allocator** Routine, see Section 27.10
- **omp_get_default_device** Routine, see Section 24.2
- **omp_get_device_num** Routine, see Section 24.4
- **omp_get_device_num_teams** Routine, see Section 24.11
- **omp_get_device_teams_thread_limit** Routine, see Section 24.13
- **omp_get_dynamic** Routine, see Section 21.8
- **omp_get_level** Routine, see Section 21.14
- **omp_get_max_active_levels** Routine, see Section 21.13
- **omp_get_max_task_priority** Routine, see Section 23.1.1
- **omp_get_max_teams** Routine, see Section 22.4
- **omp_get_max_threads** Routine, see Section 21.4
- **omp_get_num_devices** Routine, see Section 24.3
- **omp_get_num_procs** Routine, see Section 24.5
- **omp_get_num_teams** Routine, see Section 22.1
- **omp_get_num_threads** Routine, see Section 21.2
- **omp_get_partition_num_places** Routine, see Section 29.6
- **omp_get_partition_place_nums** Routine, see Section 29.7
- **omp_get_place_num_procs** Routine, see Section 29.3
- **omp_get_place_proc_ids** Routine, see Section 29.4
- **omp_get_proc_bind** Routine, see Section 29.1
- **omp_get_schedule** Routine, see Section 21.10
- **omp_get_supported_active_levels** Routine, see Section 21.11
- **omp_get_team_num** Routine, see Section 22.3
- **omp_get_teams_thread_limit** Routine, see Section 22.5
- **omp_get_thread_limit** Routine, see Section 21.5
- **omp_get_thread_num** Routine, see Section 21.3
- **omp_in_explicit_task** Routine, see Section 23.1.2
- **omp_in_final** Routine, see Section 23.1.3
- **omp_set_affinity_format** Routine, see Section 29.8

- **omp_set_default_allocator** Routine, see Section 27.9
- **omp_set_default_device** Routine, see Section 24.1
- **omp_set_device_num_teams** Routine, see Section 24.12
- **omp_set_device_teams_thread_limit** Routine, see Section 24.14
- **omp_set_dynamic** Routine, see Section 21.7
- **omp_set_max_active_levels** Routine, see Section 21.12
- **omp_set_num_teams** Routine, see Section 22.2
- **omp_set_num_threads** Routine, see Section 21.1
- **omp_set_schedule** Routine, see Section 21.9
- **omp_set_teams_thread_limit** Routine, see Section 22.6
- **thread_limit** Clause, see Section 15.3

3.4 How the Per-Data Environment ICVs Work

When a task-generating construct, a **parallel** construct or a **teams** construct is encountered, each generated task inherits the values of the ICVs with data environment ICV scope from the ICV values of the generating task, unless otherwise specified.

When a **parallel** construct is encountered, the value of each ICV with implicit task ICV scope is inherited from the binding implicit task of the generating task unless otherwise specified.

When a task-generating construct is encountered, each generated task inherits the value of *nthreads-var* from the *nthreads-var* value of the generating task. If a **parallel** construct is encountered on which a **num_threads** clause is specified with a *nthreads* list of more than one list item, the value of *nthreads-var* for the generated implicit tasks is the list obtained by deletion of the first item of the *nthreads* list. Otherwise, when a **parallel** construct is encountered, if the *nthreads-var* list of the generating task contains a single element, the generated implicit tasks inherit that list as the value of *nthreads-var*; if the *nthreads-var* list of the generating task contains multiple elements, the generated implicit tasks inherit the value of *nthreads-var* as the list obtained by deletion of the first element from the *nthreads-var* value of the generating task. The *bind-var* ICV is handled in the same way as the *nthreads-var* ICV, except that an override list cannot be specified through the **proc_bind** clause of an encountered **parallel** construct.

When a **target** construct corresponds to an active target region, the resulting initial task uses the values of the data environment scoped ICVs from the device data environment ICV values of the device that will execute the region, unless otherwise specified.

When a **target** construct corresponds to an inactive target region, the resulting initial task uses the values of the ICVs with data environment ICV scope from the data environment of the task that encountered the **target** construct, unless otherwise specified.

If a **target** construct with a **thread_limit** clause is encountered, the *thread-limit-var* ICV from the data environment of the resulting initial task is instead set to an implementation defined value between one and the value specified in the clause.

If a **target** construct with no **thread_limit** clause is encountered, the *thread-limit-var* ICV from the data environment of the resulting initial task is set to an implementation defined value that is greater than zero.

If a **teams** construct with a **thread_limit** clause is encountered, the *thread-limit-var* ICV from the data environment of the initial task for each team is instead set to an implementation defined value between one and the value specified in the clause.

If a **teams** construct with no **thread_limit** clause is encountered and *teams-thread-limit-var* is greater than zero, the *thread-limit-var* ICV from the data environment of the initial task of each team is set to an implementation defined value that is greater than zero and does not exceed *teams-thread-limit-var*. If a **teams** construct with no **thread_limit** clause is encountered and *teams-thread-limit-var* is zero, the *thread-limit-var* ICV from the data environment of the initial task of each team is set to an implementation defined value that is greater than zero.

If a **target** construct, **teams** construct, or **parallel** construct is encountered, the *team-generator-var* ICV for the data environments of the generated implicit tasks is instead set to the value of the appropriate team generator type as specified in Section 39.13.

When encountering a worksharing-loop region for which the **runtime** schedule type is specified, all implicit task regions that constitute the binding parallel region must have the same value for *run-sched-var* in their data environments. Otherwise, the behavior is unspecified.

Cross References

- OMPD **team_generator** Type, see Section 39.13

3.5 ICV Override Relationships

Section 3.5 shows the override relationships among construct clauses and ICVs. The table only lists ICVs that can be overridden by a clause.

TABLE 3.4: ICV Override Relationships

ICV	Clause, if used
bind-var	**proc_bind**
def-allocator-var	**allocate**, **allocator**
nteams-var	**num_teams**
nthreads-var	**num_threads**

ICV	Clause, if used
run-sched-var	**`schedule`**
teams-thread-limit-var	**`thread_limit`**

If a **`schedule`** clause specifies a modifier then that modifier overrides any modifier that is specified in the *run-sched-var* ICV.

If *bind-var* is not set to *false* then the **`proc_bind`** clause overrides the value of the first element of the *bind-var* ICV; otherwise, the **`proc_bind`** clause has no effect.

Cross References

- **`allocate`** Clause, see Section 8.6
- **`allocator`** Clause, see Section 8.4
- **`num_teams`** Clause, see Section 12.2.1
- **`num_threads`** Clause, see Section 12.1.2
- **`proc_bind`** Clause, see Section 12.1.4
- **`schedule`** Clause, see Section 13.6.3
- **`thread_limit`** Clause, see Section 15.3

4 Environment Variables

This chapter describes the OpenMP environment variables that specify the settings of the ICVs that affect the execution of OpenMP programs (see Chapter 3). The names of the environment variables must be upper case. Unless otherwise specified, the values assigned to the environment variables are case insensitive and may have leading and trailing white space. The assigned values for most environment variables are strings or integers. In particular, boolean values are specified as the string **true** or **false**. Modifications to the environment variables after the program has started, even if modified by the program itself, are ignored by the OpenMP implementation. However, the settings of some of the ICVs can be modified during the execution of the OpenMP program by the use of the appropriate directive clauses or OpenMP API routines. These examples demonstrate how to set the OpenMP environment variables in different environments:

- csh-like shells:

```
setenv OMP_SCHEDULE "dynamic"
```

- bash-like shells:

```
export OMP_SCHEDULE="dynamic"
```

- Windows Command Line:

```
set OMP_SCHEDULE=dynamic
```

As defined in Section 3.2, device-specific environment variables extend many of the environment variables defined in this chapter. If the corresponding environment variable for a specific device number is set, then the setting for that environment variable is used to set the value of the associated ICV of the device with the corresponding device number. If the corresponding environment variable that includes the **_DEV** suffix but no device number is set, then its setting is used to set the value of the associated ICV of any non-host device for which the device number-specific corresponding environment variable is not set. The corresponding environment variable without a suffix sets the associated ICV of the host device. If the corresponding environment variable includes the **_ALL** suffix, the setting of that environment variable is used to set the value of the associated ICV of any host or non-host device for which corresponding environment variables that are device number specific through the use of the **_DEV** suffix or the absence of a suffix are not set.

Restrictions

Restrictions to device-specific environment variables are as follows:

- Device-specific environment variables must not correspond to environment variables that initialize ICVs with global ICV scope.
- Device-specific environment variables must not specify the host device.

4.1 Parallel Region Environment Variables

This section defines environment variables that affect the operation of **parallel** regions.

4.1.1 Abstract Name Values

This section defines abstract names that must be understood by the execution and runtime environment for the environment variables that explicitly allow them. The entities defined by the abstract names are implementation defined. There are two kinds of abstract names: conceptual abstract names and numeric abstract names.

Conceptual abstract names include place-list abstract names that are the strings defined in Table 4.1. If an environment variable is set to a value that includes a place-list abstract name, the behavior is as if the place-list abstract name were replaced with the list of places associated with that abstract name on each device where the environment variable is applied.

TABLE 4.1: Predefined Place-list Abstract Names

Abstract Name	Meaning
threads	A set where each place corresponds to a single hardware thread of the device.
cores	A set where each place corresponds to a single core of the device.
ll_caches	A set where each place corresponds to the set of cores for a single last-level cache of the device.
numa_domains	A set where each place corresponds to the set of cores for a single NUMA domain of the device.
sockets	A set where each place corresponds to the set of cores for a single socket of the device.

For each place-list abstract name specified in Table 4.1, a corresponding place-count abstract name prefixed with **n_** also exists for which the associated value is the number of places in the list of places specified by the place-list abstract name, as described above.

If an environment variable is set to a value that includes a numeric abstract name, the behavior is as if the numeric abstract name were replaced with the value associated with that numeric abstract name.

4.1.2 OMP_DYNAMIC

The **OMP_DYNAMIC** environment variable controls dynamic adjustment of the number of threads to use for executing **parallel** regions by setting the initial value of the *dyn-var* ICV.

The value of this environment variable must be one of the following:

true | **false**

If the environment variable is set to **true**, the OpenMP implementation may adjust the number of threads to use for executing **parallel** regions in order to optimize the use of system resources. If the environment variable is set to **false**, the dynamic adjustment of the number of threads is disabled. The behavior of the program is implementation defined if the value of **OMP_DYNAMIC** is neither **true** nor **false**.

Example:

```
export OMP_DYNAMIC=true
```

Cross References

- *dyn-var* ICV, see Table 3.1
- **omp_get_dynamic** Routine, see Section 21.8
- **omp_set_dynamic** Routine, see Section 21.7
- **parallel** Construct, see Section 12.1

4.1.3 OMP_NUM_THREADS

The **OMP_NUM_THREADS** environment variable sets the number of threads to use for **parallel** regions by setting the initial value of the *nthreads-var* ICV. See Chapter 3 for a comprehensive set of rules about the interaction between the **OMP_NUM_THREADS** environment variable, the **num_threads** clause, the **omp_set_num_threads** routine and dynamic adjustment of threads, and Section 12.1.1 for a complete algorithm that describes how the number of threads for a **parallel** region is determined.

The value of this environment variable must be a list of positive integer values and/or numeric abstract names. The values of the list set the number of threads to use for **parallel** regions at the corresponding nested levels.

The behavior of the program is implementation defined if any value of the list specified in the **OMP_NUM_THREADS** environment variable leads to a number of threads that is greater than an implementation can support or if any value is not a positive integer.

The **OMP_NUM_THREADS** environment variable sets the *max-active-levels-var* ICV to the number of active levels of parallelism that the implementation supports if the **OMP_NUM_THREADS** environment variable is set to a comma-separated list of more than one value. The value of the *max-active-levels-var* ICV may be overridden by setting **OMP_MAX_ACTIVE_LEVELS**. See Section 4.1.5 for details.

Example:

```
export OMP_NUM_THREADS=4,3,2
export OMP_NUM_THREADS=n_cores,2
```

Cross References

- **OMP_MAX_ACTIVE_LEVELS**, see Section 4.1.5
- *nthreads-var* ICV, see Table 3.1
- **num_threads** Clause, see Section 12.1.2
- **omp_set_num_threads** Routine, see Section 21.1
- **parallel** Construct, see Section 12.1

4.1.4 OMP_THREAD_LIMIT

The **OMP_THREAD_LIMIT** environment variable sets the number of threads to use for a contention group by setting the *thread-limit-var* ICV. The value of this environment variable must be a positive integer or a numeric abstract name. The behavior of the program is implementation defined if the requested value of **OMP_THREAD_LIMIT** is greater than the number of threads that an implementation can support, or if the value is not a positive integer.

Cross References

- *thread-limit-var* ICV, see Table 3.1

4.1.5 OMP_MAX_ACTIVE_LEVELS

The **OMP_MAX_ACTIVE_LEVELS** environment variable controls the maximum number of nested active **parallel** regions by setting the initial value of the *max-active-levels-var* ICV. The value of this environment variable must be a non-negative integer. The behavior of the program is implementation defined if the requested value of **OMP_MAX_ACTIVE_LEVELS** is greater than the maximum number of active levels an implementation can support, or if the value is not a non-negative integer.

Cross References

- *max-active-levels-var* ICV, see Table 3.1

4.1.6 OMP_PLACES

The **OMP_PLACES** environment variable sets the initial value of the *place-partition-var* ICV. A list of places can be specified in the **OMP_PLACES** environment variable. The value of **OMP_PLACES** can be one of two types of values: either a place-list abstract name that describes a set of places or an explicit list of places described by non-negative numbers.

The **`OMP_PLACES`** environment variable can be defined using an explicit ordered list of comma-separated places. A place is defined by an unordered set of comma-separated non-negative numbers enclosed by braces, or a non-negative number. The meaning of the numbers and how the numbering is done are implementation defined. Generally, the numbers represent the smallest unit of execution exposed by the execution environment, typically a hardware thread.

Intervals may also be used to define places. Intervals can be specified using the <*lower-bound*> : <*length*> : <*stride*> notation to represent the following list of numbers: "<*lower-bound*>, <*lower-bound*> + <*stride*>, ..., <*lower-bound*> + (<*length*> - 1)*<*stride*>." When <*stride*> is omitted, a unit stride is assumed. Intervals can specify numbers within a place as well as sequences of places.

An exclusion operator "**`!`**" can also be used to exclude the number or place immediately following the operator.

Alternatively, the place-list abstract names listed in Table 4.1 should be understood by the execution and runtime environment. The entities defined by the abstract names are implementation defined. An implementation may also add abstract names as appropriate for the target platform.

The abstract name may be appended with one or two positive numbers in parentheses, that is, *abstract_name* **`(`**<*len* >**`)`** or *abstract_name* **`(`**<*len* > **`:`** <*stride* >**`)`** where *abstract_name* is a place-list abstract name listed in Table 4.1, *len* denotes the length of the place list and *stride* denotes the increment between consecutive places in the place list. When requesting fewer places than available on the system, the determination of which resources of type *abstract_name* are to be included in the place list is implementation defined. When requesting more resources than available, the length of the place list is implementation defined.

The behavior of the program is implementation defined when the execution environment cannot map a numerical value (either explicitly defined or implicitly derived from an interval) within the **`OMP_PLACES`** list to a processor on the target platform, or if it maps to an unavailable processor. The behavior is also implementation defined when the **`OMP_PLACES`** environment variable is defined using a place-list abstract name.

The following grammar describes the values accepted for the **`OMP_PLACES`** environment variable.

$$
\begin{aligned}
\langle\text{list}\rangle &\models \langle\text{p-list}\rangle \mid \langle\text{aname}\rangle \\
\langle\text{p-list}\rangle &\models \langle\text{p-interval}\rangle \mid \langle\text{p-list}\rangle,\langle\text{p-interval}\rangle \\
\langle\text{p-interval}\rangle &\models \langle\text{place}\rangle:\langle\text{len}\rangle:\langle\text{stride}\rangle \mid \langle\text{place}\rangle:\langle\text{len}\rangle \mid \langle\text{place}\rangle \mid !\langle\text{place}\rangle \\
\langle\text{place}\rangle &\models \{\langle\text{res-list}\rangle\} \mid \langle\text{res}\rangle \\
\langle\text{res-list}\rangle &\models \langle\text{res-interval}\rangle \mid \langle\text{res-list}\rangle,\langle\text{res-interval}\rangle \\
\langle\text{res-interval}\rangle &\models \langle\text{res}\rangle:\langle\text{len}\rangle:\langle\text{stride}\rangle \mid \langle\text{res}\rangle:\langle\text{len}\rangle \mid \langle\text{res}\rangle \mid !\langle\text{res}\rangle \\
\langle\text{aname}\rangle &\models \langle\text{word}\rangle(\langle\text{len}\rangle:\langle\text{stride}\rangle) \mid \langle\text{word}\rangle(\langle\text{len}\rangle) \mid \langle\text{word}\rangle
\end{aligned}
$$

⟨word⟩ ⊨ sockets | cores | ll_caches | numa_domains
| threads | <implementation-defined abstract name>
⟨res⟩ ⊨ *non-negative integer*
⟨len⟩ ⊨ *positive integer*
⟨stride⟩ ⊨ *integer*

Examples:

```
export OMP_PLACES=threads
export OMP_PLACES="threads(4)"
export OMP_PLACES="threads(8:2)"
export OMP_PLACES
   ="{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"
export OMP_PLACES="{0:4},{4:4},{8:4},{12:4}"
export OMP_PLACES="{0:4}:4:4"
```

where each of the last three definitions corresponds to the same four places including the smallest units of execution exposed by the execution environment numbered, in turn, 0 to 3, 4 to 7, 8 to 11, and 12 to 15.

Cross References

- *place-partition-var* ICV, see Table 3.1

4.1.7 OMP_PROC_BIND

The **OMP_PROC_BIND** environment variable sets the initial value of the *bind-var* ICV. The value of this environment variable is either **true**, **false**, or a comma separated list of **primary**, **close**, or **spread**. The values of the list set the thread affinity policy to be used for parallel regions at the corresponding nested level. The first value also sets the thread affinity policy to be used for implicit parallel regions.

If the environment variable is set to **false**, the execution environment may move OpenMP threads between OpenMP places, thread affinity is disabled, and **proc_bind** clauses on **parallel** constructs are ignored.

Otherwise, the execution environment should not move team-worker threads between places, thread affinity is enabled, and the initial thread is bound to the first place in the *place-partition-var* ICV prior to the first active parallel region, or immediately after encountering the first task-generating construct. An initial thread that is created by a **teams** construct is bound to the first place in its *place-partition-var* ICV before it begins execution of the associated structured block. A free-agent thread that executes a task bound to a team is assigned a place according to the rules described in Section 12.1.3.

If the environment variable is set to **true**, the thread affinity policy is implementation defined but must conform to the previous paragraph. The behavior of the program is implementation defined if the value in the **OMP_PROC_BIND** environment variable is not **true**, **false**, or a comma separated list of **primary**, **close**, or **spread**. The behavior is also implementation defined if an initial thread cannot be bound to the first place in the *place-partition-var* ICV.

The **OMP_PROC_BIND** environment variable sets the *max-active-levels-var* ICV to the number of active levels of parallelism that the implementation supports if the **OMP_PROC_BIND** environment variable is set to a comma-separated list of more than one element. The value of the *max-active-levels-var* ICV may be overridden by setting **OMP_MAX_ACTIVE_LEVELS**. See Section 4.1.5 for details.

Examples:

```
export OMP_PROC_BIND=false
export OMP_PROC_BIND="spread, spread, close"
```

Cross References

- **OMP_MAX_ACTIVE_LEVELS**, see Section 4.1.5
- Controlling OpenMP Thread Affinity, see Section 12.1.3
- *bind-var* ICV, see Table 3.1
- *max-active-levels-var* ICV, see Table 3.1
- *place-partition-var* ICV, see Table 3.1
- **omp_get_proc_bind** Routine, see Section 29.1
- **parallel** Construct, see Section 12.1
- **proc_bind** Clause, see Section 12.1.4
- **teams** Construct, see Section 12.2

4.2 Teams Environment Variables

This section defines environment variables that affect the operation of **teams** regions.

4.2.1 OMP_NUM_TEAMS

The **OMP_NUM_TEAMS** environment variable sets the maximum number of teams created by a **teams** construct by setting the *nteams-var* ICV. The value of this environment variable must be a non-negative integer. The behavior of the program is implementation defined if the requested value of **OMP_NUM_TEAMS** is greater than the number of teams that an implementation can support, or if the value is not a positive integer.

Cross References

- *nteams-var* ICV, see Table 3.1
- **teams** Construct, see Section 12.2

4.2.2 OMP_TEAMS_THREAD_LIMIT

The **OMP_TEAMS_THREAD_LIMIT** environment variable sets the maximum number of OpenMP threads that can execute tasks in each contention group created by a **teams** construct by setting the *teams-thread-limit-var* ICV. The value of this environment variable must be a positive integer or a numeric abstract name. The behavior of the program is implementation defined if the requested value of **OMP_TEAMS_THREAD_LIMIT** is greater than the number of threads that an implementation can support, or if the value is neither a positive integer nor one of the allowed abstract names.

Cross References

- *teams-thread-limit-var* ICV, see Table 3.1
- **teams** Construct, see Section 12.2

4.3 Program Execution Environment Variables

This section defines environment variables that affect program execution.

4.3.1 OMP_SCHEDULE

The **OMP_SCHEDULE** environment variable controls the schedule type and chunk size of all worksharing-loop constructs that have the schedule type **runtime**, by setting the value of the *run-sched-var* ICV. The value of this environment variable takes the form *[modifier:]kind[, chunk]*, where:

- *modifier* is one of **monotonic** or **nonmonotonic**;
- *kind* specifies the schedule type and is one of **static**, **dynamic**, **guided**, or **auto**;
- *chunk* is an optional positive integer that specifies the chunk size.

If the *modifier* is not present, the *modifier* is set to **monotonic** if *kind* is **static**; for any other *kind* it is set to **nonmonotonic**.

If *chunk* is present, white space may be on either side of the "**,**".

The behavior of the program is implementation defined if the value of **OMP_SCHEDULE** does not conform to the above format.

Examples:

```
export OMP_SCHEDULE="guided,4"
export OMP_SCHEDULE="dynamic"
export OMP_SCHEDULE="nonmonotonic:dynamic,4"
```

Cross References

- *run-sched-var* ICV, see Table 3.1
- **schedule** Clause, see Section 13.6.3

4.3.2 OMP_STACKSIZE

The **OMP_STACKSIZE** environment variable controls the size of the stack for threads, by setting the value of the *stacksize-var* ICV. The environment variable does not control the size of the stack for an initial thread. Whether this environment variable also controls the size of the stack of native threads is implementation defined. The value of this environment variable takes the form *size[unit]*, where:

- *size* is a positive integer that specifies the size of the stack for threads.
- *unit* is **B**, **K**, **M**, or **G** and specifies whether the given size is in Bytes, Kilobytes (1024 Bytes), Megabytes (1024 Kilobytes), or Gigabytes (1024 Megabytes), respectively. If *unit* is present, white space may occur between *size* and it, whereas if *unit* is not present then **K** is assumed.

The behavior of the program is implementation defined if **OMP_STACKSIZE** does not conform to the above format, or if the implementation cannot provide a stack with the requested size.

Examples:

```
export OMP_STACKSIZE=2000500B
export OMP_STACKSIZE="3000 k "
export OMP_STACKSIZE=10M
export OMP_STACKSIZE=" 10 M "
export OMP_STACKSIZE="20 m "
export OMP_STACKSIZE=" 1G"
export OMP_STACKSIZE=20000
```

Cross References

- *stacksize-var* ICV, see Table 3.1

4.3.3 OMP_WAIT_POLICY

The **OMP_WAIT_POLICY** environment variable provides a hint to an OpenMP implementation about the desired behavior of waiting native threads by setting the *wait-policy-var* ICV. A compliant implementation may or may not abide by the setting of the environment variable. The value of this environment variable must be one of the following:

active | **passive**

The **active** value specifies that waiting native threads should mostly be active, consuming processor cycles, while waiting. A compliant implementation may, for example, make waiting native threads spin. The **passive** value specifies that waiting native threads should mostly be passive, not consuming processor cycles, while waiting. For example, a compliant implementation may make waiting native threads yield the processor to other native threads or go to sleep. The details of the **active** and **passive** behaviors are implementation defined. The behavior of the program is implementation defined if the value of **OMP_WAIT_POLICY** is neither **active** nor **passive**.

Examples:

```
export OMP_WAIT_POLICY=ACTIVE
export OMP_WAIT_POLICY=active
export OMP_WAIT_POLICY=PASSIVE
export OMP_WAIT_POLICY=passive
```

Cross References

- *wait-policy-var* ICV, see Table 3.1

4.3.4 OMP_DISPLAY_AFFINITY

The **OMP_DISPLAY_AFFINITY** environment variable sets the *display-affinity-var* ICV so that the runtime displays formatted affinity information for the host device. Affinity information is printed for all OpenMP threads in each parallel region upon first entering it. Also, if the information accessible by the format specifiers listed in Table 4.2 changes for any thread in the parallel region then thread affinity information for all threads in that region is again displayed. If the thread affinity for each respective parallel region at each nesting level has already been displayed and the thread affinity has not changed, then the information is not displayed again. Thread affinity information for threads in the same parallel region may be displayed in any order. The value of the **OMP_DISPLAY_AFFINITY** environment variable may be set to one of these values:

true | **false**

The **true** value instructs the runtime to display the thread affinity information, and uses the format setting defined in the *affinity-format-var* ICV. The runtime does not display the thread affinity information when the value of the **OMP_DISPLAY_AFFINITY** environment variable is **false** or undefined. For all values of the environment variable other than **true** or **false**, the display action is implementation defined.

Example:

```
export OMP_DISPLAY_AFFINITY=TRUE
```

For this example, an OpenMP implementation displays thread affinity information during program execution, in a format given by the *affinity-format-var* ICV. The following is a sample output:

```
nesting_level=  1,   thread_num=   0,   thread_affinity=   0,1
nesting_level=  1,   thread_num=   1,   thread_affinity=   2,3
```

Cross References

- **OMP_AFFINITY_FORMAT**, see Section 4.3.5
- Controlling OpenMP Thread Affinity, see Section 12.1.3
- *affinity-format-var* ICV, see Table 3.1
- *display-affinity-var* ICV, see Table 3.1

4.3.5 OMP_AFFINITY_FORMAT

The **OMP_AFFINITY_FORMAT** environment variable sets the initial value of the *affinity-format-var* ICV which defines the format when displaying thread affinity information. The value of this environment variable is case sensitive and leading and trailing white space is significant. Its value is a character string that may contain as substrings one or more field specifiers (as well as other characters). The format of each field specifier is

```
%[[[0].] size ] type
```

where each specifier must contain the percent symbol (`%`) and a type, that must be either a single character short name or its corresponding long name delimited with curly braces, such as **%n** or **%{thread_num}**. A literal percent is specified as `%%`. Field specifiers can be provided in any order. The behavior is implementation defined for field specifiers that do not conform to this format.

The `0` modifier indicates whether or not to add leading zeros to the output, following any indication of sign or base. The `.` modifier indicates the output should be right justified when *size* is specified. By default, output is left justified. The minimum field length is *size*, which is a decimal digit string with a non-zero first digit. If no *size* is specified, the actual length needed to print the field will be used. If the `0` modifier is used with *type* of **A**, **{thread_affinity}**, **H**, **{host}**, or a type that is not printed as a number, the result is unspecified. Any other characters in the format string that are not part of a field specifier will be included literally in the output.

TABLE 4.2: Available Field Types for Formatting OpenMP Thread Affinity Information

Short Name	Long Name	Meaning
t	**team_num**	The value returned by **omp_get_team_num**
T	**num_teams**	The value returned by **omp_get_num_teams**

table continued on next page

table continued from previous page

Short Name	Long Name	Meaning
L	**nesting_level**	The value returned by **omp_get_level**
n	**thread_num**	The value returned by **omp_get_thread_num**
N	**num_threads**	The value returned by **omp_get_num_threads**
a	**ancestor_tnum**	The value returned by **omp_get_ancestor_thread_num** with an argument of one less than the value returned by **omp_get_level**
H	**host**	The name for the host device on which the OpenMP program is running
P	**process_id**	The process identifier used by the implementation
i	**native_thread_id**	The native thread identifier used by the implementation
A	**thread_affinity**	The list of numerical identifiers, in the format of a comma-separated list of integers or integer ranges, that represent processors on which a thread may execute, subject to OpenMP thread affinity control and/or other external affinity mechanisms

Implementations may define additional field types. If an implementation does not have information for a field type or an unknown field type is part of a field specifier, "undefined" is printed for this field when displaying thread affinity information.

Example:

```
export OMP_AFFINITY_FORMAT=\
"Thread Affinity: %0.3L %.8n %.15{thread_affinity} %.12H"
```

The above example causes an OpenMP implementation to display thread affinity information in the following form:

```
Thread Affinity: 001        0       0-1,16-17       nid003
Thread Affinity: 001        1       2-3,18-19       nid003
```

Cross References

- Controlling OpenMP Thread Affinity, see Section 12.1.3
- *affinity-format-var* ICV, see Table 3.1
- **omp_get_ancestor_thread_num** Routine, see Section 21.15
- **omp_get_level** Routine, see Section 21.14

- **omp_get_num_teams** Routine, see Section 22.1
- **omp_get_num_threads** Routine, see Section 21.2
- **omp_get_thread_num** Routine, see Section 21.3

4.3.6 OMP_CANCELLATION

The **OMP_CANCELLATION** environment variable sets the initial value of the *cancel-var* ICV. The value of this environment variable must be one of the following:

true | **false**

If the environment variable is set to **true**, the effects of the **cancel** construct and of cancellation points are enabled (i.e., cancellation is enabled). If the environment variable is set to **false**, cancellation is disabled and **cancel** constructs and cancellation points are effectively ignored. The behavior of the program is implementation defined if **OMP_CANCELLATION** is set to neither **true** nor **false**.

Cross References

- **cancel** Construct, see Section 18.2
- *cancel-var* ICV, see Table 3.1

4.3.7 OMP_AVAILABLE_DEVICES

The **OMP_AVAILABLE_DEVICES** environment variable sets the *available-devices-var* ICV and determines the available non-host devices and their device numbers by permitting selection of devices from the set of supported accessible devices and by ordering them. This ICV is initialized before any other ICV that uses a device number, depends on the number of available devices, or permits device-specific environment variables. After the *available-devices-var* ICV is initialized, only those devices that the ICV identifies are available devices and the **omp_get_num_devices** routine returns the number of devices stored in the ICV.

The value of this environment variable must be a comma-separated list. Each item is either a trait specification as specified in the following or *. A * expands to all non-host accessible devices that are supported devices while a trait specification expands to a possibly empty set of accessible and supported devices for which the specification is fulfilled. After expansion, further selection via an optional array subscript syntax and removal of devices that appear in previous items, each item contains an unordered set of devices. A consecutive unique device number is then assigned to each device in the sets, starting with device number zero, where the device number of the first device in an item is the total number of devices in all previous items.

Traits are specified by the case-insensitive trait name followed by the argument in parentheses. The permitted traits are **kind(***kind-name***)**, **isa(***isa-name***)**, **arch(***arch-name***)**, **vendor(***vendor-name***)**, and **uid(***uid-string***)**, where the names are as specified in Section 9.1

and the OpenMP Additional Definitions document; the *kind-name* **host** is not permitted. Multiple traits can be combined using the binary operators **&&** and **||** to require both or either trait, respectively. Parentheses can be used for grouping, but are optional except that **&&** and **||** may not appear in the same grouping level. The unary **!** operator inverts the meaning of the immediately following trait or parenthesized group.

Each trait specification or * yields a (possibly zero-sized) array of non-host devices with the lowest array element, if it exists, having index zero. The C/C++ syntax **[***index***]** can be used to select an element and the array section syntax for C/C++ as specified in Section 5.2.5 can be used to specify a subset of elements. Any array element specified by the subscript that is outside the bounds of the array resulting from the trait specification or * is silently excluded.

Example:

Four GPUs are accessible and supported, with unique identifiers represented as <uid-gpu0>,...,<uid-gpu3>.

```
export OMP_AVAILABLE_DEVICES="kind(gpu)"
```

```
export OMP_AVAILABLE_DEVICES="uid(<uid-gpu0>),kind(gpu)"
```

```
export OMP_AVAILABLE_DEVICES="uid(<uid-gpu1>),kind(gpu)[:2]"
```

where the above **OMP_AVAILABLE_DEVICES** assignments select:

- All GPUs;
- All GPUs with device <uid-gpu0> assigned device number 0; and
- Device <uid-gpu1>, which is assigned device number 0, and two other GPUs.

Cross References

- Device Directives and Clauses, see Chapter 15
- *available-devices-var* ICV, see Table 3.1

4.3.8 OMP_DEFAULT_DEVICE

The **OMP_DEFAULT_DEVICE** environment variable sets the initial value of the *default-device-var* ICV. The value of this environment variable must be a comma-separated list, each item being either a non-negative integer value that denotes the device number, a trait specification with an optional subscript selector, or one of the following case-insensitive string literals: **initial** to specify the host device, **invalid** to specify the device number **omp_invalid_device**, or **default** to set the ICV as if this environment variable was not specified (see Section 1.2).

The trait specification is as described for **OMP_AVAILABLE_DEVICES** (see Section 4.3.7), except that in addition the trait **device_num(***device number***)** may be specified and **host** is permitted as *kind-name*. The device numbers yielded by the trait specification are sorted in ascending order by device number and form a set; the array-element syntax as described for

OMP_AVAILABLE_DEVICES can be used to select an element from this set. If an item is an empty set, non-existing element, or does not evaluate to an available device, the next item is evaluated; otherwise, the *default-device-var* ICV is set to the first value of the set. However, **initial**, **invalid**, and **default** always match. If none of the list items match, the *default-device-var* ICV is set to **omp_invalid_device**.

Example:

Four GPUs are accessible and supported, with unique identifiers represented as <uid-gpu0>,...,<uid-gpu3>. The default device is set to device <uid-gpu0>.

```
export OMP_DEFAULT_DEVICE="uid(<uid-gpu0>)"
```

Cross References

- Device Directives and Clauses, see Chapter 15
- *default-device-var* ICV, see Table 3.1

4.3.9 OMP_TARGET_OFFLOAD

The **OMP_TARGET_OFFLOAD** environment variable sets the initial value of the *target-offload-var* ICV. Its value must be one of the following:

mandatory | **disabled** | **default**

The **mandatory** value specifies that the effect of any device construct or device routine that uses a device that is not an available device or a supported device, or uses a non-conforming device number, is as if the **omp_invalid_device** device number was used. Support for the **disabled** value is implementation defined. If an implementation supports it, the behavior is as if the only device is the host device. The **default** value specifies the default behavior as described in Section 1.2.

Example:

```
export OMP_TARGET_OFFLOAD=mandatory
```

Cross References

- Device Directives and Clauses, see Chapter 15
- Device Memory Routines, see Chapter 25
- *target-offload-var* ICV, see Table 3.1

4.3.10 OMP_THREADS_RESERVE

The **OMP_THREADS_RESERVE** environment variable controls the number of reserved threads in each contention group by setting the initial value of the *structured-thread-limit-var* and the *free-agent-thread-limit-var* ICVs.

The **OMP_THREADS_RESERVE** environment variable can be defined using a non-negative integer or an unordered list of reservations. Each reservation specifies a thread-reservation type, for which the possible values are listed in Table 4.3. The reservation type may be appended with one non-negative number in parentheses, that is, *reservation_type* **(**<*num-threads*>**)**, where <*num-threads*> denotes the number of threads to reserve for that reservation type. If only a non-negative integer is provided, this number denotes the number of threads to reserve for structured parallelism. If only one reservation type is provided, and its <*num-threads*> is not specified, the number of threads to reserve is *thread-limit-var* if the reservation type is **structured**, or *thread-limit-var* minus 1 if the reservation type is **free_agent**.

TABLE 4.3: Reservation Types for **OMP_THREADS_RESERVE**

Reservation Type	Meaning	Default Value
structured	Threads reserved for structured threads	1
free_agent	Threads reserved for free-agent threads	0

The **OMP_THREADS_RESERVE** environment variable sets the initial value of the *structured-thread-limit-var* and the *free-agent-thread-limit-var* ICVs according to Algorithm 4.1.

Algorithm 4.1 Initial *structured-thread-limit-var* and *free-agent-thread-limit-var* ICVs Values

let *structured-reserve* be the number of threads to reserve for structured threads;

let *free-agent-reserve* be the number of threads to reserve for free-agent threads;

let *threads-reserve* be the sum of *structured-reserve* and *free-agent-reserve*;

if (*structured-reserve* $<$ 1) **then** *structured-reserve* = 1;

if (*free-agent-reserve* = *thread-limit-var*) **then** *free-agent-reserve* = *free-agent-reserve* - 1;

if (*threads-reserve* $\leq$ *thread-limit-var*) **then**

  structured-thread-limit-var = *thread-limit-var* - *free-agent-reserve*;

  free-agent-thread-limit-var = *thread-limit-var* - *structured-reserve*;

else behavior is implementation defined

The following grammar describes the values accepted for the **OMP_THREADS_RESERVE** environment variable.

$$
\begin{aligned}
\langle\text{reserve}\rangle &\models \langle\text{res-list}\rangle \;|\; \langle\text{res-type}\rangle \;|\; \langle\text{res-num}\rangle \\
\langle\text{res-list}\rangle &\models \langle\text{res}\rangle \;|\; \langle\text{res-list}\rangle,\langle\text{res}\rangle \\
\langle\text{res}\rangle &\models \langle\text{res-type}\rangle(\langle\text{res-num}\rangle) \\
\langle\text{res-type}\rangle &\models \text{structured} \;|\; \text{free\_agent} \\
\langle\text{res-num}\rangle &\models \textit{non-negative integer}
\end{aligned}
$$

Examples:

```
export OMP_THREADS_RESERVE=4
export OMP_THREADS_RESERVE="structured(4)"
export OMP_THREADS_RESERVE="structured"
export OMP_THREADS_RESERVE="structured(2),free_agent(2)"
```

where the first two definitions correspond to the same reservation for structured parallelism, the third definition reserves all available threads for structured parallelism, and the last one reserves threads for both structured parallelism and free-agent threads.

Cross References

- *free-agent-thread-limit-var* ICV, see Table 3.1
- *structured-thread-limit-var* ICV, see Table 3.1
- **parallel** Construct, see Section 12.1
- **threadset** Clause, see Section 14.8

4.3.11 OMP_MAX_TASK_PRIORITY

The **OMP_MAX_TASK_PRIORITY** environment variable controls the use of task priorities by setting the initial value of the *max-task-priority-var* ICV. The value of this environment variable must be a non-negative integer.

Example:

```
export OMP_MAX_TASK_PRIORITY=20
```

Cross References

- *max-task-priority-var* ICV, see Table 3.1

4.4 Memory Allocation Environment Variables

This section defines environment variables that affect memory allocations.

4.4.1 OMP_ALLOCATOR

The **OMP_ALLOCATOR** environment variable sets the initial value of the *def-allocator-var* ICV that specifies the default allocator for allocation calls, directives and clauses that do not specify an allocator. The following grammar describes the values accepted for the **OMP_ALLOCATOR** environment variable.

⟨allocator⟩ ⊨ ⟨predef-allocator⟩ | ⟨predef-mem-space⟩ | ⟨predef-mem-space⟩:⟨traits⟩
⟨traits⟩ ⊨ ⟨trait⟩=⟨value⟩ | ⟨trait⟩=⟨value⟩,⟨traits⟩
⟨predef-allocator⟩ ⊨ *one of the predefined allocators from Table 8.3*
⟨predef-mem-space⟩ ⊨ *one of the predefined memory spaces from Table 8.1*
⟨trait⟩ ⊨ *one of the allocator trait names from Table 8.2*
⟨value⟩ ⊨ *one of the allowed values from Table 8.2* | *non-negative integer*
| ⟨predef-allocator⟩

The *value* can be an integer only if the *trait* accepts a numerical value, for the **fb_data** *trait* the *value* can only be *predef-allocator*. If the value of this environment variable is not a predefined allocator then a new allocator with the given predefined memory space and optional traits is created and set as the *def-allocator-var* ICV. If the new allocator cannot be created, the *def-allocator-var* ICV will be set to **omp_default_mem_alloc**.

Example:

```
export OMP_ALLOCATOR=omp_high_bw_mem_alloc
export OMP_ALLOCATOR="omp_large_cap_mem_space:alignment=16,\
pinned=true"
export OMP_ALLOCATOR="omp_high_bw_mem_space:pool_size=1048576,\
fallback=allocator_fb,fb_data=omp_low_lat_mem_alloc"
```

Cross References

- Memory Allocators, see Section 8.2
- *def-allocator-var* ICV, see Table 3.1

4.5 OMPT Environment Variables

This section defines environment variables that affect operation of the OMPT tool interface.

4.5.1 OMP_TOOL

The **OMP_TOOL** environment variable sets the *tool-var* ICV, which controls whether an OpenMP runtime will try to register a first-party tool. The value of this environment variable must be one of the following:

enabled | **disabled**

If **OMP_TOOL** is set to any value other than **enabled** or **disabled**, the behavior is unspecified. If **OMP_TOOL** is not defined, the default value for *tool-var* is **enabled**.

Example:

```
export OMP_TOOL=enabled
```

Cross References

- OMPT Overview, see Chapter 32
- *tool-var* ICV, see Table 3.1

4.5.2 OMP_TOOL_LIBRARIES

The **OMP_TOOL_LIBRARIES** environment variable sets the *tool-libraries-var* ICV to a list of tool libraries that are considered for use on a device on which an OpenMP implementation is being initialized. The value of this environment variable must be a list of names of dynamically-loadable libraries, separated by an implementation specific, platform typical separator. Whether the value of this environment variable is case sensitive is implementation defined.

If the *tool-var* ICV is not **enabled**, the value of *tool-libraries-var* is ignored. Otherwise, if **ompt_start_tool** is not visible in the address space on a device where OpenMP is being initialized or if **ompt_start_tool** returns NULL, an OpenMP implementation will consider libraries in the *tool-libraries-var* list in a left-to-right order. The OpenMP implementation will search the list for a library that meets two criteria: it can be dynamically loaded on the current device and it defines the symbol **ompt_start_tool**. If an OpenMP implementation finds a suitable library, no further libraries in the list will be considered.

Example:

```
export OMP_TOOL_LIBRARIES=libtoolXY64.so:/usr/local/lib/
    libtoolXY32.so
```

Cross References

- OMPT Overview, see Chapter 32
- *tool-libraries-var* ICV, see Table 3.1
- **ompt_start_tool** Procedure, see Section 32.2.1

4.5.3 OMP_TOOL_VERBOSE_INIT

The **OMP_TOOL_VERBOSE_INIT** environment variable sets the *tool-verbose-init-var* ICV, which controls whether an OpenMP implementation will verbosely log the registration of a tool. The value of this environment variable must be one of the following:

disabled | **stdout** | **stderr** | <*filename*>

If **OMP_TOOL_VERBOSE_INIT** is set to any value other than case insensitive **disabled**, **stdout**, or **stderr**, the value is interpreted as a filename and the OpenMP runtime will try to log to a file with prefix *filename*. If the value is interpreted as a filename, whether it is case sensitive is implementation defined. If opening the logfile fails, the output will be redirected to **stderr**. If **OMP_TOOL_VERBOSE_INIT** is not defined, the default value for *tool-verbose-init-var* is **disabled**. Support for logging to **stdout** or **stderr** is implementation defined. Unless *tool-verbose-init-var* is **disabled**, the OpenMP runtime will log the steps of the tool activation process defined in Section 32.2.2 to a file with a name that is constructed using the provided filename prefix. The format and detail of the log is implementation defined. At a minimum, the log will contain one of the following:

- That the *tool-var* ICV is **disabled**;
- An indication that a tool was available in the address space at program launch; or
- The path name of each tool in **OMP_TOOL_LIBRARIES** that is considered for dynamic loading, whether dynamic loading was successful, and whether the **ompt_start_tool** procedure is found in the loaded library.

In addition, if an **ompt_start_tool** procedure is called the log will indicate whether or not the tool will use the OMPT interface.

Example:

```
export OMP_TOOL_VERBOSE_INIT=disabled
export OMP_TOOL_VERBOSE_INIT=STDERR
export OMP_TOOL_VERBOSE_INIT=ompt_load.log
```

Cross References

- OMPT Overview, see Chapter 32
- *tool-verbose-init-var* ICV, see Table 3.1

4.6 OMPD Environment Variables

This section defines environment variables that affect operation of the OMPD tool interface.

4.6.1 OMP_DEBUG

The **OMP_DEBUG** environment variable sets the *debug-var* ICV, which controls whether an OpenMP runtime collects information that an OMPD library may need to support a tool. The value of this environment variable must be one of the following:

enabled | **disabled**

If **OMP_DEBUG** is set to any value other than **enabled** or **disabled** then the behavior is implementation defined.

Example:

```
export OMP_DEBUG=enabled
```

Cross References

- Enabling Runtime Support for OMPD, see Section 38.3.1
- OMPD Overview, see Chapter 38
- *debug-var* ICV, see Table 3.1

4.7 OMP_DISPLAY_ENV

The **OMP_DISPLAY_ENV** environment variable instructs the runtime to display the information as described in the **omp_display_env** routine section (Section 30.4). The value of the **OMP_DISPLAY_ENV** environment variable may be set to one of these values:

true | **false** | **verbose**

If the environment variable is set to **true**, the effect is as if the **omp_display_env** routine is called with the *verbose* argument set to *false* at the beginning of the program. If the environment variable is set to **verbose**, the effect is as if the **omp_display_env** routine is called with the *verbose* argument set to *true* at the beginning of the program. If the environment variable is undefined or set to **false**, the runtime does not display any information. For all values of the environment variable other than **true**, **false**, and **verbose**, the displayed information is unspecified.

Example:

```
export OMP_DISPLAY_ENV=true
```

For the output of the above example, see Section 30.4.

Cross References

- **omp_display_env** Routine, see Section 30.4

5 Directive and Construct Syntax

This chapter describes the syntax of directives and clauses and their association with base language code. Directives are specified with various base language mechanisms that allow compilers to ignore the directives and conditionally compiled code if support of the OpenMP API is not provided or enabled. A compliant implementation must provide an option or interface that ensures that underlying support of all directives and conditional compilation mechanisms is enabled. In the remainder of this document, the phrase *OpenMP compilation* is used to mean a compilation with these OpenMP features enabled.

Restrictions

Restrictions on OpenMP programs include:

- Unless otherwise specified, a program must not depend on any ordering of the evaluations of the expressions that appear in the clauses specified on a directive.
- Unless otherwise specified, a program must not depend on any side effects of the evaluations of the expressions that appear in the clauses specified on a directive.

C / C++

- The use of **`omp`** as the first preprocessing token of a pragma directive must be for OpenMP directives that are defined in this specification; OpenMP reserves these uses for OpenMP directives.
- The use of **`omp`** as the attribute namespace of an attribute specifier, or as the optional namespace qualifier within a **`sequence`** attribute, must be for OpenMP directives that are defined in this specification; OpenMP reserves these uses for such directives.
- The use of **`ompx`** as the first preprocessing token of a pragma directive must be for implementation defined extensions to the OpenMP directives; OpenMP reserves these uses for such extensions.
- The use of **`ompx`** as the attribute namespace of an attribute specifier, or as the optional namespace qualifier within a **`sequence`** attribute, must be for implementation defined extensions to the OpenMP directives; OpenMP reserves these uses for such extensions.

C / C++

Fortran

- In free form source files, the **`!$omp`** sentinel must be used for OpenMP directives that are defined in this specification; OpenMP reserves these uses for such directives.

- In fixed form source files, sentinels that end with **omp** must be used for OpenMP directives that are defined in this specification; OpenMP reserves these uses for such directives.
- In free form source files, the **!$ompx** sentinel must be used for implementation defined extensions to the OpenMP directives; OpenMP reserves these uses for such extensions.
- In fixed form source files, sentinels that end with **omx** must be used for implementation defined extensions to the OpenMP directives; OpenMP reserves these uses for such extensions.

Fortran

- A clause name must be the name of a clause that is defined in this specification except for those that begin with **ompx_**, which may be used for implementation defined extensions and which OpenMP reserves for such extensions.
- OpenMP reserves names that begin with the **omp_**, **ompt_** and **ompd_** prefixes for names defined in this specification so OpenMP programs must not declare names that begin with them.
- OpenMP reserves names that begin with the **ompx_** prefix for implementation defined extensions so OpenMP programs must not declare names that begin with it.

C++

- OpenMP programs must not declare a namespace with the **omp**, **ompx**, **ompt** or **ompd** names, as these are reserved for the OpenMP implementation.

C++

Restrictions on explicit regions (that arise from executable directives) are as follows:

C++

- A **throw** executed inside a region that arises from a thread-limiting construct must cause execution to resume within the same region, and the same thread that threw the exception must catch it. If the directive also has the exception-aborting property then whether the exception is caught or the **throw** results in runtime error termination is implementation defined.

C++

Fortran

- A directive may not appear in a pure or simple procedure unless it has the pure property.
- A directive may not appear in a **WHERE** or **FORALL** construct.
- A directive may not appear in a **DO CONCURRENT** construct unless it has the pure property.
- If more than one image is executing the program, any image control statement, **ERROR STOP** statement, **FAIL IMAGE** statement, **NOTIFY WAIT** statement, collective subroutine call or access to a coindexed object that appears in an explicit region will result in unspecified behavior.

Fortran

5.1 Directive Format

This section defines several categories of directives and constructs. Directives are specified with a directive specification. A directive specification consists of the directive specifier and any clauses that may optionally be associated with the directive, Thus, the *directive-specifation* is:

directive-specifier [[**,** *] clause[[* **,** *] clause] ...]*

where the *directive-specifier* is:

directive-name

or for argument-modified directives:

directive-name[**(***directive-arguments***)** *]*

where *directive-name* is the directive name of the directive.

Some directives specify a paired end directive. If the *directive-name* of such a directive starts with **begin**, the end directive has the same directive name except **begin** is replaced with **end**. If the *directive-name* does not start with **begin**, unless otherwise specified the directive name of the end directive is **end** *directive-name*.

Some directives have underscores in their *directive-name*. Some of those directives are explicitly specified alternatively to allow the underscores in their *directive-name* to be replaced with white space. In addition, if a *directive-name* starts with either **begin** or **end** then it is separated from the rest of the *directive-name* by white space.

The *directive-specification* of a paired end directive may include one or more optional *end-clause*:

directive-specifier [[**,** *] end-clause[[* **,** *] end-clause]...]*

where *end-clause* has the end-clause property, which explicitly allows it on a paired end directive.

C / C++

A directive may be specified as a pragma directive:

#pragma omp *directive-specification new-line*

or a pragma operator:

_Pragma("omp *directive-specification***")**

Note – In this directive, *directive-name* is **depobj**, *directive-arguments* is **o**. *directive-specifier* is **depobj(o)** and *directive-specification* is **depobj(o) depend(inout: d)**.

```
#pragma omp depobj(o) depend(inout: d)
```

White space can be used before and after the **#**. Preprocessing tokens in a *directive-specification* of **#pragma** and **_Pragma** pragmas are subject to macro expansion.

In C23 and later versions or C++11 and later versions, a *directive* may be specified as a C/C++ attribute specifier:

```
[[ omp :: directive-attr ]]
```

C++

or

```
[[ using omp : directive-attr ]]
```

C++

where *directive-attr* is

```
directive( directive-specification )
```

or

```
sequence( [omp::]directive-attr [[, [omp::]directive-attr] ... ] )
```

Multiple attributes on the same statement are allowed. Attribute directives that apply to the same statement are unordered unless the **sequence** attribute is specified, in which case the right-to-left ordering applies. The **omp::** namespace qualifier within a **sequence** attribute is optional. The application of multiple attributes in a **sequence** attribute is ordered as if each directive had been specified as a pragma directive on subsequent lines. The **directive** attribute must not be specified inside a **sequence** attribute unless it specifies a block-associated directive.

Note – This example shows the expected transformation:

```
[[ omp::sequence(directive(parallel), directive(for)) ]]
for(...) {}
// becomes
#pragma omp parallel
#pragma omp for
for(...) {}
```

The pragma and attribute forms are interchangeable for any directive. Some directives may be composed of consecutive attribute specifiers if specified in their syntax. Any two consecutive attribute specifiers may be reordered or expressed as a single attribute specifier, as permitted by the base language, without changing the behavior of the directive.

Directives are case-sensitive. Each expression used in the OpenMP syntax inside of a clause must be a valid *assignment-expression* of the base language unless otherwise specified.

C / C++

C++

Directives may not appear in **constexpr** functions or in constant expressions.

C++

Fortran

A directive for Fortran is specified with a stylized comment as follows:

```
sentinel directive-specification
```

All directives must begin with a directive *sentinel*. The format of a sentinel differs between fixed form and free form source files, as described in Section 5.1.1 and Section 5.1.2. In order to simplify the presentation, free form is used for the syntax of directives for Fortran throughout this document, except as noted.

Directives are case insensitive. Directives cannot be embedded within continued statements, and statements cannot be embedded within directives. Each expression used in the OpenMP syntax inside of a clause must be a valid *expression* of the base language unless otherwise specified.

Fortran

A directive may be categorized as one of the following:

- declarative directive;
- executable directive;
- informational directive;
- metadirective;
- subsidiary directive; or
- utility directive.

Base language code can be associated with directives. A directive may be categorized by its base language code association as one of the following:

- block-associated directive;
- declaration-associated directive;
- delimited directive;
- explicitly associated directive;
- loop-nest-associated directive;
- loop-sequence-associated directive;
- separating directive; or
- unassociated directive.

A directive and its associated base language code (if any) constitute a syntactic formation that follows the syntax given below unless otherwise specified. The *end-directive* in a specified formation refers to the paired end directive for the directive. A construct is a formation for an executable directive. An end directive is considered a subsidiary directive of a construct if it is the end directive of that construct.

Unassociated directives are not directly associated with any base language code. The resulting formation therefore has the following syntax:

directive

Unassociated directives that are declarative directives declare identifiers for use in other directives. Unassociated directives that are executable directives are stand-alone directives.

Explicitly associated directives are declarative directives that take a variable or extended list as a directive or clause argument that indicates the declarations with which the directive is associated. As a result, explicitly associated directives have the same syntax as the formation for unassociated directives.

Formations that result from a block-associated directive have the following syntax:

C / C++

```
directive
    structured-block
```

C / C++

Fortran

```
directive
    structured-block
[end-directive]
```

If *structured-block* is a loosely structured block, *end-directive* is required, unless otherwise specified. If *structured-block* is a strictly structured block, *end-directive* is optional. An *end-directive* that immediately follows a directive and its associated strictly structured block is always paired with that directive.

Fortran

Loop-nest-associated directives are block-associated directives for which the associated *structured-block* is *loop-nest*, a canonical loop nest. Loop-sequence-associated directives are block-associated directives for which the associated *structured-block* is *canonical-loop-sequence*, a canonical loop sequence.

Fortran

The associated structured block of a block-associated directive can be a **DO CONCURRENT** loop where it is explicitly allowed.

For a loop-nest-associated directive, the paired end directive is optional.

Fortran

A declaration-associated directive is directly associated with a base language declaration.

C / C++

Formations that result from a declaration-associated directive have the following syntax:

declaration-associated-specification

where *declaration-associated-specification* is either:

```
directive
   function-definition-or-declaration
```

or:

```
directive
declaration-associated-specification
```

In all cases the directive is associated with the *function-definition-or-declaration*.

C / C++

Fortran

The formation that results from a declaration-associated directive in Fortran has the same syntax as the formation for an unassociated directive as the associated declaration is determined directly from the specification part in which the directive appears.

Fortran

Fortran / C++

If a directive appears in the specification part of a module then the behavior is as if that directive, with the variables, types and procedures that have **PRIVATE** accessibility omitted, appears in the specification part of any compilation unit that references the module unless otherwise specified.

Fortran / C++

The formation that results from a delimited directive has the following syntax:

```
directive
   base-language-code
end-directive
```

Separating directives are used to split statements contained in the associated structured block of a block-associated directive (the separated construct) into multiple structured block sequences. If the separated construct is a loop-nest-associated construct then any separating directives divide the loop body of the innermost affected loop into structured block sequences. Otherwise, the separating directives divide the associated structured block into structured block sequences.

Separating directives and the containing structured block have the following syntax:

```
structured-block-sequence
directive
structured-block-sequence
[directive
structured-block-sequence ...]
```

wrapped in a single compound statement for C/C++ or optionally wrapped in a single **BLOCK** construct for Fortran.

C / C++

Formations that result from directives that are specified as attribute specifiers that use the **directive** attribute are specified as follows. If the directive is an unassociated directive, the resulting formation is an *attribute-declaration* if the directive is not executable and it consists of the attribute specifier and a null statement (i.e., "**;**") if the directive is an executable directive. For a block-associated directive, the resulting formation consists of the attribute specifier and a structured block to which the specifier applies. If the directives are separating directives or delimited directives then the resulting formation is as specified above for those associations except that the attribute specifier for each directive, including the end directive, applies to a null statement.

A declarative directive that is a declaration-associated directive may alternatively be expressed as an attribute specifier:

```
[[ omp :: decl( directive-specification ) ]]
```

C++

or

```
[[ using omp : decl( directive-specification ) ]]
```

C++

An explicitly associated directive may alternatively be expressed with an attribute specifier that also uses the **decl** attribute, applies to a variable and/or function declaration, and omits the variable list or extended list argument. The effect is as if the omitted list argument is the list of declared variables and/or functions to which the attribute specifier applies.

Formations that result from directives that are specified as attribute specifiers and are declaration-associated directives or use the **decl** attribute are specified as follows. If the directives are declaration-associated directives then the resulting formation consists of the attribute specifiers and the *function-definition-or-declaration* to which the specifiers apply. If the directive uses the **decl** attribute then the resulting formation consists of the attribute specifier and the variable and/or function declarations to which the specifier applies.

C / C++

Restrictions

Restrictions to directive format are as follows:

C / C++

- A *directive-name* must not include white space except where explicitly allowed.

C / C++

- Orphaned separating directives are prohibited. That is, the separating directives must appear within the structured block associated with the same construct with which it is associated and must not be encountered elsewhere in the region of that separated construct.
- A stand-alone directive may be placed only at a point where a base language executable statement is allowed.

Fortran

- A declarative directive must be specified in the specification part after all **USE**, **IMPORT** and **IMPLICIT** statements.

Fortran

C / C++

- A directive that uses the attribute syntax cannot be applied to the same statement or associated declaration as a directive that uses the pragma syntax.
- For any directive that has a paired end directive, both directives must use either the attribute syntax or the pragma syntax.
- The directive and subsidiary directives of a construct must all use the attribute syntax or must all use the pragma syntax.
- Neither a stand-alone directive nor a declarative directive may be used in place of a substatement in a selection statement or iteration statement, or in place of the statement that follows a label.
- If a declarative directive applies to a function declaration or definition and it is specified with one or more C or C++ attribute specifiers, the specified attributes must be applied to the function as permitted by the base language.

C / C++

Fortran

5.1.1 Free Source Form Directives

The following sentinels are recognized in free form source files:

```
!$omp | !$ompx
```

The sentinel can appear in any column as long as it is preceded only by white space. It must appear as a single word with no intervening white space. Fortran free form line length and white space rules apply to the directive line. The syntax that allows white space to be optional has been deprecated. Initial directive lines must have a space after the sentinel. The initial line of a directive must not be a continuation line for a base language statement. Fortran free form continuation rules apply. Thus, continued directive lines must have an ampersand (**&**) as the last non-blank character on the line, prior to any comment placed inside the directive; continuation directive lines can have an ampersand after the directive sentinel with optional white space before and after the ampersand.

Comments may appear on the same line as a directive. The exclamation point (**!**) initiates a comment. The comment extends to the end of the source line and is ignored. If the first non-blank character after the directive sentinel is an exclamation point, the line is ignored.

Fortran

Fortran

5.1.2 Fixed Source Form Directives

The following sentinels are recognized in fixed form source files:

```
!$omp | c$omp | *$omp | !$omx | c$omx | *$omx
```

Sentinels must start in column 1 and appear as a single word with no intervening characters. Fortran fixed form line length, white space, continuation, and column rules apply to the directive line. The syntax that allows white space to be optional has been deprecated. Initial directive lines must have a space or a zero in column 6, and continuation directive lines must have a character other than a space or a zero in column 6.

Comments may appear on the same line as a directive. The exclamation point initiates a comment when it appears after column 6. The comment extends to the end of the source line and is ignored. If the first non-blank character after the directive sentinel of an initial or continuation directive line is an exclamation point, the line is ignored.

Fortran

5.2 Clause Format

This section defines the format and categories of OpenMP clauses. Clauses are specified as part of a *directive-specification*. Clauses have the optional property and, thus, may be omitted from a *directive-specification* unless otherwise specified, in which case they have the required property. The order in which clauses appear on directives is not significant unless otherwise specified. Some clauses form natural groupings that have similar semantic effect and so are frequently specified as a clause group. A *clause-specification* specifies each clause in a *directive-specification* where *clause-specification* is:

clause-name[**(***clause-argument-specification [* **;** *clause-argument-specification [* **;** *...]]* **)** *]*

C / C++

White space in a *clause-name* is prohibited. White space within a *clause-argument-specification* and between another *clause-argument-specification* is optional.

C / C++

An implementation may allow clauses with clause names that start with the **ompx_** prefix for use on any OpenMP directive, and the format and semantics of any such clause is implementation defined.

The first *clause-argument-specification* is required unless otherwise explicitly specified while additional ones are only permitted on clauses that explicitly allow them. When the first one is omitted, the syntax is simply:

clause-name

Clause arguments may be unmodified or modified. For an unmodified argument, *clause-argument-specification* is:

clause-argument-list

Unless otherwise specified, modified arguments have the pre-modified property, in which case the format is:

[modifier-specification-list **:** *]clause-argument-list*

Some modified arguments are explicitly specified to have the post-modified property, in which case the format is:

clause-argument-list[**:** *modifier-specification-list]*

For many clauses, *clause-argument-list* is an OpenMP argument list, which is a comma-separated list of a specific kind of list items (see Section 5.2.1), in which case the format of *clause-argument-list* is:

argument-name

For all other clauses, *clause-argument-list* is a comma-separated list of arguments so the format is:

*argument-name [***,** *argument-name [***,** *...]]*

In most of these cases, the list only has a single item so the format of *clause-argument-list* is again:

argument-name

In all cases, white space in *clause-argument-list* is optional.

A *modifier-specification-list* is a comma-separated list of clause argument modifiers for which the format is:

*modifier-specification [***,** *modifier-specification [***,** *...]]*

Clause argument modifiers may be simple modifiers or complex modifier. Many clause argument modifiers are simple modifiers, for which the format of *modifier-specification* is:

modifier-name

The format of a complex modifier is:

modifier-name[**(***modifier-parameter-specification***)** *]*

where *modifier-parameter-specification* is a comma-separated list of arguments as defined above for *clause-argument-list*. The position of each *modifier-argument-name* in the list is significant. The *modifier-parameter-specification* and parentheses are required unless every *modifier-argument-name* is optional and omitted, in which case the format of the complex modifier is identical to that of a simple modifier:

modifier-name

Each *argument-name* and *modifier-name* is an OpenMP term that may be used in the definitions of the clause and any directives on which the clause may appear. Syntactically, each of these terms is one of the following:

- *keyword*: An OpenMP keyword;
- *OpenMP identifier*: An OpenMP identifier;
- *OpenMP argument list*: An OpenMP argument list;
- *expression*: An expression of some OpenMP type; or
- *OpenMP stylized expression*: An OpenMP stylized expression.

A particular lexical instantiation of an argument specifies a parameter of the clause, while a lexical instantiation of a modifier and its parameters affects how or when the argument is applied.

The order of arguments must match the order in the *clause-specification* or *modifier-specification*. The order of modifiers in a *clause-argument-specification* is not significant unless otherwise specified.

General syntactic properties govern the use of clauses, clause and directive arguments, and modifiers in a directive. These properties are summarized in Table 5.1, along with the respective default properties for clauses, arguments and modifiers.

TABLE 5.1: Syntactic Properties for Clauses, Arguments and Modifiers

Property	Property Description	Inverse Property	Clause defaults	Argument defaults	Modifier defaults
required	must be present	optional	optional	required	optional
unique	may appear at most once	repeatable	repeatable	unique	unique
exclusive	must appear alone	compatible	compatible	compatible	compatible
ultimate	must lexically appear last (or first for a modifier on a clause with the post-modified property)	free	free	free	free

A clause, argument or modifier with a given property implies that it does not have the corresponding inverse property, and vice versa. The ultimate property implies the unique property. If all arguments and modifiers of an argument-modified clause or directive are optional property and omitted then the parentheses of the syntax for the clause or directive is also omitted.

Arguments of directives, clauses and modifiers are never repeatable. Instead, argument lists are used whenever the corresponding semantics may be specified for multiple list items that serve as the arguments of the directives, clauses or modifiers.

Some clause properties determine the constituent directives to which they apply when specified on compound directives. A clause with the all-constituents property applies to all constituent directives of any compound directive on which it is specified. Unless otherwise specified, a clause has the all-constituents property. That is, the all-constituents property is a default clause property. A clause with the once-for-all-constituents property applies to the directive once, before any of the constituent directives are applied. A clause with the innermost-leaf property applies to the innermost constituent directive to which it may be applied. A clause with the outermost-leaf property applies to the outermost constituent directive to which it may be applied. A clause with the all-privatizing property applies to all constituent directives that permit the clause and to which a data-sharing attribute clause that may create a private copy of the same list item is applied.

Arguments and modifiers that are expressions may additionally have any of the following value properties: the constant property; the positive property; the non-negative property; and the region-invariant property.

Note – In this example, *clause-specification* is **depend(inout: d)**, *clause-name* is **depend** and *clause-argument-specification* is **inout: d**. The **depend** clause has an argument for which *argument-name* is *locator-list*, which syntactically is the OpenMP locator list **d** in the example. Similarly, the **depend** clause accepts a simple modifier with the name *task-dependence-type*. Syntactically, *task-dependence-type* is the keyword **inout** in the example.

```
#pragma omp depobj(o) depend(inout: d)
```

The clauses that a directive accepts may form clause sets. These clause sets may imply restrictions on their use on that directive or may otherwise capture properties for the clauses on the directive. While specific properties may be defined for a clause set on a particular directive, the following clause set properties have general meanings and implications as indicated by the restrictions below: the required property, the unique property, and the exclusive property.

All clauses that are specified as a clause group form a clause set for which properties are specified with the specification of the clause group. Some directives accept a clause group for which each member is a *directive-name* of a directive that has a specific property. These clause groups have the required property, the unique property and the exclusive property unless otherwise specified.

The restrictions for a directive apply to the union of the clauses on the directive and its paired end directive.

Restrictions

Restrictions to clauses and clause sets are as follows:

- A clause with the required property for a directive must appear on the directive.
- A clause with the unique property for a directive may appear at most once on the directive.

- A clause with the exclusive property for a directive must not appear if a clause with a different *clause-name* also appears on the directive.
- An ultimate clause, that is one that has the ultimate property for a directive, must be the lexically last clause to appear on the directive.
- If a clause set has the required property, at least one clause in the set must be present on the directive for which the clause set is specified.
- If a clause is a member of a clause set that has the unique property for a directive then the clause has the unique property for that directive regardless of whether it has the unique property when it is not part of such a clause set.
- If one clause of a clause set with the exclusive property appears on a directive, no other clauses with a different *clause-name* in that clause set may appear on the directive.
- An argument with the required property must appear in the *clause-specification*, unless otherwise specified.
- An argument with the unique property may appear at most once in a *clause-argument-specification*.
- An argument with the exclusive property must not appear if an argument with a different *argument-name* appears in the *clause-argument-specification*.
- A modifier with the required property must appear in the *clause-argument-specification*.
- A modifier with the unique property may appear at most once in a *clause-argument-specification*.
- A modifier with the exclusive property must not appear if a modifier with a different *modifier-name* also appears in the *clause-argument-specification*.
- If a clause has the pre-modified property, a modifier with the ultimate property must be the last modifier in any *clause-argument-specification* in which any modifier appears.
- If a clause has the post-modified property, a modifier with the ultimate property must be the first modifier in any *clause-argument-specification* in which any modifier appears.
- A modifier that is an expression must neither lexically match the name of a simple modifier defined for the clause that is an OpenMP keyword nor *modifier-name parenthesized-tokens*, where *modifier-name* is the *modifier-name* of a complex modifier defined for the clause and *parenthesized-tokens* is a token sequence that starts with `(` and ends with `)`.
- An argument or parameter with the constant property must be a compile-time constant.
- An argument or parameter with the positive property must be greater than zero.
- An argument or parameter with the non-negative property must be greater than or equal to zero.
- An argument or parameter with the region-invariant property must have the same value throughout any given execution of the construct or, for declarative directives, execution of the procedure with which the declaration is associated.

Cross References

- Directive Format, see Section 5.1
- OpenMP Argument Lists, see Section 5.2.1
- OpenMP Stylized Expressions, see Section 6.2
- OpenMP Types and Identifiers, see Section 6.1

5.2.1 OpenMP Argument Lists

The OpenMP API defines several kinds of lists, each of which can be used as syntactic instances of directive, clause and modifier arguments. These comma-separated argument lists allow the corresponding semantics to apply to multiple list items. In any argument list the separation of list items has precedence for commas over any base language semantics for commas. Thus, application of base language semantics for commas to any expression in an argument list may require the use of parentheses.

A list of any OpenMP type consists of a comma-separated collection of one or more expressions of that OpenMP type. A parameter list consists of a comma-separated collection of one or more parameter list items. A variable list consists of a comma-separated collection of one or more variable list items. An extended list consists of a comma-separated collection of one or more extended list items, each of which is a variable list item or the name of a procedure. A locator list consists of a comma-separated collection of one or more locator list items. A type-name list consists of a comma-separated collection of one or more type-name list items. A directive-name list consists of a comma-separated collection of one or more directive-name list items, each of which is a directive name. A directive-specification list consists of a comma-separated collection of one or more directive-specification list items, each of which is a directive specification. A preference specification list consists of a comma-separated collection of one or more preference specification list items, each of which is a preference specification as defined in Section 16.1.3. An OpenMP operation list consists of a comma-separated collection of one or more OpenMP operation list items, each of which is a OpenMP operation defined in Section 5.2.3. An iterator-specifier list consists of a comma-separated collection of one or more iterator-specifier list items, each of which is an iterator specifier defined in Section 5.2.6.

A parameter list item can be one of the following:

- A named parameter list item;
- The position of a parameter in a parameter specification specified by a positive integer, where 1 represents the first parameter; or
- A parameter range specified by lb : ub where both lb and ub must be an expression of integer OpenMP type with the constant property and the positive property.

In both lb and ub, an expression using **`omp_num_args`**, that enables identification of parameters relative to the last argument of the call, can be used with the form:

`omp_num_args` *[± logical_offset]*

where *logical_offset* is an expression of integer OpenMP type with the constant property and the non-negative property. The lb and ub expressions are both optional. If lb is not specified the first element of the range will be 1. If ub is not specified the last element of the range will be **omp_num_args**. The effect of a specified range of $lb..ub$ is as if the parameters $lb^{th}, (lb+1)^{th}, ..., ub^{th}$ had been specified individually.

C / C++

A named parameter list item is the name of a function parameter. A variable list item is a variable or an array section. A locator list item is a reserved locator, an array section, or any lvalue expression including variables. A type-name list item is a type name.

C / C++

Fortran

A named parameter list item is a dummy argument of a subroutine or function. A variable list item is one of the following:

- a variable that is not coindexed and that is not a substring;
- an array section that is not coindexed and that does not contain an element that is a substring;
- a named constant;
- a procedure pointer;
- an associate name that may appear in a variable definition context; or
- a common block name (enclosed in slashes).

A locator list item is a variable list item, a function reference with data pointer result, or a reserved locator. A type-name list item is a type specifier.

When a named common block appears in an argument list, it has the same meaning and restrictions as if every explicit member of the common block appeared in the list. An explicit member of a common block is a variable that is named in a **COMMON** statement that specifies the common block name and is declared in the same scoping unit in which the clause appears. Named common blocks do not include the blank common block.

Fortran

Restrictions

The restrictions to argument lists are as follows:

- All list items must be visible, according to the scoping rules of the base language.
- Unless otherwise specified, OpenMP list items other than parameter list items must be directive-wide unique, i.e., a list item can only appear once in one OpenMP list of all arguments, clauses, and modifiers of the directive.
- Unless otherwise specified, any given parameter list item can only be specified once across all clauses of the same type in a given directive.

- The *directive-specifier* and the clauses in a directive-specification list item must not be comma-separated.

C

- Unless otherwise specified, a variable that is part of an aggregate variable must not be a variable list item or an extended list item.

C

C++

- Unless otherwise specified, a variable that is part of an aggregate variable must not be a variable list item or an extended list item except if the list appears on a clause that is associated with a construct within a class non-static member function and the variable is an accessible data member of the object for which the non-static member function is invoked.

C++

Fortran

- A named constant or a procedure pointer can appear as a list item only in clauses where it is explicitly allowed.
- Unless otherwise specified, a variable that is part of an aggregate variable must not be a variable list item or an extended list item.
- Unless otherwise specified, an assumed-type variable must not be a variable list item, an extended list item, or a locator list item.
- A type-name list item must not specify an abstract type or be either **`CLASS(*)`** or **`TYPE(*)`**.
- Since common block names cannot be accessed by use association or host association, a common block name specified in a clause must be declared to be a common block in the same scoping unit in which the clause appears.

Fortran

5.2.2 Reserved Locators

On some directives, some clauses accept the use of reserved locators as special OpenMP identifiers that represent system storage not necessarily bound to any base language storage item. The reserved locators are:

```
omp_all_memory
```

The reserved locator **`omp_all_memory`** is an OpenMP identifier that denotes a list item treated as having storage that corresponds to the storage of all other objects in memory.

Restrictions

Restrictions to the reserved locators are as follows:

- Reserved locators may only appear in clauses and directives where they are explicitly allowed and may not otherwise be referenced in an OpenMP program.

5.2.3 OpenMP Operations

On some directives, some clauses accept the use of OpenMP operations. An OpenMP operation named <*generic_name*> is a special expression that may be specified in an OpenMP operation list and that is used to return an object of the <*generic_name*> OpenMP type (see Section 6.1). In general, the format of an OpenMP operation is the following:

<*generic_name*> **(***operation-parameter-specification***)**

C / C++

5.2.4 Array Shaping

If an expression has a type of pointer to *T*, then a shape-operator can be used to specify the extent of that pointer. In other words, the shape-operator is used to reinterpret, as an n-dimensional array, the region of memory to which that expression points.

Formally, the syntax of the shape-operator is as follows:

shaped-expression := $([s_1][s_2]...[s_n])$ *cast-expression*

The result of applying the shape-operator to an expression is an lvalue expression with an n-dimensional array type with dimensions $s_1 \times s_2 \ldots \times s_n$ and element type *T*.

The precedence of the shape-operator is the same as a type cast.

Each s_i is an integral type expression that must evaluate to a positive integer.

Restrictions

Restrictions to the shape-operator are as follows:

- The type *T* must be a complete type.
- The shape-operator can appear only in clauses for which it is explicitly allowed.
- The result of a shape-operator must be a containing array of the list item or a containing array of one of its named pointers.
- The type of the expression upon which a shape-operator is applied must be a pointer type.

C++

- If the type *T* is a reference to a type *T'*, then the type will be considered to be *T'* for all purposes of the designated array.

C++

C / C++

5.2.5 Array Sections

An array section designates a subset of the elements in an array.

C / C++

To specify an array section in an OpenMP directive, array subscript expressions are extended with one of the following syntaxes:

```
[ lower-bound : length : stride]
[ lower-bound : length : ]
[ lower-bound : length ]
[ lower-bound : : stride]
[ lower-bound : : ]
[ lower-bound : ]
[ : length : stride]
[ : length : ]
[ : length ]
[ : : stride]
[ : : ]
[ : ]
```

The array section must be a subset of the original array.

Array sections are allowed on multidimensional arrays. Base language array subscript expressions can be used to specify length-one dimensions of multidimensional array sections.

Each of the *lower-bound*, *length*, and *stride* expressions if specified must be an integral type *expression* of the base language. When evaluated they represent a set of integer values as follows:

{ *lower-bound*, *lower-bound* + *stride*, *lower-bound* + 2 * *stride*,... , *lower-bound* + ((*length* - 1) * *stride*) }

The *length* must evaluate to a non-negative integer.

The *stride* must evaluate to a positive integer.

When the *stride* is absent it defaults to 1.

When the *length* is absent and the size of the dimension is known, it defaults to $\lceil (size - lower\text{-}bound)/stride \rceil$, where *size* is the size of the array dimension. When the *length* is absent and the size of the dimension is not known, the array section is an assumed-size array.

When the *lower-bound* is absent it defaults to 0.

The precedence of a subscript operator that uses the array section syntax is the same as the precedence of a subscript operator that does not use the array section syntax.

Note – The following are examples of array sections:

```
a[0:6]
a[0:6:1]
a[1:10]
a[1:]
a[:10:2]
b[10][:][:]
b[10][:][:0]
c[42][0:6][:]
c[42][0:6:2][:]
c[1:10][42][0:6]
S.c[:100]
p->y[:10]
this->a[:N]
(p+10)[:N]
```

Assume **a** is declared to be a 1-dimensional array with dimension size 11. The first two examples are equivalent, and the third and fourth examples are equivalent. The fifth example specifies a stride of 2 and therefore is not contiguous.

Assume **b** is declared to be a pointer to a 2-dimensional array with dimension sizes 10 and 10. The sixth example refers to all elements of the 2-dimensional array given by **b[10]**. The seventh example is a zero-length array section.

Assume **c** is declared to be a 3-dimensional array with dimension sizes 50, 50, and 50. The eighth example is contiguous, while the ninth and tenth examples are not contiguous.

The final four examples show array sections that are formed from more general array bases.

The following are examples that are non-conforming array sections:

```
s[:10].x
p[:10]->y
*(xp[:10])
```

For all three examples, a base language operator is applied in an undefined manner to an array section. The only operator that may be applied to an array section is a subscript operator for which the array section appears as the postfix expression.

C / C++

Fortran

Fortran has built-in support for array sections although some restrictions apply to their use in OpenMP directives, as enumerated at the end of this section.

Fortran

Restrictions

Restrictions to array sections are as follows:

- An array section can appear only in clauses for which it is explicitly allowed.
- A *stride* expression may not be specified unless otherwise stated.

C / C++

- An assumed-size array can appear only in clauses for which it is explicitly allowed.
- An element of an array section with a non-zero size must have a complete type.
- The array base of an array section must have an array or pointer type.
- If a consecutive sequence of array subscript expressions appears in an array section, and the first subscript expression in the sequence uses the extended array section syntax defined in this section, then only the last subscript expression in the sequence may select array elements that have a pointer type.

C / C++

C++

- If the type of the array base of an array section is a reference to a type *T*, then the type will be considered to be *T* for all purposes of the array section.
- An array section cannot be used in an overloaded `[]` operator.

C++

Fortran

- If a stride expression is specified, it must be positive.
- The upper bound for the last dimension of a dummy assumed-size array must be specified.
- If a list item is an array section with vector subscripts, the first array element must be the lowest in the array element order of the array section.
- If a list item is an array section, the last *part-ref* of the list item must have a section subscript list.

Fortran

5.2.6 `iterator` Modifier

Modifiers

Name	Modifies	Type	Properties
iterator	*locator-list*	Complex, name: **`iterator`** Arguments: ***iterator-specifier*** list of iterator specifier list item type (*default*)	unique

Clauses

`affinity`, **`depend`**, **`from`**, **`map`**, **`to`**

An *iterator* modifier is a unique, complex modifier that defines a set of iterators, each of which is an *iterator-identifier* and an associated iterator value set. An *iterator-identifier* expands to those values in the clause argument for which it is specified. Each list item of the *iterator* argument is an iterator specifier with this format:

C / C++

[iterator-type] *iterator-identifier* `=` *range-specification*

C / C++

Fortran

[iterator-type `::` *]* *iterator-identifier* `=` *range-specification*

Fortran

where:

- *iterator-identifier* is a base language identifier.
- *iterator-type* is a type that is permitted in a type-name list.
- *range-specification* is of the form *begin* `:` *end[* `:` *step]*, where *begin* and *end* are expressions for which their types can be converted to *iterator-type* and *step* is an integral expression.

C / C++

In an iterator specifier, if the *iterator-type* is not specified then that iterator is of **`int`** type.

C / C++

Fortran

In an iterator specifier, if the *iterator-type* is not specified then that iterator has default integer type.

Fortran

In a *range-specification*, if the *step* is not specified its value is implicitly defined to be 1.

An iterator only exists in the context of the clause argument that its *iterator* modifier modifies. An iterator also hides all accessible symbols with the same name in the context of that clause argument.

The use of a variable in an expression that appears in the *range-specification* causes an implicit reference to the variable in all enclosing constructs.

C / C++

The iterator value set of the iterator are the set of values $i_0, \ldots, i_{N-1}$ where:

- $i_0 = (\textit{iterator-type})\ begin$;
- $i_j = (\textit{iterator-type})\ (i_{j-1} + step)$, where $j \geq 1$; and
- if $step > 0$,
  - $i_0 < (\textit{iterator-type})\ end$;
  - $i_{N-1} < (\textit{iterator-type})\ end$; and
  - $(\textit{iterator-type})\ (i_{N-1} + step) \geq (\textit{iterator-type})\ end$;
- if $step < 0$,
  - $i_0 > (\textit{iterator-type})\ end$;
  - $i_{N-1} > (\textit{iterator-type})\ end$; and
  - $(\textit{iterator-type})\ (i_{N-1} + step) \leq (\textit{iterator-type})\ end$.

C / C++

Fortran

The iterator value set of the *iterator* are the set of values $i_1, \ldots, i_N$ where:

- $i_1 = begin$;
- $i_j = i_{j-1} + step$, where $j \geq 2$; and
- if $step > 0$,
  - $i_1 \leq end$;
  - $i_N \leq end$; and
  - $i_N + step > end$;
- if $step < 0$,
  - $i_1 \geq end$;
  - $i_N \geq end$; and
  - $i_N + step < end$.

Fortran

The iterator value set will be empty if no possible value complies with the conditions above.

If an *iterator-identifier* appears in a list item expression of the modified argument, the effect is as if the list item is instantiated within the clause for each member of the iterator value set, substituting each occurrence of *iterator-identifier* in the list item expression with the member of the iterator value set. If the iterator value set is empty then the effect is as if the list item was not specified.

Restrictions

Restrictions to *iterator* modifiers are as follows:

- The *iterator-type* must not declare a new type.
- For each value i in an iterator value set, the mathematical result of $i + step$ must be representable in *iterator-type*.

C / C++

- The *iterator-type* must be an integral or pointer type.
- The *iterator-type* must not be **const** qualified.

C / C++

Fortran

- The *iterator-type* must be an integer type.

Fortran

- If the step expression of a *range-specification* equals zero, the behavior is unspecified.
- Each *iterator-identifier* can only be defined once in the *modifier-parameter-specification*.
- An *iterator-identifier* must not appear in the *range-specification*.
- If an *iterator* modifier appears in a clause that is specified on a **task_iteration** directive then the loop-iteration variables of **taskloop**-affected loops of the associated **taskloop** construct must not appear in the *range-specification*.

Cross References

- **affinity** Clause, see Section 14.10
- **depend** Clause, see Section 17.9.5
- **from** Clause, see Section 7.10.2
- **map** Clause, see Section 7.9.6
- **to** Clause, see Section 7.10.1

5.3 Conditional Compilation

In implementations that support a preprocessor, the **_OPENMP** macro name is defined to have the decimal value *yyyymm* where *yyyy* and *mm* are the year and month designations of the version of the OpenMP API that the implementation supports.

Fortran

The OpenMP API requires Fortran lines to be compiled conditionally, as described in the following sections.

Fortran

Restrictions

Restrictions to conditional compilation are as follows:

- A **#define** or a **#undef** preprocessing directive in user code must not define or undefine the **_OPENMP** macro name.

Fortran

5.3.1 Free Source Form Conditional Compilation Sentinel

The following conditional compilation sentinel is recognized in free form source files:

```
!$
```

To enable conditional compilation, a line with a conditional compilation sentinel must satisfy the following criteria:

- The sentinel can appear in any column but must be preceded only by white space;
- The sentinel must appear as a single word with no intervening white space;
- Initial lines must have a blank character after the sentinel; and
- Continued lines must have an ampersand as the last non-blank character on the line, prior to any comment appearing on the conditionally compiled line.

Continuation lines can have an ampersand after the sentinel, with optional white space before and after the ampersand. If these criteria are met, the sentinel is replaced by two spaces. If these criteria are not met, the line is left unchanged.

Note – In the following example, the two forms for specifying conditional compilation in free source form are equivalent (the first line represents the position of the first 9 columns):

```
!23456789
 !$ iam = omp_get_thread_num() +      &
 !$&    index

#ifdef _OPENMP
    iam = omp_get_thread_num() +      &
   &    index
#endif
```

Fortran

Fortran

5.3.2 Fixed Source Form Conditional Compilation Sentinels

The following conditional compilation sentinels are recognized in fixed form source files:

```
!$ | *$ | c$
```

To enable conditional compilation, a line with a conditional compilation sentinel must satisfy the following criteria:

- The sentinel must start in column 1 and appear as a single word with no intervening white space;
- After the sentinel is replaced with two spaces, initial lines must have a space or zero in column 6 and only white space and numbers in columns 1 through 5; and
- After the sentinel is replaced with two spaces, continuation lines must have a character other than a space or zero in column 6 and only white space in columns 1 through 5.

If these criteria are met, the sentinel is replaced by two spaces. If these criteria are not met, the line is left unchanged.

Note – In the following example, the two forms for specifying conditional compilation in fixed source form are equivalent (the first line represents the position of the first 9 columns):

```
c23456789
!$ 10 iam = omp_get_thread_num() +
!$   &          index

#ifdef _OPENMP
   10 iam = omp_get_thread_num() +
     &            index
#endif
```

Fortran

5.4 *directive-name-modifier* Modifier

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Clauses

absent, **acq_rel**, **acquire**, **adjust_args**, **affinity**, **align**, **aligned**, **allocate**, **allocator**, **append_args**, **apply**, **at**, **atomic_default_mem_order**, **bind**, **capture**, **collapse**, **collector**, **combiner**, **compare**, **contains**, **copyin**, **copyprivate**, **default**, **defaultmap**, **depend**, **destroy**, **detach**, **device**, **device_safesync**, **device_type**, **dist_schedule**, **doacross**, **dynamic_allocators**, **enter**, **exclusive**, **fail**, **filter**, **final**, **firstprivate**, **from**, **full**, **grainsize**, **graph_id**, **graph_reset**, **has_device_addr**, **hint**, **holds**, **if**, **in_reduction**, **inbranch**, **inclusive**, **indirect**, **induction**, **inductor**, **init**, **init_complete**, **initializer**, **interop**, **is_device_ptr**, **lastprivate**, **linear**, **link**, **local**, **map**, **match**, **memscope**, **mergeable**, **message**, **no_openmp**, **no_openmp_constructs**, **no_openmp_routines**, **no_parallelism**, **nocontext**, **nogroup**, **nontemporal**, **notinbranch**, **novariants**, **nowait**, **num_tasks**, **num_teams**, **num_threads**, **order**, **ordered**, **otherwise**, **partial**, **permutation**, **priority**, **private**, **proc_bind**, **read**, **reduction**, **relaxed**, **release**, **replayable**, **reverse_offload**, **safelen**, **safesync**, **schedule**, **self_maps**, **seq_cst**, **severity**, **shared**, **simd**, **simdlen**, **sizes**, **task_reduction**, **thread_limit**, **threads**, **threadset**, **to**, **transparent**, **unified_address**, **unified_shared_memory**, **uniform**, **untied**, **update**, **update**, **use**, **use_device_addr**, **use_device_ptr**, **uses_allocators**, **weak**, **when**, **write**

Semantics

The *directive-name-modifier* is a universal modifier that can be used on any clause. The *directive-name-modifier* specifies *directive-name*, which is the directive name of a directive, construct or constituent construct to which the clause applies. If the directive name is that of a compound construct, then the leaf constructs to which the clause applies are determined as specified in Section 19.2. If no *directive-name-modifier* is specified then the effect is as if a *directive-name-modifier* was specified with the directive name of the directive on which the clause appears.

Restrictions

Restrictions to the *directive-name-modifier* are as follows:

- The *directive-name-modifier* must specify the directive name of either the directive on which the clause appears or a constituent directive of that directive.

Cross References

- **absent** Clause, see Section 10.6.1.1
- **acq_rel** Clause, see Section 17.8.1.1
- **acquire** Clause, see Section 17.8.1.2
- **adjust_args** Clause, see Section 9.6.2
- **affinity** Clause, see Section 14.10

- **align** Clause, see Section 8.3
- **aligned** Clause, see Section 7.12
- **allocate** Clause, see Section 8.6
- **allocator** Clause, see Section 8.4
- **append_args** Clause, see Section 9.6.3
- **apply** Clause, see Section 11.1
- **at** Clause, see Section 10.2
- **atomic_default_mem_order** Clause, see Section 10.5.1.1
- **bind** Clause, see Section 13.8.1
- **capture** Clause, see Section 17.8.3.1
- **full** Clause, see Section 11.9.1
- **partial** Clause, see Section 11.9.2
- **collapse** Clause, see Section 6.4.5
- **collector** Clause, see Section 7.6.19
- **combiner** Clause, see Section 7.6.15
- **compare** Clause, see Section 17.8.3.2
- **contains** Clause, see Section 10.6.1.2
- **copyin** Clause, see Section 7.8.1
- **copyprivate** Clause, see Section 7.8.2
- **default** Clause, see Section 7.5.1
- **defaultmap** Clause, see Section 7.9.9
- **depend** Clause, see Section 17.9.5
- **destroy** Clause, see Section 5.7
- **detach** Clause, see Section 14.11
- **device** Clause, see Section 15.2
- **device_safesync** Clause, see Section 10.5.1.7
- **device_type** Clause, see Section 15.1
- **dist_schedule** Clause, see Section 13.7.1
- **doacross** Clause, see Section 17.9.7

- **dynamic_allocators** Clause, see Section 10.5.1.2
- **enter** Clause, see Section 7.9.7
- **exclusive** Clause, see Section 7.7.2
- **fail** Clause, see Section 17.8.3.3
- **filter** Clause, see Section 12.5.1
- **final** Clause, see Section 14.7
- **firstprivate** Clause, see Section 7.5.4
- **from** Clause, see Section 7.10.2
- **grainsize** Clause, see Section 14.2.1
- **graph_id** Clause, see Section 14.3.1
- **graph_reset** Clause, see Section 14.3.2
- **has_device_addr** Clause, see Section 7.5.9
- **hint** Clause, see Section 17.1
- **holds** Clause, see Section 10.6.1.3
- **if** Clause, see Section 5.5
- **in_reduction** Clause, see Section 7.6.12
- **inbranch** Clause, see Section 9.8.1.1
- **inclusive** Clause, see Section 7.7.1
- **indirect** Clause, see Section 9.9.3
- **induction** Clause, see Section 7.6.13
- **inductor** Clause, see Section 7.6.18
- **init** Clause, see Section 5.6
- **init_complete** Clause, see Section 7.7.3
- **initializer** Clause, see Section 7.6.16
- **interop** Clause, see Section 9.7.1
- **is_device_ptr** Clause, see Section 7.5.7
- **lastprivate** Clause, see Section 7.5.5
- **linear** Clause, see Section 7.5.6
- **link** Clause, see Section 7.9.8

- **local** Clause, see Section 7.14
- **map** Clause, see Section 7.9.6
- **match** Clause, see Section 9.6.1
- **memscope** Clause, see Section 17.8.4
- **mergeable** Clause, see Section 14.5
- **message** Clause, see Section 10.3
- **no_openmp** Clause, see Section 10.6.1.4
- **no_openmp_constructs** Clause, see Section 10.6.1.5
- **no_openmp_routines** Clause, see Section 10.6.1.6
- **no_parallelism** Clause, see Section 10.6.1.7
- **nocontext** Clause, see Section 9.7.3
- **nogroup** Clause, see Section 17.7
- **nontemporal** Clause, see Section 12.4.1
- **notinbranch** Clause, see Section 9.8.1.2
- **novariants** Clause, see Section 9.7.2
- **nowait** Clause, see Section 17.6
- **num_tasks** Clause, see Section 14.2.2
- **num_teams** Clause, see Section 12.2.1
- **num_threads** Clause, see Section 12.1.2
- **order** Clause, see Section 12.3
- **ordered** Clause, see Section 6.4.6
- **otherwise** Clause, see Section 9.4.2
- **permutation** Clause, see Section 11.4.1
- **priority** Clause, see Section 14.9
- **private** Clause, see Section 7.5.3
- **proc_bind** Clause, see Section 12.1.4
- **read** Clause, see Section 17.8.2.1
- **reduction** Clause, see Section 7.6.10
- **relaxed** Clause, see Section 17.8.1.3

- **release** Clause, see Section 17.8.1.4
- **replayable** Clause, see Section 14.6
- **reverse_offload** Clause, see Section 10.5.1.3
- **safelen** Clause, see Section 12.4.2
- **safesync** Clause, see Section 12.1.5
- **schedule** Clause, see Section 13.6.3
- **self_maps** Clause, see Section 10.5.1.6
- **seq_cst** Clause, see Section 17.8.1.5
- **severity** Clause, see Section 10.4
- **shared** Clause, see Section 7.5.2
- **simd** Clause, see Section 17.10.3.2
- **simdlen** Clause, see Section 12.4.3
- **sizes** Clause, see Section 11.2
- **task_reduction** Clause, see Section 7.6.11
- **thread_limit** Clause, see Section 15.3
- **threads** Clause, see Section 17.10.3.1
- **threadset** Clause, see Section 14.8
- **to** Clause, see Section 7.10.1
- **transparent** Clause, see Section 17.9.6
- **unified_address** Clause, see Section 10.5.1.4
- **unified_shared_memory** Clause, see Section 10.5.1.5
- **uniform** Clause, see Section 7.11
- **untied** Clause, see Section 14.4
- **update** Clause, see Section 17.8.2.2
- **update** Clause, see Section 17.9.4
- **use** Clause, see Section 16.1.2
- **use_device_addr** Clause, see Section 7.5.10
- **use_device_ptr** Clause, see Section 7.5.8
- **uses_allocators** Clause, see Section 8.8

- **weak** Clause, see Section 17.8.3.4
- **when** Clause, see Section 9.4.1
- **write** Clause, see Section 17.8.2.3

5.5 if Clause

Name: **if**	**Properties:** target-consistent

Arguments

Name	Type	Properties
if-expression	expression of OpenMP logical type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

cancel, **parallel**, **simd**, **target**, **target_data**, **target_enter_data**, **target_exit_data**, **target_update**, **task**, **task_iteration**, **taskgraph**, **taskloop**, **teams**

Semantics

The effect of the **if** clause depends on the construct to which it is applied. If the construct is not a compound construct then the effect is described in the section that describes that construct.

Restrictions

Restrictions to the **if** clause are as follows:

- At most one **if** clause can be specified that applies to the semantics of any construct or constituent construct of a *directive-specification*.

Cross References

- **cancel** Construct, see Section 18.2
- **parallel** Construct, see Section 12.1
- **simd** Construct, see Section 12.4
- **target** Construct, see Section 15.8
- **target_data** Construct, see Section 15.7
- **target_enter_data** Construct, see Section 15.5

- **target_exit_data** Construct, see Section 15.6
- **target_update** Construct, see Section 15.9
- **task** Construct, see Section 14.1
- **task_iteration** Directive, see Section 14.2.3
- **taskgraph** Construct, see Section 14.3
- **taskloop** Construct, see Section 14.2
- **teams** Construct, see Section 12.2

5.6 init Clause

Name: init	**Properties:** innermost-leaf

Arguments

Name	Type	Properties
init-var	variable of OpenMP type	*default*

Modifiers

Name	Modifies	Type	Properties
prefer-type	*init-var*	Complex, name: **prefer_type** Arguments: ***prefer-type-specification*** list of preference specification list item type (*default*)	complex, unique
depinfo-modifier	*init-var*	Complex, Keyword: **in**, **inout**, **inoutset**, **mutexinoutset**, **out** Arguments: ***locator-list-item*** locator list item (*default*)	complex, unique
interop-type	*init-var*	Keyword: **target**, **targetsync**	repeatable
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

depobj, **interop**

Semantics

When the **init** clause appears on a **depobj** construct, it specifies that *init-var* is a depend object for which the state is set to *initialized*. The effect is that *init-var* is set to represent a dependence type and locator list item as specified by the name and argument of the *depinfo-modifier*.

When the **init** clause appears on an **interop** construct, it specifies that *init-var* is an interoperability object that is initialized to refer to the list of properties associated with any *interop-type*. For any *interop-type*, the properties **type**, **type_name**, **vendor**, **vendor_name** and **device_num** will be available. If the implementation cannot initialize *interop-var*, it is initialized to **omp_interop_none**.

The **targetsync** *interop-type* will additionally provide the **targetsync** property, which is the handle to a foreign synchronization object for enabling synchronization between OpenMP tasks and foreign tasks that execute in the foreign execution context.

The **target** *interop-type* will additionally provide the following properties:

- **device**, which will be a foreign device handle;
- **device_context**, which will be a foreign device context handle; and
- **platform**, which will be a handle to a foreign platform of the device.

Restrictions

- *init-var* must not be constant.
- If the **init** clause appears on a **depobj** construct, *init-var* must refer to a variable of **depend** OpenMP type that is *uninitialized*.
- If the **init** clause appears on a **depobj** construct then the *depinfo-modifier* has the required property and otherwise it must not be present.
- If the **init** clause appears on an **interop** construct, *init-var* must refer to a variable of **interop** OpenMP type.
- If the **init** clause appears on an **interop** construct, the *interop-type* modifier has the required property and each *interop-type* keyword has the unique property. Otherwise, the *interop-type* modifier must not be present.
- The *prefer-type* modifier must not be present unless the **init** clause appears on an **interop** construct.

Cross References

- **depobj** Construct, see Section 17.9.3
- **interop** Construct, see Section 16.1

5.7 `destroy` Clause

Name: `destroy`	**Properties:** *default*

Arguments

Name	Type	Properties
destroy-var	variable of OpenMP variable type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`depobj`, **`interop`**

Additional information

When the **`destroy`** clause appears on a **`depobj`** directive that specifies *depend-object* as a directive argument, the *destroy-var* argument may be omitted. If omitted, the effect is as if *destroy-var* refers to the *depend-object* argument.

Semantics

When the **`destroy`** clause appears on a **`depobj`** construct, the state of *destroy-var* is set to uninitialized.

When the **`destroy`** clause appears on an **`interop`** construct, the *interop-type* is inferred based on the *interop-type* used to initialize *destroy-var*, and *destroy-var* is set to the value of **`omp_interop_none`** after resources associated with *destroy-var* are released. The object referred to by *destroy-var* is unusable after destruction and the effect of using values associated with it is unspecified until it is initialized again by another **`interop`** construct.

Restrictions

- *destroy-var* must not be constant.
- If the **`destroy`** clause appears on a **`depobj`** construct, *destroy-var* must refer to a variable of **`depend`** OpenMP type that is *initialized*.
- If the **`destroy`** clause appears on an **`interop`** construct, *destroy-var* must refer to a variable of **`interop`** OpenMP type that is *initialized*.

Cross References

- **`depobj`** Construct, see Section 17.9.3
- **`interop`** Construct, see Section 16.1

6 Base Language Formats and Restrictions

This section defines concepts and restrictions on base language code used in OpenMP. The concepts help support base language neutrality for OpenMP directives and their associated semantics.

6.1 OpenMP Types and Identifiers

An OpenMP identifier is a special identifier for use within OpenMP programs for some specific purpose. For example, reduction identifiers specify the combiner OpenMP operation to use in a reduction, OpenMP mapper identifiers specify the name of a user-defined mapper, and foreign runtime identifiers specify the name of a foreign runtime.

Predefined identifiers can be used in base language code. Many predefined identifiers have the constant property, as is indicated where they are defined in this specification. The implementation implicitly declares these OpenMP identifiers and evaluates them when they are referenced in a given context.

Generic OpenMP types specify the type of expression or variable that is used in OpenMP contexts regardless of the base language. These OpenMP types support the definition of many important OpenMP concepts independently of the base language in which they are used.

Assignable OpenMP type instances are defined to facilitate base language neutrality. An assignable OpenMP type instance can be used as an argument of a construct in order for the implementation to modify the value of that instance.

C / C++

An assignable OpenMP type instance is an lvalue expression of that OpenMP type.

C / C++

Fortran

An assignable OpenMP type instance is a variable or a function reference with data pointer result of that OpenMP type.

Fortran

The logical OpenMP type supports logical variables and expressions in any base language.

C / C++

Any expression of logical OpenMP type is a scalar expression. This document uses *true* as a generic term for a non-zero integer value and *false* as a generic term for an integer value of zero.

C / C++

Fortran

Any expression of logical OpenMP type is a scalar logical expression. This document uses *true* as a generic term for a logical value of **.TRUE.** and *false* as a generic term for a logical value of **.FALSE.**.

Fortran

The integer OpenMP type supports integer variables and expressions in any base language.

C / C++

Any expression of integer OpenMP type is an integer expression.

C / C++

Fortran

Any expression of integer OpenMP type is a scalar integer expression.

Fortran

The string OpenMP type supports character string variables and expressions in any base language.

C / C++

Any expression of string OpenMP type is an expression of type qualified or unqualified **const char *** or **char *** pointing to a null-terminated character string.

C / C++

Fortran

Any expression of string OpenMP type is a character string of default kind.

Fortran

OpenMP function identifiers support procedure names in any base language. Regardless of the base language, any OpenMP function identifier is the name of a procedure as a base language identifier.

Each OpenMP type other than those specifically defined in this section has a generic name, <*generic_name*>, by which it is referred throughout this document and that is used to construct the base language construct that corresponds to that OpenMP type. Some OpenMP types are OMPD types or OMPT types; all of these OpenMP types have generic names.

C / C++

Unless otherwise specified, an OMPD trace record has a <*generic_name*> OMPD type, which corresponds to the type **ompd_record_**<*generic_name*>**_t** and an OMPD callback has a <*generic_name*> OMPD type signature, which corresponds to the type **ompd_callback_**<*generic_name*>**_fn_t**. Unless otherwise specified, all other <*generic_name*> OMPD types correspond to the type **ompd_**<*generic_name*>**_t**.

Unless otherwise specified, an OMPT trace record has a <*generic_name*> OMPT type, which corresponds to the type **ompt_record_**<*generic_name*>**_t** and an OMPT callback has a <*generic_name*> OMPT type signature, which corresponds to the type **ompt_callback_**<*generic_name*>**_t**. Unless otherwise specified, all other <*generic_name*> OMPT types correspond to the type **ompt_**<*generic_name*>**_t**.

Otherwise, unless otherwise specified, a variable of <*generic_name*> OpenMP type is a variable of type **omp_**<*generic_name*>**_t**.

C / C++

Fortran

Unless otherwise specified, the type of an OMPD trace record is not defined and the type signature of an OMPD callback is not defined. Unless otherwise specified, a variable of a <*generic_name*> OMPD type is an integer scalar variable of kind **ompd_**<*generic_name*>**_kind**.

Unless otherwise specified, the type of an OMPT trace record is not defined and the type signature of an OMPT callback is not defined. Unless otherwise specified, a variable of a <*generic_name*> OMPT type is an integer scalar variable of kind **ompt_**<*generic_name*>**_kind**.

Otherwise, unless otherwise specified, a variable of <*generic_name*> OpenMP type is an integer scalar variable of kind **omp_**<*generic_name*>**_kind**.

Fortran

Cross References

- OpenMP Foreign Runtime Identifiers, see Section 16.1.1
- OpenMP Reduction and Induction Identifiers, see Section 7.6.1
- Mapper Identifiers and **mapper** Modifiers, see Section 7.9.4

6.2 OpenMP Stylized Expressions

An OpenMP stylized expression is a base language expression that is subject to restrictions that enable its use within an OpenMP implementation. OpenMP stylized expressions often use OpenMP identifiers that the implementation binds to well-defined internal state.

Cross References

- OpenMP Collector Expressions, see Section 7.6.2.4
- OpenMP Combiner Expressions, see Section 7.6.2.1
- OpenMP Inductor Expressions, see Section 7.6.2.3
- OpenMP Initializer Expressions, see Section 7.6.2.2

6.3 Structured Blocks

This section specifies the concept of a structured block. A structured block:

- may contain infinite loops where the point of exit is never reached;
- may halt due to an IEEE exception;

C / C++

- may contain calls to **`exit()`**, **`_Exit()`**, **`quick_exit()`**, **`abort()`** or functions with a **`_Noreturn`** specifier (in C) or a **`noreturn`** attribute (in C/C++);
- may be an expression statement, iteration statement, selection statement, or try block, provided that the corresponding compound statement obtained by enclosing it in **`{`** and **`}`** would be a structured block; and

C / C++

Fortran

- may contain **`STOP`** or **`ERROR STOP`** statements.

Fortran

C / C++

A structured block sequence that consists of no statements or more than one statement may appear only for executable directives that explicitly allow it. The corresponding compound statement obtained by enclosing the sequence in **`{`** and **`}`** must be a structured block and the structured block sequence then should be considered to be a structured block with all of its restrictions.

C / C++

The remainder of this section covers OpenMP context-specific structured blocks that conform to specific syntactic forms and restrictions that are required for certain block-associated directives.

Restrictions

Restrictions to structured blocks are as follows:

- Entry to a structured block must not be the result of a branch.
- The point of exit cannot be a branch out of the structured block.

C / C++

- The point of entry to a structured block must not be a call to **`setjmp`**.
- **`longjmp`** must not violate the entry/exit criteria of structured blocks.

C / C++

C++

- **`throw`**, **`co_await`**, **`co_yield`** and **`co_return`** must not violate the entry/exit criteria of structured blocks.

C++

Fortran

- If a **BLOCK** construct appears in a structured block, that **BLOCK** construct must not contain any **ASYNCHRONOUS** or **VOLATILE** statements, nor any specification statements that include the **ASYNCHRONOUS** or **VOLATILE** attributes.

Fortran

6.3.1 OpenMP Allocator Structured Blocks

Fortran

An OpenMP allocator structured block is a context-specific structured block that is associated with an **allocators** directive. It consists of *allocate-stmt*, where *allocate-stmt* is a Fortran **ALLOCATE** statement. For an **allocators** directive, the paired end directive is optional.

Fortran

Cross References

- **allocators** Construct, see Section 8.7

6.3.2 OpenMP Function Dispatch Structured Blocks

An OpenMP function-dispatch structured block is a context-specific structured block that is associated with a **dispatch** directive. It identifies the location of a function dispatch.

C / C++

A function-dispatch structured block is an expression statement with one of the following forms:

```
lvalue-expression = target-call ( [expression-list] );
```

or

```
target-call ( [expression-list] );
```

C / C++

Fortran

A function-dispatch structured block is an expression statement with one of the following forms, where *expression* can be a variable or a function reference with data pointer result:

```
expression = target-call ( [arguments] )
```

or

```
CALL target-call [ ( [arguments] )]
```

For a **dispatch** directive, the paired end directive is optional.

Fortran

Restrictions

Restrictions to the function-dispatch structured blocks are as follows:

C++

- The *target-call* expression can only be a direct call.

C++

Fortran

- *target-call* must be a procedure name.
- *target-call* must not be a procedure pointer.

Fortran

Cross References

- **dispatch** Construct, see Section 9.7

6.3.3 OpenMP Atomic Structured Blocks

An OpenMP atomic structured block is a context-specific structured block that is associated with an **atomic** directive. The form of an atomic structured block depends on the atomic semantics that the directive enforces.

C / C++

Any instance of any atomic structured block in which any statement is enclosed in braces remains an instance of the same kind of atomic structured block.

C / C++

Fortran

Enclosing any instance of any atomic structured block in the pair of **BLOCK** and **END BLOCK** remains an instance of the same kind of atomic structured block, in which case the paired end directive is optional.

Fortran

In the following definitions:

C / C++

- *x*, *r* (result), and *v* (as applicable) are lvalue expressions with scalar type.
- *e* (expected) is an expression with scalar type.
- *d* (desired) is an expression with scalar type.
- *e* and *v* may refer to, or access, the same storage location.
- *expr* is an expression with scalar type.
- The order operation, *ordop*, is either **<** or **>**.
- *binop* is one of **+**, *****, **-**, **/**, **&**, **^**, **|**, **<<**, or **>>**.

- `==` comparisons are performed by comparing the value representation of operand values for equality after the usual arithmetic conversions; if the object representation does not have any padding bits, the comparison is performed as if with **`memcmp`**.
- For forms that allow multiple occurrences of *x*, the number of times that *x* is evaluated is unspecified but will be at least one.
- For forms that allow multiple occurrences of *expr*, the number of times that *expr* is evaluated is unspecified but will be at least one.
- The number of times that *r* is evaluated is unspecified but will be at least one.
- Whether *d* is evaluated if *x* `==` *e* evaluates to *false* is unspecified.

C / C++

Fortran

- *x* and *v* (as applicable) are either scalar variables or function references with scalar data pointer result of non-character intrinsic type or variables that are non-polymorphic scalar pointers and any length type parameter must be constant.
- *e* (expected) and *d* (desired) are either scalar expressions or scalar variables.
- *expr* is a scalar expression or scalar variable.
- *r* (result) is a scalar logical variable.
- *expr-list* is a comma-separated, non-empty list of scalar expressions and scalar variables.
- *intrinsic-procedure-name* is one of **`MAX`**, **`MIN`**, **`IAND`**, **`IOR`**, **`IEOR`**, **`PREVIOUS`**, or **`NEXT`**.
- *operator* is one of **`+`**, **`*`**, **`-`**, **`/`**, **`.AND.`**, **`.OR.`**, **`.EQV.`**, or **`.NEQV.`**.
- *equalop* is **`==`**, **`.EQ.`**, or **`.EQV.`**.
- The order operation, *ordop*, is one of **`<`**, **`.LT.`**, **`>`**, or **`.GT.`**.
- **`==`** or **`.EQ.`** comparisons are performed by comparing the physical representation of operand values for equality after the usual conversions as described in the base language, while ignoring padding bits, if any.
- **`.EQV.`** comparisons are performed as described in the base language.
- For forms that allow multiple occurrences of *x*, the number of times that *x* is evaluated is unspecified but will be at least one.
- For forms that allow multiple occurrences of *expr*, the number of times that *expr* is evaluated is unspecified but will be at least one.
- The number of times that *r* is evaluated is unspecified but will be at least one.
- Whether *d* is evaluated if *x equalop e* evaluates to *false* is unspecified.

Fortran

A read structured block can be specified for **atomic** directives that enforce atomic read semantics but not capture semantics.

C / C++

A read structured block is *read-expr-stmt*, a read expression statement that has the following form:

```
v = x;
```

C / C++

Fortran

A read structured block is *read-statement*, a read statement that has one of the following forms:

```
v = x
v => x
```

Fortran

A write structured block can be specified for **atomic** directives that enforce atomic write semantics but not capture semantics.

C / C++

A write structured block is *write-expr-stmt*, a write expression statement that has the following form:

```
x = expr;
```

C / C++

Fortran

A write structured block is *write-statement*, a write statement that has one of the following forms:

```
x = expr
x => expr
```

Fortran

An update structured block can be specified for **atomic** directives that enforce atomic update semantics but not capture semantics.

C / C++

An update structured block is *update-expr-stmt*, an update expression statement that has one of the following forms:

```
x++;
x--;
++x;
--x;
x binop= expr;
x = x binop expr;
x = expr binop x;
```

C / C++

Fortran

An update structured block is *update-statement*, an update statement that has one of the following forms:

```
x = x operator expr
x = expr operator x
x = intrinsic-procedure-name (x)
x = intrinsic-procedure-name (x, expr-list)
x = intrinsic-procedure-name (expr-list, x)
```

Fortran

A conditional-update structured block can be specified for **atomic** directives that enforce atomic conditional update semantics but not capture semantics.

C / C++

A conditional-update structured block is either *cond-expr-stmt*, a conditional expression statement that has one of the following forms:

```
x = expr ordop x ? expr : x;
x = x ordop expr ? expr : x;
x = x == e ? d : x;
```

or *cond-update-stmt*, a conditional update statement that has one of the following forms:

```
if(expr ordop x) x = expr;
if(x ordop expr) x = expr;
if(x == e) x = d;
```

C / C++

Fortran

A conditional-update structured block is *conditional-update-statement*, a conditional update statement that has one of the following forms:

```
if (x equalop e) x = d
if (x equalop e) then; x = d; end if
x = ( x equalop e ? d : x )
if (x ordop expr) x = expr
if (x ordop expr) then; x = expr; end if
x = ( x ordop expr ? expr : x )
if (expr ordop x) x = expr
if (expr ordop x) then; x = expr; end if
x = ( expr ordop x ? expr : x )
if (associated(x)) x => expr
if (associated(x)) then; x => expr; end if
if (associated(x, e)) x => expr
if (associated(x, e)) then; x => expr; end if
```

For an **atomic** construct with a read structured block, write structured block, update structured block, or conditional-update structured block, the paired end directive is optional.

Fortran

A capture structured block can be specified for **atomic** directives that enforce capture semantics. It is further categorized as write-capture structured block, update-capture structured block, or conditional-update-capture structured block, which can be specified for **atomic** directives that enforce write, update or conditional update atomic semantics in addition to capture semantics.

C / C++

A capture structured block is *capture-stmt*, a capture statement that has one of the following forms:

```
v = expr-stmt
{ v = x; expr-stmt }
{ expr-stmt v = x; }
```

If *expr-stmt* is *write-expr-stmt* or *expr-stmt* is *update-expr-stmt* as specified above then it is an update-capture structured block. If *expr-stmt* is *cond-expr-stmt* as specified above then it is a conditional-update-capture structured block. In addition, a conditional-update-capture structured block can have one of the following forms:

```
{ v = x; cond-update-stmt }
{ cond-update-stmt v = x; }
if(x == e) x = d; else v = x;
{ r = x == e; if(r) x = d; }
{ r = x == e; if(r) x = d; else v = x; }
```

C / C++

Fortran

A capture structured block has one of the following forms:

```
statement
capture-statement
```

or

```
capture-statement
statement
```

where *capture-statement* has either of the following forms:

```
v = x
v => x
```

If *statement* is *write-statement* as specified above then it is a write-capture structured block. If *statement* is *update-statement* as specified above then it is an update-capture structured block and may be used in **atomic** constructs that enforce atomic captured update semantics. If *statement* is *conditional-update-statement* as specified above then it is a conditional-update-capture structured block. In addition, for a conditional-update-capture structured block, *statement* can have either of the following forms:

```
x = expr
x => expr
```

In addition, a conditional-update-capture structured block can have one of the following forms:

```
if (cond) then
   x assign d
else
   v assign x
end if
```

or

```
r = cond
if (r) x assign d
```

or

```
r = cond
if (r) then
   x assign d
else
   v assign x
endif
```

where *assign* is either **=** or **=>** and *cond* denotes one of the following conditions:

```
x equalop e
ASSOCIATED(x)
ASSOCIATED(x, e)
```

Fortran

Restrictions

Restrictions to OpenMP atomic structured blocks are as follows:

C / C++

- In forms where *e* is assigned it must be an lvalue.
- *r* must be of integral type.
- During the execution of an **atomic** region, multiple syntactic occurrences of *x* must designate the same storage location.

- During the execution of an **atomic** region, multiple syntactic occurrences of *r* must designate the same storage location.
- During the execution of an **atomic** region, multiple syntactic occurrences of *expr* must evaluate to the same value.
- None of *v*, *x*, *r*, *d* and *expr* (as applicable) may access the storage location designated by any other symbol in the list.
- In forms that capture the original value of *x* in *v*, *v* and *e* may not refer to, or access, the same storage location.
- *binop*, *binop*`=`, *ordop*, `==`, `++`, and `--` are not overloaded operators.
- The expression *x binop expr* must be numerically equivalent to *x binop (expr)*. This requirement is satisfied if the operators in *expr* have precedence greater than *binop*, or by using parentheses around *expr* or subexpressions of *expr*.
- The expression *expr binop x* must be numerically equivalent to *(expr) binop x*. This requirement is satisfied if the operators in *expr* have precedence equal to or greater than *binop*, or by using parentheses around *expr* or subexpressions of *expr*.
- The expression *x ordop expr* must be numerically equivalent to *x ordop (expr)*. This requirement is satisfied if the operators in *expr* have precedence greater than *ordop*, or by using parentheses around *expr* or subexpressions of *expr*.
- The expression *expr ordop x* must be numerically equivalent to *(expr) ordop x*. This requirement is satisfied if the operators in *expr* have precedence equal to or greater than *ordop*, or by using parentheses around *expr* or subexpressions of *expr*.
- The expression *x* `==` *e* must be numerically equivalent to *x* `==` *(e)*. This requirement is satisfied if the operators in *e* have precedence equal to or greater than `==`, or by using parentheses around *e* or subexpressions of *e*.

C / C++

Fortran

- *x* must not have the **ALLOCATABLE** attribute.
- During the execution of an **atomic** region, multiple syntactic occurrences of *x* must designate the same storage location.
- During the execution of an **atomic** region, multiple syntactic occurrences of *r* must designate the same storage location.
- During the execution of an **atomic** region, multiple syntactic occurrences of *expr* must evaluate to the same value.
- None of *v*, *x*, *d*, *r*, *expr*, and *expr-list* (as applicable) may access the same storage location as any other symbol in the list.

- In forms that capture the original value of *x* in *v*, *v* may not access the same storage location as *e*.
- If *intrinsic-procedure-name* refers to **IAND**, **IOR**, **IEOR**, **PREVIOUS**, or **NEXT** then exactly one expression must appear in *expr-list*.
- The expression *x operator expr* must be, depending on its type, either mathematically or logically equivalent to *x operator (expr)*. This requirement is satisfied if the operators in *expr* have precedence greater than *operator*, or by using parentheses around *expr* or subexpressions of *expr*.
- The expression *expr operator x* must be, depending on its type, either mathematically or logically equivalent to *(expr) operator x*. This requirement is satisfied if the operators in *expr* have precedence equal to or greater than *operator*, or by using parentheses around *expr* or subexpressions of *expr*.
- The expression *x equalop e* must be, depending on its type, either mathematically or logically equivalent to *x equalop (e)*. This requirement is satisfied if the operators in *e* have precedence equal to or greater than *equalop*, or by using parentheses around *e* or subexpressions of *e*.
- *intrinsic-procedure-name* must refer to the intrinsic procedure name and not to other program entities.
- *operator* must refer to the intrinsic operator and not to a user-defined operator.
- Assignments must be either all intrinsic assignments or all pointer assignments.
- If the **ASSOCIATED** intrinsic function is referenced in a condition, all assignments must be pointer assignments. If pointer assignments are used, only the **ASSOCIATED** intrinsic function may be referenced in a condition.
- Unless *x* is a scalar variable or a function references with scalar data pointer result of non-character intrinsic type, intrinsic assignments, *equalop*, and *ordop* must not be used.
- Arguments to an **ASSOCIATED** intrinsic function must not have zero-sized storage sequences.

Fortran

Cross References

- **atomic** Construct, see Section 17.8.5

6.4 Loop Concepts

OpenMP semantics frequently involve loops that occur in the base language code. As detailed in this section, OpenMP defines several concepts that facilitate the specification of those semantics and their associated syntax.

6.4.1 Canonical Loop Nest Form

A loop nest has canonical loop nest form if it conforms to *loop-nest* in the following grammar:

loop-nest One of the following:

C / C++

```
for (init-expr; test-expr; incr-expr)
   loop-body
```

or

```
{
   loop-nest
}
```

C / C++

or

C++

```
for (range-decl: range-expr)
   loop-body
```

A range-based **for** loop is equivalent to a regular **for** loop using iterators, as defined in the base language. A range-based **for** loop has no loop-iteration variable.

C++

or

Fortran

```
DO [ label ] var = lb , ub [ , incr ]
   [intervening-code]
   loop-body
   [intervening-code]
[ label ] END DO
```

If the *loop-nest* is a *nonblock-do-construct*, it is treated as a *block-do-construct* for each **DO** construct.

The value of *incr* is the increment of the loop. If not specified, its value is assumed to be 1.

or

```
BLOCK
   loop-nest
END BLOCK
```

Fortran

or

loop-nest-generating-construct

or

generated-canonical-loop

loop-body One of the following:

loop-nest

or

C / C++

```
{
    [intervening-code]
    loop-body
    [intervening-code]
}
```

C / C++

or

Fortran

```
BLOCK
    [block-specification-part]
    [intervening-code]
    loop-body
    [intervening-code]
END BLOCK
```

Fortran

or if none of the previous productions match

final-loop-body

loop-nest-generating-construct
A loop-transforming construct that generates a canonical loop nest, which may be a canonical loop sequence that contains exactly one canonical loop nest.

generated-canonical-loop
A generated loop from a loop-transforming construct that has canonical loop nest form and for which the loop body matches *loop-body*.

intervening-code

C / C++

A non-empty sequence of structured blocks or declarations, referred to as intervening code. It must not contain iteration statements, **continue** statements or **break** statements that apply to the enclosing loop.

C / C++

Fortran

A non-empty structured block sequence, referred to as intervening code. It must not contain:

- loops;
- **CYCLE** statements;
- **EXIT** statements;
- array expressions;
- array references with a vector subscript;
- assignment statements where the target is an array object;
- references to elemental procedures with an array actual argument;
- references to procedures where the actual argument is an array that is not simply contiguous and the corresponding dummy argument has the **CONTIGUOUS** attribute or is an explicit-shape array or assumed-size array.

Fortran

Additionally, intervening code must not contain executable directives or calls to the OpenMP runtime API in its corresponding region. If intervening code is present, then a loop at the same depth within the loop nest is not a perfectly nested loop.

final-loop-body A structured block that terminates the scope of loops in the loop nest. If the loop nest is associated with a loop-nest-associated directive, loops in this structured block cannot be associated with that directive.

C / C++

init-expr One of the following:

var = *lb*
integer-type var = *lb*

C

pointer-type var = *lb*

C

C++

random-access-iterator-type var = lb

C++

test-expr One of the following:

var relational-op ub

ub relational-op var

relational-op One of the following:

<

<=

>

>=

!=

incr-expr One of the following:

++*var*

var++

- - *var*

var - -

var += *incr*

var - = *incr*

var = *var* + *incr*

var = *incr* + *var*

var = *var* - *incr*

The value of *incr*, respectively 1 and -1 for the increment and decrement operators, is the increment of the loop.

C / C++

var One of the following:

C / C++

A variable of a signed or unsigned integer type.

C / C++

C

A variable of a pointer type.

C

C++

A variable of a random access iterator type.

C++

Fortran

A scalar variable of integer type.

Fortran

The loop-iteration variable *var* must not be modified during the execution of *intervening-code* or *loop-body* in the loop.

lb, *ub* One of the following:

Expressions of a type compatible with the type of *var* that are loop invariant with respect to the outermost loop.

or

One of the following:
var-outer
var-outer + *a2*
a2 + *var-outer*
var-outer - *a2*
where *var-outer* is of a type compatible with the type of *var*.

or

If *var* is of an integer type, one of the following:
a2 - *var-outer*
a1 * *var-outer*
a1 * *var-outer* + *a2*
a2 + *a1* * *var-outer*
a1 * *var-outer* - *a2*
a2 - *a1* * *var-outer*
var-outer * *a1*
var-outer * *a1* + *a2*
a2 + *var-outer* * *a1*
var-outer * *a1* - *a2*
a2 - *var-outer* * *a1*
where *var-outer* is of an integer type.

lb and *ub* are loop bounds. A loop for which *lb* or *ub* refers to *var-outer* is a non-rectangular loop. If *var* is of an integer type, *var-outer* must be of an integer type with the same signedness and bit precision as the type of *var*.

The coefficient in a loop bound is 0 if the bound does not refer to *var-outer*. If a loop bound matches a form in which *a1* appears, the coefficient is *-a1* if the product of *var-outer* and *a1* is subtracted from *a2*, and otherwise the coefficient is *a1*. For other matched forms where *a1* does not appear, the coefficient is -1 if *var-outer* is subtracted from *a2*, and otherwise the coefficient is 1.

a1, *a2*, *incr* — Integer expressions that are loop invariant with respect to the outermost loop of the loop nest.

If the loop is associated with a directive, the expressions are evaluated before the construct formed from that directive.

var-outer — The loop-iteration variable of a surrounding loop in the loop nest.

C++

range-decl — A declaration of a variable as defined by the base language for range-based **`for`** loops.

range-expr — An expression that is valid as defined by the base language for range-based **`for`** loops. It must be invariant with respect to the outermost loop of the loop nest and the iterator derived from it must be a random access iterator.

C++

Restrictions

Restrictions to canonical loop nests are as follows:

C / C++

- If *test-expr* is of the form *var relational-op b* and *relational-op* is < or <= then *incr-expr* must cause *var* to increase on each iteration of the loop. If *test-expr* is of the form *var relational-op b* and *relational-op* is > or >= then *incr-expr* must cause *var* to decrease on each iteration of the loop. Increase and decrease are using the order induced by *relational-op*.
- If *test-expr* is of the form *ub relational-op var* and *relational-op* is < or <= then *incr-expr* must cause *var* to decrease on each iteration of the loop. If *test-expr* is of the form *ub relational-op var* and *relational-op* is > or >= then *incr-expr* must cause *var* to increase on each iteration of the loop. Increase and decrease are using the order induced by *relational-op*.
- If *relational-op* is **`!=`** then *incr-expr* must cause *var* to always increase by 1 or always decrease by 1 and the increment must be a constant expression.
- *final-loop-body* must not contain any **`break`** statement that would cause the termination of the innermost loop.

C / C++

Fortran

- *final-loop-body* must not contain any **`EXIT`** statement that would cause the termination of the innermost loop.

Fortran

- A *loop-nest* must also be a structured block.
- For a non-rectangular loop, if *var-outer* is referenced in *lb* and *ub* then they must both refer to the same loop-iteration variable.
- For a non-rectangular loop, let a_{lb} and a_{ub} be the respective coefficients in *lb* and *ub*, $incr_{\text{inner}}$ the increment of the non-rectangular loop and $incr_{\text{outer}}$ the increment of the loop referenced by *var-outer*. $incr_{\text{inner}}(a_{\text{ub}} - a_{\text{lb}})$ must be a multiple of $incr_{\text{outer}}$.
- The loop-iteration variable may not appear in a **threadprivate** directive.

Cross References

- Canonical Loop Sequence Form, see Section 6.4.2
- Loop-Transforming Constructs, see Chapter 11
- **threadprivate** Directive, see Section 7.3

6.4.2 Canonical Loop Sequence Form

A structured block has canonical loop sequence form if it conforms to *canonical-loop-sequence* in the following grammar:

canonical-loop-sequence

C / C++

```
{
    loop-sequence
}
```

C / C++

Fortran

One of the following:

```
loop-sequence
```

or

```
BLOCK
    loop-sequence
END BLOCK
```

Fortran

loop-sequence A structured block sequence with executable statements that match *canonical-loop-sequence*, *loop-sequence-generating-construct*, or *loop-nest* (a canonical loop nest as defined in Section 6.4.1). The loops must be bounds-independent loops with respect to *canonical-loop-sequence*.

loop-sequence-generating-construct
: A loop-transforming construct that generates a canonical loop sequence or canonical loop nest.

The loop sequence length and consecutive order of canonical loop nests matched by *loop-nest* ignore how they are nested in *canonical-loop-sequence* or *loop-sequence*.

Cross References

- **looprange** Clause, see Section 6.4.7
- Canonical Loop Nest Form, see Section 6.4.1
- Loop-Transforming Constructs, see Chapter 11

6.4.3 OpenMP Loop-Iteration Spaces and Vectors

A loop-nest-associated directive affects some number of the outermost loops of an associated loop nest, called the affected loops, in accordance with its specified clauses. These affected loops and their loop-iteration variables form an OpenMP loop-iteration vector space. OpenMP loop-iteration vectors allow other directives to refer to points in that loop-iteration vector space.

A loop-transforming construct that appears inside a loop nest is replaced according to its semantics before any loop can be associated with a loop-nest-associated directive that is applied to the loop nest. The loop nest depth is determined according to the loops in the loop nest, after any such replacements have taken place. A loop counts towards the loop nest depth if it is a base language loop statement or generated loop and it matches *loop-nest* while applying the production rules for canonical loop nest form to the loop nest.

The canonical loop nest form allows the iteration count of all affected loops to be computed before executing the outermost loop. For any affected loop, the iteration count is computed as follows:

C / C++

- If *var* has a signed integer type and the *var* operand of *test-expr* after usual arithmetic conversions has an unsigned integer type then the loop iteration count is computed from *lb*, *test-expr* and *incr* using an unsigned integer type corresponding to the type of *var*.
- Otherwise, if *var* has an integer type then the loop iteration count is computed from *lb*, *test-expr* and *incr* using the type of *var*.

C / C++

C

- If *var* has a pointer type then the loop iteration count is computed from *lb*, *test-expr* and *incr* using the type **ptrdiff_t**.

C

C++

- If *var* has a random access iterator type then the loop iteration count is computed from *lb*, *test-expr* and *incr* using the type **std::iterator_traits<***random-access-iterator-type***>::difference_type**.
- For range-based **for** loops, the loop iteration count is computed from *range-expr* using the type **std::iterator_traits<***random-access-iterator-type***>::difference_type** where *random-access-iterator-type* is the iterator type derived from *range-expr*.

C++

Fortran

- The loop iteration count is computed from *lb*, *ub* and *incr* using the type of *var*.

Fortran

The behavior is unspecified if any intermediate result required to compute the iteration count cannot be represented in the type determined above.

No synchronization is implied during the evaluation of the *lb*, *ub*, *incr* or *range-expr* expressions. Whether, in what order, or how many times any side effects within the *lb*, *ub*, *incr*, or *range-expr* expressions occur is unspecified.

Let the number of loops affected with a construct be n, where all of the affected loops have a loop-iteration variable. The OpenMP loop-iteration vector space is the n-dimensional space defined by the values of var_i, $1 \leq i \leq n$, the loop-iteration variables of the affected loops, with $i = 1$ referring to the outermost loop of the loop nest. An OpenMP loop-iteration vector, which may be used as an argument of OpenMP directives and clauses, then has the form:

$$var_1\ [\pm\ offset_1],\ var_2\ [\pm\ offset_2],\ \ldots,\ var_n\ [\pm\ offset_n]$$

where $offset_i$ is a constant, non-negative expression of integer OpenMP type that facilitates identification of relative points in the loop-iteration vector space.

Alternatively, OpenMP defines a special keyword **omp_cur_iteration** that represents the current logical iteration. It enables identification of relative points in the logical iteration space with:

omp_cur_iteration *[± logical_offset]*

where *logical_offset* is a constant, non-negative expression of integer OpenMP type.

The iterations of some number of affected loops can be collapsed into one larger logical iteration space that is the collapsed iteration space. The particular integer type used to compute the iteration count for the collapsed loop is implementation defined, but its bit precision must be at least that of the widest type that the implementation would use for the iteration count of each loop if it was the only affected loop. The number of times that any intervening code between any two collapsed loops will be executed is unspecified but will be the same for all intervening code at the same depth, at least once per iteration of the loop that encloses the intervening code and at most once per collapsed logical iteration. If the iteration count of any loop is zero and that loop does not enclose the intervening code, the behavior is unspecified.

At the beginning of each collapsed iteration in a loop-collapsing construct, the loop-iteration variable or the variable declared by *range-decl* of each collapsed loop has the value that it would have if the collapsed loops were not associated with any directive.

6.4.4 Consistent Loop Schedules

A loop schedule for a given loop-nest-associated construct assigns a thread in the binding thread set of that construct to a logical iteration vector of the affected loop nest. If the loop schedules of two loop-nest-associated constructs are consistent schedules, the behavior is as if they produce the same mapping of logical iteration vectors to threads. In particular, if two loop-nest-associated construct have consistent schedules and they have the same binding thread set, the implementation will guarantee that memory effects of a logical iteration in the first loop nest have completed before the execution of the corresponding logical iteration in the second loop nest.

Two loop-nest-associated constructs have consistent schedules if all of the following conditions hold:

- The constructs have the same *directive-name*;
- The regions that correspond to the two constructs have the same binding region;
- The constructs have the same schedule specification;
- The constructs have reproducible schedules;
- The affected loops have identical logical iteration vector spaces;
- The two sets of affected loops either consist of only rectangular loops or both contain a non-rectangular loop; and
- The loop schedules of transformation-affected loops among any affected loops that are generated loops of a loop-transforming construct are all themselves consistent.

6.4.5 `collapse` Clause

Name: `collapse`	Properties: once-for-all-constituents, unique

Arguments

Name	Type	Properties
n	expression of integer type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

distribute, **do**, **for**, **loop**, **simd**, **taskloop**

Semantics

The **collapse** clause affects one or more loops of a canonical loop nest on which it appears for the purpose of identifying the portion of the depth of the canonical loop nest to which to apply the work distribution semantics of the directive. The argument *n* specifies the number of loops of the associated loop nest to which to apply those semantics. On all directives on which the **collapse** clause may appear, the effect is as if a value of one was specified for *n* if the **collapse** clause is not specified.

Restrictions

- *n* must not evaluate to a value greater than the loop nest depth.

Cross References

- **distribute** Construct, see Section 13.7
- **do** Construct, see Section 13.6.2
- **for** Construct, see Section 13.6.1
- **loop** Construct, see Section 13.8
- **simd** Construct, see Section 12.4
- **taskloop** Construct, see Section 14.2

6.4.6 ordered Clause

Name: **ordered**	**Properties:** once-for-all-constituents, unique

Arguments

Name	Type	Properties
n	expression of integer type	optional, constant, positive

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

do, **for**

Semantics

The **ordered** clause is used to specify the doacross-affected loops for the purpose of identifying cross-iteration dependences. The argument *n* specifies the number of doacross-affected loops to use for that purpose. If *n* is not specified then the behavior is as if *n* is specified with the same value as is specified for the **collapse** clause on the construct.

Restrictions

- None of the doacross-affected loops may be non-rectangular loops.
- *n* must not evaluate to a value greater than the depth of the associated loop nest.
- If *n* is explicitly specified and the **collapse** clause is also specified for the **ordered** clause on the same construct, *n* must be greater than or equal to the *n* specified for the **collapse** clause.

Cross References

- **collapse** Clause, see Section 6.4.5
- **do** Construct, see Section 13.6.2
- **for** Construct, see Section 13.6.1

6.4.7 `looprange` Clause

Name: `looprange`	**Properties:** unique

Arguments

Name	Type	Properties
first	expression of OpenMP integer type	constant, positive
count	expression of OpenMP integer type	constant, positive, ultimate

Directives

fuse

Semantics

For a loop-sequence-associated construct, the **looprange** clause determines the canonical loop nests of the associated loop sequence that are affected by the directive. The affected loop nests are the *count* consecutive canonical loop nests that begin with the canonical loop nest specified by the *first* argument.

For all directives on which the **looprange** clause may appear, if the clause is not specified then the effect is as if the clause was specified with a value equal to the loop sequence lengths of the associated canonical loop sequence.

Restrictions

Restrictions to the **looprange** clause are as follows:

- $first + count - 1$ must not evaluate to a value greater than the loop sequence length of the associated canonical loop sequence.

Cross References

- **fuse** Construct, see Section 11.3
- Canonical Loop Sequence Form, see Section 6.4.2

Part II

Directives and Clauses

7 Data Environment

This chapter presents directives and clauses for controlling data environments. These directives and clauses include the data-environment attribute clauses (or simply data-environment clauses), which explicitly determine the data-environment attributes of list items specified in an argument list. The data-environment clauses form a general clause set for which certain restrictions apply to their use on directives that accept any members of the set. In addition, these clauses are divided into two subsets that also form general clause sets: data-sharing attribute clauses (or simply data-sharing clauses) and data-mapping attribute clause (or simply data-mapping clauses). Additional restrictions apply to the use of these clause sets on directives that accept any members of them.

Data-sharing clauses control the data-sharing attributes of variables in a construct, indicating whether a variable is shared or private in the outermost scope of the construct. Any clause that indicates a variable is private in that scope is a privatization clause. Data-mapping clauses control the data-mapping attributes of variables in a data environment, indicating whether a variable is mapped from the data environment to another device data environment.

7.1 Data-Sharing Attribute Rules

This section describes how the data-sharing attributes of variables referenced in data environments are determined. The following two cases are described separately:

- Section 7.1.1 describes the data-sharing attribute rules for variables referenced in a construct.
- Section 7.1.2 describes the data-sharing attribute rules for variables referenced in a region, but outside any construct.

For any variable that is a referencing variable (including formal arguments passed by reference for C++), the data-sharing attribute rules apply only to its referring pointer unless otherwise specified.

7.1.1 Variables Referenced in a Construct

A variable that is referenced in a construct can have a predetermined data-sharing attribute, an explicitly determined data-sharing attribute, or an implicitly determined data-sharing attribute, according to the rules outlined in this section.

Specifying a variable in a **copyprivate** clause or a data-sharing attribute clause other than the **private** clause on a nested construct causes an implicit reference to the variable in the enclosing construct. Specifying a variable in a **map** clause of an enclosed construct may cause an implicit reference to the variable in the enclosing construct. Such implicit references are also subject to the data-sharing attribute rules outlined in this section.

Fortran

A type parameter inquiry or complex part designator that is referenced in a construct is treated as if its designator is referenced.

Fortran

Certain variables and objects have predetermined data-sharing attributes for the construct in which they are referenced. The first matching rule from the following list of predetermined data-sharing attribute rules applies for variables and objects that are referenced in a construct.

- Variables with automatic storage duration that are declared in a scope inside the construct are private.
- Variables and common blocks (in Fortran) that appear as arguments in **threadprivate** directives or variables with the **_Thread_local** (in C) or **thread_local** (in C/C++) storage-class specifier are threadprivate.
- Variables and common blocks (in Fortran) that appear as arguments in **groupprivate** directives are groupprivate.
- Variables and common blocks (in Fortran) that appear as list items in **local** clauses on **declare_target** directives are device-local.
- Variables with static storage duration that are declared in a scope inside the construct are shared.
- Objects with dynamic storage duration are shared.
- The loop-iteration variable in any affected loop of a **loop** or **simd** construct is lastprivate.
- The loop-iteration variable in any affected loop of a loop-nest-associated directive is otherwise private.

C++

- The implicitly declared variables of a range-based **for** loop are private.

C++

Fortran

- Loop-iteration variables inside **parallel**, **teams**, **taskgraph**, or task-generating constructs are private in the innermost such construct that encloses the loop.

Fortran

C / C++

- Variables with static storage duration that are declared in a scope inside the construct are shared.
- If a list item in a **has_device_addr** clause or in a **map** clause on the **target** construct has a base pointer, and the base pointer is a scalar variable that is not a list item in a **map** clause on the construct, the base pointer is firstprivate.

- If a list item in a **reduction** or **in_reduction** clause on the construct has a base pointer then the base pointer is private.
- Static data members are shared.
- If a list item in a **shared** clause on the construct is a referencing variable then the referring pointer of the list item is firstprivate.
- If a list item in a **map** clause on the **target** construct has a base referencing variable that does not have a containing structure, the referring pointer of the base referencing variable is firstprivate.
- The **__func__** variable and similar function-local predefined variables are shared.

C / C++

Fortran

- Assumed-size arrays and named constants are shared in constructs that are not data-mapping constructs.
- A named constant is firstprivate in **target** constructs.
- An associate name that may appear in a variable definition context is shared if its association occurs outside of the construct and otherwise it has the same data-sharing attribute as the selector with which it is associated.
- If a list item in a **map** clause on the **target** construct has a base referencing variable that is not the list item itself, the referring pointer of the base referencing variable is firstprivate unless that referencing variable is a structure element, a list item in an **enter** clause on a declare target directive, or a list item in a **map** clause on the construct where the semantics of the clause apply to its referring pointer.

Fortran

- If a list item in a **has_device_addr** clause on the **target** construct has a base referencing variable, the referring pointer of the base referencing variable is firstprivate.

Variables with predetermined data-sharing attributes may not be listed in data-sharing clauses, except for the cases listed below. For these exceptions only, listing a predetermined variable in a data-sharing clause is allowed and overrides its predetermined data-sharing attributes.

- The loop-iteration variable in any affected loop of a loop-nest-associated directive may be listed in a **private** or **lastprivate** clause.
- If a **simd** construct has just one affected loop then its loop-iteration variable may be listed in a **linear** clause with a *linear-step* that is the increment of the affected loop.

C / C++

- Variables with **const**-qualified type with no mutable members may be listed in a **firstprivate** clause, even if they are static data members.

- The `__func__` variable and similar function-local predefined variables may be listed in a **`shared`** or **`firstprivate`** clause.

C / C++

Fortran

- A loop-iteration variable of a loop that is not associated with any directive may be listed in a data-sharing attribute clause on the surrounding **`teams`**, **`parallel`** or task-generating construct, and on enclosed constructs, subject to other restrictions.
- An assumed-size array may be listed in a **`shared`** clause.
- A named constant may be listed in a **`shared`** or **`firstprivate`** clause.

Fortran

Additional restrictions on the variables that may appear in individual clauses are described with each clause in Section 7.5.

Variables with explicitly determined data-sharing attributes are those that are referenced in a given construct and are listed in a data-sharing clause on the construct. Variables with implicitly determined data-sharing attributes are those that are referenced in a given construct and do not have predetermined data-sharing attributes or explicitly determined data-sharing attributes in that construct. Rules for variables with implicitly determined data-sharing attributes are as follows:

- In a **`parallel`**, **`teams`**, or task-generating construct, the data-sharing attributes of these variables are determined by the **`default`** clause, if present (see Section 7.5.1).
- In a **`parallel`** construct, if no **`default`** clause is present, these variables are shared.
- If no **`default`** clause is present on constructs that are not task-generating constructs, these variables reference the variables with the same names that exist in the enclosing context. If no **`default`** clause is present on a task-generating construct and the generated task is a sharing task, these variables are shared.
- In a **`target`** construct, variables that are not mapped after applying data-mapping attribute rules (see Section 7.9) are firstprivate.

C++

- In an orphaned task-generating construct, if no **`default`** clause is present, formal arguments passed by reference are firstprivate.

C++

Fortran

- In an orphaned task-generating construct, if no **`default`** clause is present, dummy arguments are firstprivate.

Fortran

- In a task-generating construct, if no **default** clause is present, a variable for which the data-sharing attribute is not determined by the rules above is shared if the variable is determined to be shared by all implicit tasks bound to the current team in the enclosing context.

- In a task-generating construct, if no **default** clause is present, a variable for which the data-sharing attribute is not determined by the rules above is firstprivate.

An OpenMP program is non-conforming if a variable in a task-generating construct is implicitly determined to be firstprivate according to the above rules but is not permitted to appear in a **firstprivate** clause according to the restrictions specified in Section 7.5.4.

7.1.2 Variables Referenced in a Region but not in a Construct

The data-sharing attribute of a variable or object that is referenced in a region, but not in the corresponding construct, is determined by the first matching rule from the following list.

- Variables with automatic storage duration that are declared in called procedures in the region are private.

- Variables and common blocks (in Fortran) that appear as arguments in **threadprivate** directives or variables with the **_Thread_local** (in C) or **thread_local** (in C/C++) storage-class specifier are threadprivate.

- Variables and common blocks (in Fortran) that appear as arguments in **groupprivate** directives are groupprivate.

- Variables and common blocks (in Fortran) that appear as list items in **local** clauses on **declare_target** directives are device-local.

- Variables with static storage duration are shared.

- Objects with dynamic storage duration are shared.

Fortran

- Variables that are accessed by host or use association are shared.

- A dummy argument of a called procedure in the region that does not have the **VALUE** attribute is private if the associated actual argument is not shared.

- A dummy argument of a called procedure in the region that does not have the **VALUE** attribute is shared if the actual argument is shared and it is a scalar variable, structure, an array that is not a pointer or assumed-shape array, or a simply contiguous array section. Otherwise, the data-sharing attribute of the dummy argument is implementation defined if the associated actual argument is shared.

Fortran

7.2 `saved` Modifier

Modifiers

Name	Modifies	Type	Properties
saved	*list*	Keyword: **`saved`**	*default*

Clauses

`firstprivate`

Semantics

If the *saved* modifier is present in a data-environment attribute clause that is specified on a replayable construct then its original list items of a replay execution are defined by the saved data environment of the replayable construct. The *saved* modifier has no effect if specified in a clause that does not appear on a replayable construct.

Cross References

- **`firstprivate`** Clause, see Section 7.5.4
- **`taskgraph`** Construct, see Section 14.3

7.3 `threadprivate` Directive

Name: **`threadprivate`** **Category:** declarative	**Association:** explicit **Properties:** pure

Arguments

`threadprivate` (*list*)

Name	Type	Properties
list	list of variable list item type	*default*

Semantics

The **`threadprivate`** directive specifies that variables have the threadprivate attribute and therefore they are replicated with each thread having its own copy. Unless otherwise specified, each copy of a threadprivate variable is initialized once, in the manner specified by the program, but at an unspecified point in the program prior to the first reference to that copy. The storage of all copies of a threadprivate variable is freed according to how variables with static storage duration are handled in the base language, but at an unspecified point in the program.

C++

Each copy of a block-scope threadprivate variable that has a dynamic initializer is initialized the first time its thread encounters its definition; if its thread does not encounter its definition, whether it is initialized is unspecified. If it is initialized, its initialization occurs at an unspecified point in the program.

C++

The content of a threadprivate variable can change across a task scheduling point if the executing thread switches to another task that modifies the variable. For more details on task scheduling, see Section 1.2 and Chapter 14.

In **parallel** regions, references by the primary thread are to the copy of the variable of the thread that encountered the **parallel** region.

During a sequential part, references are to the copy of the variable of the initial thread. The values of data in the copy for the initial thread are guaranteed to persist between any two consecutive references to the threadprivate variable in the program, provided that no **teams** construct that is not nested inside of a **target** construct is encountered between the references and that the initial thread is not executing code inside of a **teams** region. For initial threads that are executing code inside of a **teams** region, the values of data in the copies of a threadprivate variable for those initial threads are guaranteed to persist between any two consecutive references to the variable inside that **teams** region.

The values of data in the threadprivate variables of threads that are not initial threads are guaranteed to persist between two consecutive active parallel regions only if all of the following conditions hold:

- Neither **parallel** region is nested inside another explicit **parallel** region;
- The sizes of the teams used to execute both **parallel** regions are the same;
- The thread affinity policies used to execute both **parallel** regions are the same;
- The value of the *dyn-var* ICV in the enclosing task region is *false* at entry to both **parallel** regions;
- No **teams** construct that is not nested inside of a **target** construct is encountered between the **parallel** regions;
- No construct with an **order** clause that specifies **concurrent** is encountered between the **parallel** regions; and
- Neither the **omp_pause_resource** nor **omp_pause_resource_all** routine is called.

If these conditions all hold, and if a threadprivate variable is referenced in both regions, then threads with the same thread number in their respective regions reference the same copy of that variable.

C / C++

If the above conditions hold, the storage duration, lifetime, and value of a copy of a threadprivate variable that does not appear in any **copyin** clause on the corresponding construct of the second region spans the two consecutive active parallel regions. Otherwise, the storage duration, lifetime, and value of the copy of the variable in the second region is unspecified.

C / C++

Fortran

If the above conditions hold, the definition, association, or allocation status of a copy of a threadprivate variable or a variable in a threadprivate common block that is not affected by any **copyin** clause that appears on the corresponding construct of the second region (a variable is affected by a **copyin** clause if the variable appears in the **copyin** clause or it is in a common block that appears in the **copyin** clause) spans the two consecutive active parallel regions. Otherwise, the definition and association status of a copy of the variable in the second region are undefined, and the allocation status of an allocatable variable are implementation defined.

If a threadprivate variable or a variable in a threadprivate common block is not affected by any **copyin** clause that appears on the corresponding construct of the first **parallel** region in which it is referenced, the copy of the variable inherits the declared type parameter and the default parameter values from the original variable. The variable or any subobject of the variable is initially defined or undefined according to the following rules:

- If it has the **ALLOCATABLE** attribute, each copy created has an initial allocation status of unallocated;
- If it has the **POINTER** attribute, each copy has the same association status as the initial association status; and
- If it does not have either the **POINTER** or the **ALLOCATABLE** attribute:
  - If it is initially defined, either through explicit initialization or default initialization, each copy created is so defined;
  - Otherwise, each copy created is undefined.

Fortran

C++

The order in which any constructors for different threadprivate variables of class type are called is unspecified. The order in which any destructors for different threadprivate variables of class type are called is unspecified. A variable that is part of an aggregate variable may appear in a **threadprivate** directive only if it is a static data member of a C++ class.

C++

Restrictions

Restrictions to the **threadprivate** directive are as follows:

- A thread must not reference a copy of a threadprivate variable that belongs to another thread.
- A threadprivate variable must not appear as the base variable of a list item in any clause except for the **copyin** and **copyprivate** clauses.
- An OpenMP program in which an untied task accesses threadprivate memory is non-conforming.

C / C++

- Each list item must be a file-scope, namespace-scope, or static block-scope variable.
- No list item may have an incomplete type.
- The address of a threadprivate variable must not be an address constant.
- If the value of a variable referenced in an explicit initializer of a threadprivate variable is modified prior to the first reference to any instance of the threadprivate variable, the behavior is unspecified.
- A **`threadprivate`** directive for file-scope variables must appear outside any definition or declaration, and must lexically precede all references to any of the variables in its argument list.
- A **`threadprivate`** directive for namespace-scope variables must appear outside any definition or declaration other than the namespace definition itself and must lexically precede all references to any of the variables in its argument list.
- Each variable in the argument list of a **`threadprivate`** directive at file, namespace, or class scope must refer to a variable declaration at file, namespace, or class scope that lexically precedes the directive.
- A **`threadprivate`** directive for a static block-scope variable must appear in the scope of the variable and not in a nested scope. The directive must lexically precede all references to any of the variables in its argument list.
- Each variable in the argument list of a **`threadprivate`** directive in block scope must refer to a variable declaration in the same scope that lexically precedes the directive. The variable must have static storage duration.
- If a variable is specified in a **`threadprivate`** directive in one compilation unit, it must be specified in a **`threadprivate`** directive in every compilation unit in which it is declared.

C / C++

C++

- A **`threadprivate`** directive for static class member variables must appear in the class definition, in the same scope in which the member variables are declared, and must lexically precede all references to any of the variables in its argument list.
- A threadprivate variable must not have an incomplete type or a reference type.
- A threadprivate variable with class type must have:
  - An accessible, unambiguous default constructor in the case of default initialization without a given initializer;
  - An accessible, unambiguous constructor that accepts the given argument in the case of direct initialization; and
  - An accessible, unambiguous copy constructor in the case of copy initialization with an explicit initializer.

C++

Fortran

- Each list item must be a named variable or a named common block; a named common block must appear between slashes.
- The *list* argument must not include any coarrays or associate names.
- The **threadprivate** directive must appear in the declaration section of a scoping unit in which the common block or variable is declared.
- If a **threadprivate** directive that specifies a common block name appears in one compilation unit, then such a directive must also appear in every other compilation unit that contains a **COMMON** statement that specifies the same name. It must appear after the last such **COMMON** statement in the compilation unit.
- If a threadprivate variable or a threadprivate common block is declared with the **BIND** attribute, the corresponding C entities must also be specified in a **threadprivate** directive in the C program.
- A variable may only appear as an argument in a **threadprivate** directive in the scope in which it is declared. It must not be an element of a common block or appear in an **EQUIVALENCE** statement.
- A variable that appears as an argument in a **threadprivate** directive must be declared in the scope of a module or have the **SAVE** attribute, either explicitly or implicitly.
- The effect of an access to a threadprivate variable in a **DO CONCURRENT** construct is unspecified.

Fortran

Cross References

- **copyin** Clause, see Section 7.8.1
- *dyn-var* ICV, see Table 3.1
- **order** Clause, see Section 12.3
- Determining the Number of Threads for a **parallel** Region, see Section 12.1.1

7.4 List Item Privatization

Some data-sharing attribute clauses, including reduction clauses, specify that list items that appear in their argument list may be privatized for the construct on which they appear. Each task that references a privatized list item in any statement in the construct receives at least one new list item if the construct is a loop-collapsing construct, and otherwise each such task receives one new list item. Each SIMD lane used in a **simd** construct that references a privatized list item in any statement in the construct receives at least one new list item. Language-specific attributes for new list items are derived from the corresponding original list items. Inside the construct, all references to the original list items are replaced by references to the new list items received by the task or SIMD lane, and the new list items have the private attribute.

If the construct is a loop-collapsing construct then, within the same collapsed logical iteration of the collapsed loops, the same new list item replaces all references to the original list item. For any two collapsed iterations, if the references to the original list item are replaced by the same new list item then the collapsed iterations must execute in some sequential order.

In the rest of the region, whether references are to a new list item or the original list item is unspecified. Therefore, if an attempt is made to reference the original list item, its value after the region is also unspecified. If a task or a SIMD lane does not reference a privatized list item, whether the task or SIMD lane receives a new list item is unspecified.

The value and/or allocation status of the original list item will change only:

- If accessed and modified via a pointer;
- If possibly accessed in the region but outside of the construct;
- As a side effect of directives or clauses; or

Fortran

- If accessed and modified via construct association.

Fortran

C++

If the construct is contained in a member function, whether accesses anywhere in the region through the implicit **this** pointer refer to the new list item or the original list item is unspecified.

C++

C / C++

A new list item of the same type, with automatic storage duration, is allocated for the construct. The storage and thus lifetime of these new list items last until the block in which they are created exits. The size and alignment of the new list item are determined by the type of the variable. This allocation occurs once for each task generated by the construct and once for each SIMD lane used by the construct.

Unless otherwise specified, the new list item is initialized, or has an undefined initial value, as if it had been locally declared without an initializer.

C / C++

C++

If the type of a list item is a reference to a type *T* then the type will be considered to be *T* for all purposes of the clause.

The order in which any default constructors for different private variables of class type are called is unspecified. The order in which any destructors for different private variables of class type are called is unspecified.

C++

Fortran

If any statement of the construct references a list item, a new list item of the same type and type parameters is allocated. This allocation occurs once for each task generated by the construct and once for each SIMD lane used by the construct. If the type of the list item has default initialization, the new list item has default initialization. Otherwise, the initial value of the new list item is undefined. The initial status of a private pointer is undefined.

For a list item or the subobject of a list item with the **ALLOCATABLE** attribute:

- If the allocation status is unallocated, the new list item or the subobject of the new list item will have an initial allocation status of unallocated;
- If the allocation status is allocated, the new list item or the subobject of the new list item will have an initial allocation status of allocated; and
- If the new list item or the subobject of the new list item is an array, its bounds will be the same as those of the original list item or the subobject of the original list item.

A privatized list item may be storage-associated with other variables when the data-sharing attribute clause is encountered. Storage association may exist because of base language constructs such as **EQUIVALENCE** or **COMMON**. If *A* is a variable that is privatized by a construct and *B* is a variable that is storage-associated with *A* then:

- The contents, allocation, and association status of *B* are undefined on entry to the region;
- Any definition of *A*, or of its allocation or association status, causes the contents, allocation, and association status of *B* to become undefined; and
- Any definition of *B*, or of its allocation or association status, causes the contents, allocation, and association status of *A* to become undefined.

A privatized list item may be a selector of an **ASSOCIATE**, **SELECT RANK** or **SELECT TYPE** construct. If the construct association is established prior to a **parallel** region, the association between the associate name and the original list item will be retained in the region.

The dynamic type of a privatized list item of a polymorphic type is the declared type.

Finalization of a list item of a finalizable type or subobjects of a list item of a finalizable type occurs at the end of the region. The order in which any final subroutines for different variables of a finalizable type are called is unspecified.

Fortran

If a list item appears in both **firstprivate** and **lastprivate** clauses, the update required for the **lastprivate** clause occurs after all initializations for the **firstprivate** clause.

Restrictions

The following restrictions apply to any list item that is privatized unless otherwise specified for a given data-sharing attribute clause:

- If a list item is an array or array section, it must specify contiguous storage.

C++

- A variable of class type (or array thereof) that is privatized requires an accessible, unambiguous default constructor for the class type.
- A variable that is privatized must not have the `constexpr` specifier unless it is of class type with a `mutable` member. This restriction does not apply to the **`firstprivate`** clause.

C++

C / C++

- A variable that is privatized must not have a `const`-qualified type unless it is of class type with a `mutable` member. This restriction does not apply to the **`firstprivate`** clause.
- A variable that is privatized must not have an incomplete type or be a reference to an incomplete type.

C / C++

Fortran

- Variables that appear in namelist statements, in variable format expressions, and in expressions for statement function definitions, must not be privatized.
- Pointers with the `INTENT(IN)` attribute must not be privatized. This restriction does not apply to the **`firstprivate`** clause.
- A private variable must not be coindexed or appear as an actual argument to a procedure where the corresponding dummy argument is a coarray.
- Assumed-size arrays must not be privatized.
- An optional dummy argument that is not present must not appear as a list item in a privatization clause or be privatized as a result of an implicitly determined data-sharing attribute or predetermined data-sharing attribute.

Fortran

7.5 Data-Sharing Attribute Clauses

Several constructs accept clauses that allow a user to control the data-sharing attributes of variables referenced in the construct. Not all of the clauses listed in this section are valid on all directives. The set of clauses that is valid on a particular directive is described with the directive. The reduction clauses are explained in Section 7.6.

A list item may be specified in both **`firstprivate`** and **`lastprivate`** clauses.

C++

If a variable referenced in a data-sharing attribute clause has a type derived from a template and the OpenMP program does not otherwise reference that variable, any behavior related to that variable is unspecified.

C++

Fortran

If individual members of a common block appear in a data-sharing attribute clause other than the **shared** clause, the variables no longer have a Fortran storage association with the common block.

Fortran

7.5.1 default Clause

Name: default	**Properties:** unique, post-modified

Arguments

Name	Type	Properties
data-sharing-attribute	Keyword: **firstprivate**, **none**, **private**, **shared**	*default*

Modifiers

Name	Modifies	Type	Properties
variable-category	*implicit-behavior*	Keyword: **aggregate**, **all**, **allocatable**, **pointer**, **scalar**	*default*
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

parallel, **target**, **target_data**, **task**, **taskloop**, **teams**

Semantics

The **default** clause determines the implicitly determined data-sharing attributes of certain variables that are referenced in the construct, in accordance with the rules given in Section 7.1.1.

The *variable-category* specifies the variables for which the attribute may be set, and the attribute is specified by *implicit-behavior*. If no *variable-category* is specified in the clause then the effect is as if **all** was specified for the *variable-category*.

C / C++

The **scalar** *variable-category* specifies non-pointer scalar variables.

C / C++

Fortran

The **scalar** *variable-category* specifies non-pointer and non-allocatable scalar variables. The **allocatable** *variable-category* specifies variables with the **ALLOCATABLE** attribute.

Fortran

The **pointer** *variable-category* specifies variables of pointer type. The **aggregate** *variable-category* specifies aggregate variables. Finally, the **all** *variable-category* specifies all variables.

If *data-sharing-attribute* is not **none**, the data-sharing attributes of the selected variables will be *data-sharing-attribute*. If *data-sharing-attribute* is **none**, the data-sharing attribute is not implicitly determined. If *data-sharing-attribute* is **shared** then the clause has no effect on a **target** construct; otherwise, its effect on a **target** construct is equivalent to specifying the **defaultmap** clause with the same *data-sharing-attribute* and *variable-category*. If both the **default** and **defaultmap** clauses are specified on a **target** construct, and their *variable-category* modifiers specify intersecting categories, the **defaultmap** clause has precedence over the **default** clause for variables of those categories.

Restrictions

Restrictions to the **default** clause are as follows:

- If *data-sharing-attribute* is **none**, each variable that is referenced in the construct and does not have a predetermined data-sharing attribute must have an explicitly determined data-sharing attribute.

C / C++

- If *data-sharing-attribute* is **firstprivate** or **private**, each variable with static storage duration that is declared in a namespace or global scope, is referenced in the construct, and does not have a predetermined data-sharing attribute must have an explicitly determined data-sharing attribute.

C / C++

Cross References

- **defaultmap** Clause, see Section 7.9.9
- **parallel** Construct, see Section 12.1
- **target** Construct, see Section 15.8
- **target_data** Construct, see Section 15.7
- **task** Construct, see Section 14.1
- **taskloop** Construct, see Section 14.2
- **teams** Construct, see Section 12.2

7.5.2 shared Clause

Name: shared	Properties: data-environment attribute, data-sharing attribute

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

parallel, **target_data**, **task**, **taskloop**, **teams**

Semantics

The **shared** clause declares one or more list items to have a shared attribute in tasks generated by the construct on which it appears. All references to a list item within a task refer to the storage area of the original list item at the point the directive was encountered.

The programmer must ensure, by adding proper synchronization, that storage shared by an explicit task region does not reach the end of its lifetime before the explicit task region completes its execution.

Fortran

The list items may include assumed-type variables and procedure pointers.

The association status of a shared pointer becomes undefined upon entry to and exit from the construct if it is associated with a target or a subobject of a target that appears as a privatized list item in a data-sharing attribute clause on the construct. A reference to the shared storage that is associated with the dummy argument by any other task must be synchronized with the reference to the procedure to avoid possible data races.

Fortran

Cross References

- **parallel** Construct, see Section 12.1
- **target_data** Construct, see Section 15.7
- **task** Construct, see Section 14.1
- **taskloop** Construct, see Section 14.2
- **teams** Construct, see Section 12.2

7.5.3 private Clause

Name: **private**	**Properties:** data-environment attribute, data-sharing attribute, innermost-leaf, privatization

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

distribute, **do**, **for**, **loop**, **parallel**, **scope**, **sections**, **simd**, **single**, **target**, **target_data**, **task**, **taskloop**, **teams**

Semantics

The **private** clause specifies that its list items are to be privatized list item according to Section 7.4. Each task or SIMD lane that references a list item in the construct receives only one new list item, unless the construct has one or more affected loops and an **order** clause that specifies **concurrent** is also present. Each new list item is a private-only variable, unless otherwise specified.

Fortran

The list items may include procedure pointers.

Fortran

Restrictions

Restrictions to the **private** clause are as specified in Section 7.4.

Cross References

- **distribute** Construct, see Section 13.7
- **do** Construct, see Section 13.6.2
- **for** Construct, see Section 13.6.1
- List Item Privatization, see Section 7.4
- **loop** Construct, see Section 13.8
- **parallel** Construct, see Section 12.1
- **scope** Construct, see Section 13.2
- **sections** Construct, see Section 13.3
- **simd** Construct, see Section 12.4
- **single** Construct, see Section 13.1
- **target** Construct, see Section 15.8
- **target_data** Construct, see Section 15.7
- **task** Construct, see Section 14.1
- **taskloop** Construct, see Section 14.2
- **teams** Construct, see Section 12.2

7.5.4 `firstprivate` Clause

Name: `firstprivate`	**Properties:** data-environment attribute, data-sharing attribute, privatization

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
saved	*list*	Keyword: **`saved`**	*default*
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`distribute`, **`do`**, **`for`**, **`parallel`**, **`scope`**, **`sections`**, **`single`**, **`target`**, **`target_data`**, **`task`**, **`taskloop`**, **`teams`**

Semantics

The **`firstprivate`** clause provides a superset of the functionality provided by the **`private`** clause. A list item that appears in a **`firstprivate`** clause is subject to the **`private`** clause semantics describled in Section 7.5.3, except as noted. In addition, the new list item has the firstprivate attribute and is initialized from the original list item. The initialization of the new list item is done once for each task that references the list item in any statement in the construct. The initialization is done prior to the execution of the construct.

For a **`firstprivate`** clause on a construct that is not a work-distribution construct, the initial value of the new list item is the value of the original list item that exists immediately prior to the construct in the task region where the construct is encountered unless otherwise specified. For a **`firstprivate`** clause on a work-distribution construct, the initial value of the new list item for each implicit task of the threads that execute the construct is the value of the original list item that exists in the implicit task immediately prior to the point in time that the construct is encountered unless otherwise specified.

To avoid data races, concurrent updates of the original list item must be synchronized with the read of the original list item that occurs as a result of the **`firstprivate`** clause.

C / C++

For variables of non-array type, the initialization occurs by copy assignment. For an array of elements of non-array type, each element is initialized as if by assignment from an element of the original array to the corresponding element of the new array.

C / C++

C++

For each variable of class type:

- If the **firstprivate** clause is not on a **target** construct then a copy constructor is invoked to perform the initialization; and
- If the **firstprivate** clause is on a **target** construct then how many copy constructors, if any, are invoked is unspecified.

If copy constructors are called, the order in which copy constructors for different variables of class type are called is unspecified.

C++

Fortran

If the **firstprivate** clause is on a **target** construct and a variable is of polymorphic type, the behavior is unspecified.

If an original list item does not have the **POINTER** attribute, initialization of the new list items occurs as if by intrinsic assignment unless the original list item has a compatible type-bound defined assignment, in which case initialization of the new list items occurs as if by the defined assignment. If an original list item that does not have the **POINTER** attribute has an allocation status of unallocated, the new list items will have the same status.

If an original list item has the **POINTER** attribute, the new list items receive the same association status as the original list item, as if by pointer assignment.

The list items may include named constants and procedure pointers.

Fortran

Restrictions

Restrictions to the **firstprivate** clause are as follows:

- A list item that is private within a **parallel** region must not appear in a **firstprivate** clause on a worksharing construct if any of the worksharing regions that arise from the worksharing construct ever bind to any of the **parallel** regions that arise from the **parallel** construct.
- A list item that is private within a **teams** region must not appear in a **firstprivate** clause on a **distribute** construct if any of the **distribute** regions that arise from the **distribute** construct ever bind to any of the **teams** regions that arise from the **teams** construct.
- A list item that appears in a **reduction** clause on a **parallel** construct must not appear in a **firstprivate** clause on a **task** or **taskloop** construct if any of the **task** regions that arise from the **task** or **taskloop** construct ever bind to any of the **parallel** regions that arise from the **parallel** construct.

- A list item that appears in a **reduction** clause on a worksharing construct must not appear in a **firstprivate** clause on a **task** construct encountered during execution of any of the worksharing regions that arise from the worksharing construct.

C++

- A variable of class type (or array thereof) that appears in a **firstprivate** clause requires an accessible, unambiguous copy constructor for the class type.
- If the original list item in a **firstprivate** clause on a work-distribution construct has a reference type then it must bind to the same object for all threads in the binding thread set of the work-distribution region.

C++

Cross References

- **distribute** Construct, see Section 13.7
- **do** Construct, see Section 13.6.2
- **for** Construct, see Section 13.6.1
- **parallel** Construct, see Section 12.1
- **private** Clause, see Section 7.5.3
- **scope** Construct, see Section 13.2
- **sections** Construct, see Section 13.3
- **single** Construct, see Section 13.1
- **target** Construct, see Section 15.8
- **target_data** Construct, see Section 15.7
- **task** Construct, see Section 14.1
- **taskloop** Construct, see Section 14.2
- **teams** Construct, see Section 12.2

7.5.5 lastprivate Clause

Name: **lastprivate**	**Properties:** data-environment attribute, data-sharing attribute, original list-item updating, privatization

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
lastprivate-modifier	*list*	Keyword: **`conditional`**	*default*
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`distribute`, **`do`**, **`for`**, **`loop`**, **`sections`**, **`simd`**, **`taskloop`**

Semantics

The **`lastprivate`** clause provides a superset of the functionality provided by the **`private`** clause. A list item that appears in a **`lastprivate`** clause is subject to the **`private`** clause semantics described in Section 7.5.3. In addition, each new list item has the lastprivate attribute. Further, when a **`lastprivate`** clause without the **`conditional`** modifier appears on a directive and the list item is not a loop-iteration variable of any affected loop, the value of each new list item from the sequentially last iteration of the affected loops, or the lexically last structured block sequence associated with a **`sections`** construct, is assigned to the original list item. Alternatively, when the **`conditional`** modifier appears on the clause or the list item is a loop-iteration variable of one of the affected loops, if execution of the canonical loop nest, when it is not associated with a directive, would assign a value to the list item then the original list item is assigned that value.

C++

For class types, the copy assignment operator is invoked. The order in which copy assignment operators for different variables of the same class type are invoked is unspecified.

C++

C / C++

For an array of elements of non-array type, each element is assigned to the corresponding element of the original array.

C / C++

Fortran

If the original list item does not have the **`POINTER`** attribute, its update occurs as if by intrinsic assignment unless it has a type bound procedure as a defined assignment.

If the original list item has the **`POINTER`** attribute, its update occurs as if by pointer assignment.

Fortran

When the **`conditional`** modifier does not appear on the **`lastprivate`** clause, any list item that is not a loop-iteration variable of the affected loops and that is not assigned a value by the sequentially last iteration of the loops, or by the lexically last structured block sequence associated with a **`sections`** construct, has an unspecified value after the construct. When the **`conditional`** modifier does not appear on the **`lastprivate`** clause, a list item that is the loop-iteration variable of an affected loop has an unspecified value after the construct if it would not be assigned a value during execution of the canonical loop nest when the loop nest is not associated with a directive. Unassigned subcomponents also have unspecified values after the construct.

If the **lastprivate** clause is used on a construct to which neither the **nowait** nor the **nogroup** clauses are applied, the original list item becomes defined at the end of the construct. Otherwise, if the **lastprivate** clause is used on a construct to which the **nowait** or the **nogroup** clauses are applied, accesses to the original list item may create a data race so if an assignment to the original list item occurs then other synchronization must ensure that the assignment completes and the original list item is flushed to memory. In either case, to avoid data races, concurrent reads or updates of the original list item must be synchronized with any update of the original list item that occurs as a result of the **lastprivate** clause.

If a list item that appears in a **lastprivate** clause with the **conditional** modifier is modified in the region by an assignment outside the construct or by an assignment that does not lexically assign to the list item then the value assigned to the original list item is unspecified.

Restrictions

Restrictions to the **lastprivate** clause are as follows:

- A list item must not appear in a **lastprivate** clause on a work-distribution construct if the corresponding region binds to the region of a parallelism-generating construct in which the list item is private.
- A list item that appears in a **lastprivate** clause with the **conditional** modifier must be a scalar variable.

C++

- A variable of class type (or array thereof) that appears in a **lastprivate** clause requires an accessible, unambiguous default constructor for the class type, unless the list item is also specified in a **firstprivate** clause.
- A variable of class type (or array thereof) that appears in a **lastprivate** clause requires an accessible, unambiguous copy assignment operator for the class type.
- If an original list item in a **lastprivate** clause on a work-distribution construct has a reference type then it must bind to the same object for all threads in the binding thread set of the work-distribution region.

C++

Fortran

- A variable that appears in a **lastprivate** clause must be definable.
- If the original list item has the **ALLOCATABLE** attribute, the corresponding list item of which the value is assigned to the original list item must have an allocation status of allocated upon exit from the sequentially last iteration of the affected loops or lexically last structured block sequence associated with a **sections** construct.
- If the list item is a polymorphic variable with the **ALLOCATABLE** attribute, the behavior is unspecified.

Fortran

Cross References

- **distribute** Construct, see Section 13.7
- **do** Construct, see Section 13.6.2
- **for** Construct, see Section 13.6.1
- **loop** Construct, see Section 13.8
- **private** Clause, see Section 7.5.3
- **sections** Construct, see Section 13.3
- **simd** Construct, see Section 12.4
- **taskloop** Construct, see Section 14.2

7.5.6 linear Clause

Name: **linear**	**Properties:** data-environment attribute, data-sharing attribute, privatization, innermost-leaf, post-modified

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
step-simple-modifier	*list*	OpenMP integer expression	exclusive, region-invariant, unique
step-complex-modifier	*list*	Complex, name: **step** Arguments: ***linear-step*** expression of integer type (region-invariant)	unique
linear-modifier	*list*	Keyword: **ref**, **uval**, **val**	unique
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

declare_simd, **do**, **for**, **simd**

Semantics

The **linear** clause provides a superset of the functionality provided by the **private** clause. A list item that appears in a **linear** clause is subject to the **private** clause semantics described in Section 7.5.3, except as noted. Additionally, each new list item has the linear attribute and so is a linear variable. If the *step-simple-modifier* is specified, the behavior is as if the *step-complex-modifier* is instead specified with *step-simple-modifier* as its *linear-step* argument. If *linear-step* is not specified, it is assumed to be one.

When a **linear** clause is specified on a loop-collapsing construct and a list item is the loop-iteration variable of an affected loop, the effect is as if that list item had appeared in a **lastprivate** clause. Otherwise, when a **linear** clause is specified on a loop-collapsing construct, the value of the new list item on each collapsed iteration corresponds to the value of the original list item before entering the construct plus the logical number of the iteration times *linear-step*. The value that corresponds to the sequentially last collapsed iteration of the collapsed loops is assigned to the original list item.

When a **linear** clause is specified on a **declare_simd** directive, the list items refer to parameters of the procedure to which the directive applies. For a given call to the procedure, the clause determines whether the SIMD version generated by the directive may be called. If the clause does not specify the **ref** *linear-modifier*, the SIMD version requires that the value of the corresponding argument at the callsite is equal to the value of the argument from the first lane plus the logical number of the SIMD lane times the *linear-step*. If the clause specifies the **ref** *linear-modifier*, the SIMD version requires that the storage locations of the corresponding arguments at the callsite from each SIMD lane correspond to storage locations within a hypothetical array of elements of the same type, indexed by the logical number of the SIMD lane times the *linear-step*.

Restrictions

Restrictions to the **linear** clause are as follows:

- If a **reduction** clause with the **inscan** modifier also appears on the construct, only loop-iteration variables of affected loops may appear as list items in a **linear** clause.
- A *linear-modifier* may be specified as **ref** or **uval** only for **linear** clauses on **declare_simd** directives.
- For a **linear** clause that appears on a loop-nest-associated directive, the difference between the value of a list item at the end of a collapsed iteration and its value at the beginning of the collapsed iteration must be equal to *linear-step*.
- If *linear-modifier* is **uval** for a list item in a **linear** clause that is specified on a **declare_simd** directive and the list item is modified during a call to the SIMD version of the procedure, the OpenMP program must not depend on the value of the list item upon return from the procedure.
- If *linear-modifier* is **uval** for a list item in a **linear** clause that is specified on a **declare_simd** directive, the OpenMP program must not depend on the storage of the

argument in the procedure being the same as the storage of the corresponding argument at the callsite.

- None of the affected loops of a loop-nest-associated construct that has a **linear** clause may be a non-rectangular loop.

C

- All list items must be of integral or pointer type.
- If specified, *linear-modifier* must be **val**.

C

C++

- If *linear-modifier* is not **ref**, all list items must be of integral or pointer type, or must be a reference to an integral or pointer type.
- If *linear-modifier* is **ref** or **uval**, all list items must be of a reference type.
- If a list item in a **linear** clause on a worksharing construct has a reference type then it must bind to the same object for all threads of the team.
- If a list item in a **linear** clause that is specified on a **declare_simd** directive is of a reference type and *linear-modifier* is not **ref**, the difference between the value of the argument on exit from the function and its value on entry to the function must be the same for all SIMD lanes.

C++

Fortran

- If *linear-modifier* is not **ref**, all list items must be of type **integer**.
- If *linear-modifier* is **ref** or **uval**, all list items must be dummy arguments without the **VALUE** attribute.
- List items must not be variables that have the **POINTER** attribute.
- If *linear-modifier* is not **ref** and a list item has the **ALLOCATABLE** attribute, the allocation status of the list item in the last collapsed iteration must be allocated upon exit from that collapsed iteration.
- If *linear-modifier* is **ref**, list items must be polymorphic variables, assumed-shape arrays, or variables with the **ALLOCATABLE** attribute.
- If a list item in a **linear** clause that is specified on a **declare_simd** directive is a dummy argument without the **VALUE** attribute and *linear-modifier* is not **ref**, the difference between the value of the argument on exit from the procedure and its value on entry to the procedure must be the same for all SIMD lanes.
- A common block name must not be a list item in a **linear** clause.

Fortran

Cross References

- **declare_simd** Directive, see Section 9.8
- **do** Construct, see Section 13.6.2
- **for** Construct, see Section 13.6.1
- **private** Clause, see Section 7.5.3
- **simd** Construct, see Section 12.4
- **taskloop** Construct, see Section 14.2

7.5.7 is_device_ptr Clause

Name: **is_device_ptr**	**Properties:** data-environment attribute, data-sharing attribute, device-associated, innermost-leaf, privatization

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

dispatch, **target**

Semantics

The **is_device_ptr** clause indicates that its list items are device pointers. Support for device pointers created outside of any OpenMP mechanism that returns a device pointer, is implementation defined.

If the **is_device_ptr** clause is specified on a **target** construct, each list item is privatized inside the construct. Each new list item has the is-device-ptr attribute and is initialized to the device address to which the original list item refers.

Restrictions

Restrictions to the **is_device_ptr** clause are as follows:

- Each list item must be a valid device pointer for the device data environment.

Cross References

- **dispatch** Construct, see Section 9.7
- **has_device_addr** Clause, see Section 7.5.9
- **target** Construct, see Section 15.8

7.5.8 use_device_ptr Clause

Name: **use_device_ptr**	**Properties:** all-data-environments, data-environment attribute, data-sharing attribute, device-associated, privatization

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

target_data

Semantics

Each list item in the **use_device_ptr** clause results in a new list item that has the use-device-ptr attribute and is a device pointer that refers to a device address. Since the **use_device_ptr** clause is an all-data-environments clause, it has this effect even for minimal data environments. The device address is determined as follows. A list item is treated as if a zero-offset assumed-size array at the storage location to which the list item points is mapped by a **map** clause on the construct with a *map-type* of **storage**. If a matched candidate is found for the assumed-size array (see Section 7.9.6), the new list item refers to the device address that is the base address of the array section that corresponds to the assumed-size array in the device data environment. Otherwise, the new list item refers to the address stored in the original list item. All references to the list item inside the structured block associated with the construct are replaced with the new list item that is a private copy in the associated data environment on the encountering device. Thus, the **use_device_ptr** clause is a privatization clause.

Restrictions

Restrictions to the **use_device_ptr** clause are as follows:

- Each list item must be a C pointer for which the value is the address of an object that has corresponding storage or is accessible on the target device.

Cross References

- **target_data** Construct, see Section 15.7

7.5.9 has_device_addr Clause

Name: has_device_addr	Properties: data-environment attribute, data-sharing attribute, device-associated, outermost-leaf

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

dispatch, **target**

Semantics

The **has_device_addr** clause indicates that its list items already have device addresses and therefore they may be directly accessed from a target device. Inside the construct, the list items have the has-device-addr attribute. The list items may include array sections. If the list item is a referencing variable, the semantics of the **has_device_addr** clause apply to its referenced pointee. When the clause appears on the **target** construct, if the device address of a list item is not for the device on which the **target** region executes, accessing the list item inside the region results in unspecified behavior.

Fortran

For a list item in a **has_device_addr** clause, the **CONTIGUOUS** attribute, storage location, storage size, array bounds, character length, association status and allocation status (as applicable) are the same inside the construct on which the clause appears as for the original list item. The result of inquiring about other list item properties inside the structured block is implementation defined. For a list item that is an array section, the array bounds and result when invoking **C_LOC** inside the structured block is the same as if the array base had been specified in the clause instead.

Fortran

Restrictions

Restrictions to the **has_device_addr** clause are as follows:

C / C++

- Each list item must have a valid device address for the device data environment.

C / C++

Fortran

- A list item must either have a valid device address for the device data environment, be an unallocated allocatable variable, or be a disassociated data pointer.
- The association status of a list item that is a pointer must not be undefined unless it is a structure component and it results from a predefined default mapper.

Fortran

Cross References

- **dispatch** Construct, see Section 9.7
- **target** Construct, see Section 15.8

7.5.10 use_device_addr Clause

Name: use_device_addr	Properties: all-data-environments, data-environment attribute, data-sharing attribute, device-associated

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

target_data

Semantics

Each list item in a **use_device_addr** clause has the use-device-addr attribute inside the construct. If the list item is present in the device data environment on entry to the construct, the list item is treated as if it is implicitly mapped by a **map** clause on the construct with a *map-type* of **storage** and all references to the list item inside the structured block associated with the construct are to the corresponding list item in the device data environment. The list items in a **use_device_addr** clause may include array sections and assumed-size arrays. Since the **use_device_addr** clause is an all-data-environments clause, it has this effect even for minimal data environments.

If the list item is a referencing variable, the semantics of the **use_device_addr** clause apply to its referenced pointee. A private copy of the referring pointer that refers to the corresponding referenced pointee is used in place of the original referring pointer in the structured block.

C / C++

If a list item is an array section that has a base pointer, all references to the base pointer inside the structured block are replaced with a new pointer that contains the base address of the corresponding list item. This conversion may be elided if no corresponding list item is present.

C / C++

Restrictions

Restrictions to the **use_device_addr** clause are as follows:

- Each list item must have a corresponding list item in the device data environment or be accessible on the target device.
- If a list item is an array section, the array base must be a base language identifier.

Cross References

- **target_data** Construct, see Section 15.7

7.6 Reduction and Induction Clauses and Directives

The reduction clauses and the **induction** clause are data-sharing attribute clauses that can be used to perform reductions and inductions in parallel. These recurrence calculations involve the repeated application of reduction operations or induction operations. Reduction clauses include reduction-scoping clauses and reduction-participating clauses. Reduction-scoping clauses define the region in which a reduction is computed. Reduction-participating clauses define the participants in the reduction. The **induction** clause can be used to express induction operations in a loop.

7.6.1 OpenMP Reduction and Induction Identifiers

The syntax of OpenMP reduction identifiers and induction identifiers is defined as follows:

C

A reduction identifier is either an *identifier* or one of the following operators: **+**, *****, **&**, **|**, **^**, **&&** or **||**.

An induction identifier is either an *identifier* or one of the following operators: **+** or *****.

C

C++

A reduction identifier is either an *id-expression* or one of the following operators: **+**, *****, **&**, **|**, **^**, **&&** or **||**.

An induction identifier is either an *id-expression* or one of the following operators: **+** or *****.

C++

Fortran

A reduction identifier is either a base language identifier, a user-defined operator, an allowed intrinsic procedure name or one of the following operators: **+**, *****, **.and.**, **.or.**, **.eqv.** or **.neqv.**. The intrinsic procedure names that are allowed as reduction identifiers are **max**, **min**, **iand**, **ior** and **ieor**.

An induction identifier is either a base language identifier, a user-defined operator, or one of the following operators: **+** or *****.

Fortran

7.6.2 OpenMP Reduction and Induction Expressions

A reduction expression is an OpenMP stylized expression that is relevant to reduction clauses. An induction expression is an OpenMP stylized expression that is relevant to the **induction** clause.

Restrictions

Restrictions to reduction expressions and induction expressions are as follows:

- The execution of a reduction expression or induction expression must not result in the execution of a construct or an OpenMP API routine.
- A declare target directive must be specified for any procedure that can be accessed through any reduction expression or induction expression that respectively corresponds to a reduction identifier or an induction identifier that is used in a **target** region.

Fortran

- Any generic identifier, defined operation, defined assignment, or specific procedure used in a reduction expression or an induction expression must be resolvable to a procedure with an explicit interface that has only scalar dummy arguments.
- Any procedure used in a reduction expression or an induction expression must not have any alternate returns appear in the argument list.
- Any procedure called in the region of a reduction expression or an induction expression must be pure and must not reference any host-associated or use-associated variables nor any variables in a common block.

Fortran

7.6.2.1 OpenMP Combiner Expressions

A combiner expression specifies how a reduction combines partial results into a single value.

Fortran

A combiner expression is an assignment statement or a subroutine name followed by an argument list.

Fortran

In the definition of a combiner expression, **omp_in** and **omp_out** are OpenMP identifiers for special variables that refer to storage of the type of the list item to which the reduction applies. If the list item is an array or array section, the OpenMP identifiers **omp_in** and **omp_out** each refer to an array element of that list item. Each of these OpenMP identifiers denotes one of the values to be combined before executing the combiner expression. The **omp_out** OpenMP identifier refers to the storage that holds the resulting combined value after executing the combiner expression. The number of times that the combiner expression is executed and the order of these executions for any reduction clause are unspecified.

Fortran

If the combiner expression is a subroutine name with an argument list, the combiner expression is evaluated by calling the subroutine with the specified argument list. If the combiner expression is an assignment statement, the combiner expression is evaluated by executing the assignment statement.

If a generic name is used in a combiner expression and the list item in the corresponding reduction clause is an array or array section, that generic name is resolved to the specific procedure that is elemental or only has scalar dummy arguments.

Fortran

Restrictions

Restrictions to combiner expressions are as follows:

- The only variables allowed in a combiner expression are **omp_in** and **omp_out**.

Fortran

- Any selectors in the designator of **omp_in** and **omp_out** must be component selectors.

Fortran

7.6.2.2 OpenMP Initializer Expressions

If the initialization of the private copies of list items in a reduction clause is not determined *a priori*, the syntax of an initializer expression is as follows:

C

```
omp_priv = initializer
```

C

or

C++

```
omp_priv initializer
```

C++

or

C / C++

```
function-name (argument-list)
```

C / C++

or

Fortran

```
omp_priv = expression
```

or

```
subroutine-name (argument-list)
```

Fortran

In the definition of an initializer expression, the **omp_priv** OpenMP identifier represents a special variable that refers to the storage to be initialized. The OpenMP identifier **omp_orig** represents a special variable that can be used in an initializer expression to refer to the storage of the original list item to be reduced. The number of times that an initializer expression is evaluated and the order of these evaluations are unspecified.

C / C++

If an initializer expression is a function name with an argument list, it is evaluated by calling the function with the specified argument list. Otherwise, an initializer expression specifies how **omp_priv** is declared and initialized.

C / C++

Fortran

If an initializer expression is a subroutine name with an argument list, it is evaluated by calling the subroutine with the specified argument list. If an initializer expression is an assignment statement, the initializer expression is evaluated by executing the assignment statement.

Fortran

C

The *a priori* initialization of private copies that are created for reductions follows the rules for initialization of objects with static storage duration.

C

C++

The *a priori* initialization of private copies that are created for reductions follows the base language rules for default initialization.

C++

Fortran

The rules for *a priori* initialization of private copies that are created for reductions are as follows:

- For **complex**, **real**, or **integer** types, the value 0 will be used.
- For **logical** types, the value **.false.** will be used.

- For derived types for which default initialization is specified, default initialization will be used.
- Otherwise, the behavior is unspecified.

Fortran

Restrictions

Restrictions to initializer expressions are as follows:

- The only variables allowed in an initializer expression are **omp_priv** and **omp_orig**.
- An initializer expression must not modify the variable **omp_orig**.

C

- If an initializer expression is a function name with an argument list, one of the arguments must be the address of **omp_priv**.

C

C++

- If an initializer expression is a function name with an argument list, one of the arguments must be **omp_priv** or the address of **omp_priv**.

C++

Fortran

- If an initializer expression is a subroutine name with an argument list, one of the arguments must be **omp_priv**.

Fortran

7.6.2.3 OpenMP Inductor Expressions

An inductor expression specifies an inductor, which is how an induction operation determines a new value of the induction variable from its previous value and a step expression.

Fortran

An inductor expression is either an assignment statement or a subroutine name followed by an argument list.

Fortran

In the definition of an inductor expression, the OpenMP identifier **omp_var** is a special variable that refers to storage of the type of the induction variable to which the induction operation applies, and the OpenMP identifier **omp_step** is a special variable that refers to the step expression of the induction operation. If the list item is an array or array section, the OpenMP identifier **omp_var** refers to an array element of that list item.

Fortran

If the inductor expression is a subroutine name with an argument list, the inductor expression is evaluated by calling the subroutine with the specified argument list. If the inductor expression is an assignment statement, the inductor expression is evaluated by executing the assignment statement.

If a generic name is used in an inductor expression and the list item in the corresponding **induction** clause is an array or array section, that generic name is resolved to the specific procedure that is elemental or only has scalar dummy arguments.

Fortran

Restrictions

Restrictions to inductor expressions are as follows:

- The only variables allowed in an inductor expression are **omp_var** and **omp_step**.

Fortran

- Any selectors in the designator of **omp_var** and **omp_step** must be component selectors.

Fortran

7.6.2.4 OpenMP Collector Expressions

A collector expression evaluates to the value of the collective step expression of a collapsed iteration. In the definition of a collector expression, the OpenMP identifier **omp_step** is a special variable that refers to the step expression and the OpenMP identifier **omp_idx** is a special variable that refers to the collapsed iteration number.

Restrictions

Restrictions to collector expressions are as follows:

- The only variables allowed in a collector expression are **omp_step** and **omp_idx**.

7.6.3 Implicitly Declared OpenMP Reduction Identifiers

C / C++

Table 7.1 lists each reduction identifier that is implicitly declared at every scope and its semantic initializer expression. The actual initializer value is that value as expressed in the data type of the reduction list item if that list item is an arithmetic type. In C++, list items of class type are assigned or constructed with an integral value that matches the initializer value as specified in Section 7.6.6.

TABLE 7.1: Implicitly Declared C/C++ Reduction Identifiers

Identifier	Initializer	Combiner
+	**omp_priv = 0**	**omp_out += omp_in**
*****	**omp_priv = 1**	**omp_out *= omp_in**
&	**omp_priv = ~ 0**	**omp_out &= omp_in**
|	**omp_priv = 0**	**omp_out |= omp_in**

table continued on next page

table continued from previous page

Identifier	Initializer	Combiner
^	**omp_priv = 0**	**omp_out ^= omp_in**
&&	**omp_priv = 1**	**omp_out = omp_in && omp_out**
||	**omp_priv = 0**	**omp_out = omp_in || omp_out**
max	**omp_priv =** *Minimal representable number in the reduction list item type*	**omp_out = omp_in > omp_out ? omp_in : omp_out**
min	**omp_priv =** *Maximal representable number in the reduction list item type*	**omp_out = omp_in < omp_out ? omp_in : omp_out**

C / C++

Fortran

Table 7.2 lists each reduction identifier that is implicitly declared for numeric and logical types and its semantic initializer value. The actual initializer value is that value as expressed in the data type of the reduction list item.

TABLE 7.2: Implicitly Declared Fortran Reduction Identifiers

Identifier	Initializer	Combiner
+	**omp_priv = 0**	**omp_out = omp_in + omp_out**
*****	**omp_priv = 1**	**omp_out = omp_in * omp_out**
.and.	**omp_priv = .true.**	**omp_out = omp_in .and. omp_out**
.or.	**omp_priv = .false.**	**omp_out = omp_in .or. omp_out**
.eqv.	**omp_priv = .true.**	**omp_out = omp_in .eqv. omp_out**
.neqv.	**omp_priv = .false.**	**omp_out = omp_in .neqv. omp_out**
max	**omp_priv =** *Minimal representable number in the reduction list item type*	**omp_out = max(omp_in, omp_out)**
min	**omp_priv =** *Maximal representable number in the reduction list item type*	**omp_out = min(omp_in, omp_out)**

table continued on next page

table continued from previous page

Identifier	Initializer	Combiner
`iand`	`omp_priv =` *All bits on*	`omp_out = iand(omp_in, omp_out)`
`ior`	`omp_priv = 0`	`omp_out = ior(omp_in, omp_out)`
`ieor`	`omp_priv = 0`	`omp_out = ieor(omp_in, omp_out)`

Fortran

7.6.4 Implicitly Declared OpenMP Induction Identifiers

C / C++

Table 7.3 lists each induction identifier that is implicitly declared at every scope for arithmetic types and its corresponding inductor expression and collector expression.

TABLE 7.3: Implicitly Declared C/C++ Induction Identifiers

Identifier	Inductor Expression	Collector Expression
`+`	`omp_var = omp_var + omp_step`	`omp_step * omp_idx`
`*`	`omp_var = omp_var * omp_step`	`pow(omp_step, omp_idx)`

C / C++

Fortran

Table 7.4 lists each induction identifier that is implicitly declared for numeric types and its corresponding inductor expression and collector expression.

TABLE 7.4: Implicitly Declared Fortran Induction Identifiers

Identifier	Inductor Expression	Collector Expression
`+`	`omp_var = omp_var + omp_step`	`omp_step * omp_idx`
`*`	`omp_var = omp_var * omp_step`	`omp_step ** omp_idx`

Fortran

7.6.5 Properties Common to Reduction and `induction` Clauses

The list items that appear in a reduction clause or an **`induction`** clause may include array sections and array elements.

C++

If the type is a derived class then any reduction identifier or induction identifier that matches its base classes is also a match if no specific match for the type has been specified.

If the reduction identifier or induction identifier is an implicitly declared reduction identifier or induction identifier or otherwise not an *id-expression* then it is implicitly converted to one by prepending the keyword operator (for example, **+** becomes *operator***+**). This conversion is valid for the **+**, *****, **/**, **&&** and **||** operators.

If the reduction identifier or induction identifier is qualified then a qualified name lookup is used to find the declaration.

If the reduction identifier or induction identifier is unqualified then an argument-dependent name lookup must be performed using the type of each list item.

C++

If a list item is an array or array section, it will be treated as if a reduction clause or an **`induction`** clause would be applied to each separate element of the array or array section.

If a list item is an array section, the elements of any copy of the array section will be stored contiguously.

Fortran

If the original list item has the **`POINTER`** attribute, any copies of the list item are associated with private targets.

Fortran

Restrictions

Restrictions common to reduction clauses and **`induction`** clauses are as follows:

- Any array element must be specified at most once in all list items on a directive.
- For a reduction identifier or an induction identifier declared in a **`declare_reduction`** or a **`declare_induction`** directive, the directive must appear before its use in a reduction clause or **`induction`** clause.
- If a list item is an array section, it must not be a zero-length array section and its array base must be a base language identifier.
- If a list item is an array section or an array element, accesses to the elements of the array outside the specified array section or array element result in unspecified behavior.

C / C++

- The type of a list item that appears in a reduction clause must be valid for the reduction identifier. The type of a list item and of the step expression that appear in an **induction** clause must be valid for the induction identifier.
- A list item that appears in a reduction clause or an **induction** clause must not be **const**-qualified.
- The reduction identifier or induction identifier for any list item must be unambiguous and accessible.

C / C++

Fortran

- The type, type parameters and rank of a list item that appears in a reduction clause must be valid for the combiner expression and the initializer expression. The type, type parameters and rank of a list item and of the step expression that appear in an **induction** clause must be valid for the inductor expression.
- A list item that appears in a reduction clause or an **induction** clause must be definable.
- A procedure pointer must not appear in a reduction clause or an **induction** clause.
- A pointer with the **INTENT(IN)** attribute must not appear in a reduction clause or an **induction** clause.
- An original list item with the **POINTER** attribute or any pointer component of an original list item that is referenced in a combiner expression or an inductor expression must be associated at entry with the construct that contains the reduction clause or **induction** clause. Additionally, the list item or the pointer component of the list item must not be deallocated, allocated, or pointer assigned within the region.
- An original list item with the **ALLOCATABLE** attribute or any allocatable component of an original list item that corresponds to a special variable identifier in a combiner expression, initializer expression, or inductor expression must be in the allocated state at entry to the construct that contains the reduction clause or **induction** clause. Additionally, the list item or the allocatable component of the list item must be neither deallocated nor allocated, explicitly or implicitly, within the region.
- If the reduction identifier or induction identifier is defined in a **declare_reduction** or **declare_induction** directive, that directive must be in the same subprogram, or accessible by host or use association.
- If the reduction identifier or induction identifier is a user-defined operator, the same explicit interface for that operator must be accessible at the location of the **declare_reduction** or **declare_induction** directive that defines the reduction or induction identifier.

- If the reduction identifier or induction identifier is defined in a **declare_reduction** or **declare_induction** directive, any procedure referenced in the **initializer**, **combiner**, **inductor**, or **collector** clause must be an intrinsic function, or must have an explicit interface where the same explicit interface is accessible as at the **declare_reduction** or **declare_induction** directive.

Fortran

7.6.6 Properties Common to All Reduction Clauses

The *clause-specification* of a reduction clause has a *clause-argument-specification* that specifies a variable list and has a required *reduction-identifier* modifier that specifies the reduction identifier to use for the list items. This match is done by means of a name lookup in the base language.

C++

If the type is of class type and the reduction identifier is implicitly declared, then it must provide the operator as described in Section 7.6.5 as well as one of:

- A default constructor and an assignment operator that accepts a type *T* that can be implicitly constructed from an integer expression, such that the following requirement is valid:

```
template<typename T>
requires(T&& t) {
      T();
      t = 0;
};
```

- A single-argument constructor that accepts a type *T* that can be implicitly constructed from an integer expression, such that the following requirement is valid:

```
template<typename T>
requires() {
      T(0);
};
```

The first of these that matches will be used, with the initializer value being passed to the assignment operator or constructor.

C++

Any copies of a list item associated with the reduction have the reduction attribute and so are reduction variables. These reduction variables are initialized with the initializer value of the reduction identifier. Any copies are combined using the combiner associated with the reduction identifier.

Execution Model Events

The *reduction-begin* event occurs before a task begins to perform loads and stores that belong to the implementation of a reduction and the *reduction-end* event occurs after the task has completed loads and stores associated with the reduction. If a task participates in multiple reductions, each reduction may be bracketed by its own pair of *reduction-begin*/*reduction-end* events or multiple reductions may be bracketed by a single pair of events. The interval defined by a pair of *reduction-begin*/*reduction-end* events will not contain a task scheduling point.

Tool Callbacks

A thread dispatches a registered **reduction** callback with **ompt_sync_region_reduction** in its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *reduction-begin* event in that thread. Similarly, a thread dispatches a registered **reduction** callback with **ompt_sync_region_reduction** in its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of a *reduction-end* event in that thread. These callbacks occur in the context of the task that performs the reduction.

Restrictions

Restrictions common to reduction clauses are as follows:

C

- For a **max** or **min** reduction, the type of the list item must be an allowed arithmetic data type: **char**, **int**, **float**, **double**, or **_Bool**, possibly modified with **long**, **short**, **signed**, or **unsigned**.

C

C++

- For a **max** or **min** reduction, the type of the list item must be an allowed arithmetic data type: **char**, **wchar_t**, **int**, **float**, **double**, or **bool**, possibly modified with **long**, **short**, **signed**, or **unsigned**.

C++

Cross References

- **reduction** Callback, see Section 34.7.6
- OMPT **scope_endpoint** Type, see Section 33.27
- OMPT **sync_region** Type, see Section 33.33

7.6.7 Reduction Scoping Clauses

Reduction-scoping clauses define the region in which a reduction is computed by tasks or SIMD lanes. All properties common to all reduction clauses, which are defined in Section 7.6.5 and Section 7.6.6, apply to reduction-scoping clauses.

The number of copies created for each list item and the point at which those copies are initialized are determined by the particular reduction-scoping clause that appears on the construct. The point at which the original list item contains the result of the reduction is determined by the particular reduction-scoping clause. To avoid data races, concurrent reads or updates of the original list item must be synchronized with the update of the original list item that occurs as a result of the reduction, which may occur after execution of the construct on which the reduction-scoping clause appears, for example, due to the use of a **nowait** clause.

The location in the OpenMP program at which values are combined and the order in which values are combined are unspecified. Thus, when comparing sequential and parallel executions, or when comparing one parallel execution to another (even if the number of threads used is the same), bitwise-identical results are not guaranteed. Similarly, side effects (such as floating-point exceptions) may not be identical and may not occur at the same location in the OpenMP program.

7.6.8 Reduction Participating Clauses

A reduction-participating clause specifies a task or a SIMD lane as a participant in a reduction defined by a reduction-scoping clause. All properties common to all reduction clauses, which are defined in Section 7.6.5 and Section 7.6.6, apply to reduction-participating clauses.

Accesses to the original list item may be replaced by accesses to copies of the original list item created by a region that corresponds to a construct with a reduction-scoping clause.

In any case, the final value of the reduction must be determined as if all tasks or SIMD lanes that participate in the reduction are executed sequentially in some arbitrary order.

7.6.9 *reduction-identifier* Modifier

Modifiers

Name	Modifies	Type	Properties
reduction-identifier	*all arguments*	An OpenMP reduction identifier	required, ultimate

Clauses

in_reduction, **reduction**, **task_reduction**

Semantics

Reduction clauses use the *reduction-identifier* modifier to specify the reduction identifier for the clause. The reduction identifier determines the initializer expression and combiner expression to use for the reduction.

Cross References

- OpenMP Reduction and Induction Identifiers, see Section 7.6.1
- **in_reduction** Clause, see Section 7.6.12
- **reduction** Clause, see Section 7.6.10
- **task_reduction** Clause, see Section 7.6.11

7.6.10 `reduction` Clause

Name: `reduction`	Properties: data-environment attribute, data-sharing attribute, original list-item updating, privatization, reduction scoping, reduction participating

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
reduction-identifier	*all arguments*	An OpenMP reduction identifier	required, ultimate
reduction-modifier	*list*	Keyword: **`default`**, **`inscan`**, **`task`**	*default*
original-sharing-modifier	*list*	Complex, name: **`original`** Arguments: ***sharing*** Keyword: **`default`**, **`private`**, **`shared`** (*default*)	*default*
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`do`, **`for`**, **`loop`**, **`parallel`**, **`scope`**, **`sections`**, **`simd`**, **`taskloop`**, **`teams`**

Semantics

The **`reduction`** clause is a reduction-scoping clause and a reduction-participating clause, as described in Section 7.6.7 and Section 7.6.8. For each list item, a private copy is created for each implicit task or SIMD lane and is initialized with the initializer value of the *reduction-identifier*. After the end of the region, the original list item is updated with the values of the private copies using the combiner associated with the *reduction-identifier*. If the clause appears on a worksharing construct and the original list item is private in the enclosing context of that construct, the behavior is as if a shared copy (initialized with the initializer value) specific to the worksharing region is updated by combining its value with the values of the private copies created by the clause; once an encountering thread observes that all of those updates are completed, the original list item for that thread is then updated by combining its value with the value of the shared copy.

If the *original-sharing-modifier* is not present, the behavior is as if it were present with the *sharing* argument specified as **`default`**. If the *sharing* argument is specified as **`default`**, original list items are assumed to be shared in the enclosing context unless determined not to be shared according to the rules specified in Section 7.1. If **`shared`** or **`private`** is specified as the *original-sharing-modifier* *sharing* argument, the original list items are assumed to be shared or private, respectively, in the enclosing context.

If *reduction-modifier* is not present or the **default** *reduction-modifier* is present, the behavior is as follows. For **parallel** and worksharing constructs, one or more private copies of each list item are created for each implicit task, as if the **private** clause had been used. For the **simd** construct, one or more private copies of each list item are created for each SIMD lane, as if the **private** clause had been used. For the **taskloop** construct, private copies are created according to the rules of the reduction-scoping clause. For the **teams** construct, one or more private copies of each list item are created for the initial task of each team in the league, as if the **private** clause had been used. For the **loop** construct, private copies are created and used in the construct according to the description and restrictions in Section 7.4. At the end of a region that corresponds to a construct for which the **reduction** clause was specified, the original list item is updated by combining its original value with the final value of each of the private copies, using the combiner of the specified *reduction-identifier*.

If the **inscan** *reduction-modifier* is present, a scan computation is performed over updates to the list item performed in each logical iteration of the affected loops (see Section 7.7). The list items are privatized in the construct according to the description and restrictions in Section 7.4. At the end of the region, each original list item is assigned the value described in Section 7.7.

If the **task** *reduction-modifier* is present for a **parallel** or worksharing construct, then each list item is privatized according to the description and restrictions in Section 7.4, and an unspecified number of additional private copies may be created to support task reductions. Any copies associated with the reduction are initialized before they are accessed by the tasks that participate in the reduction, which include all implicit tasks in the corresponding region and all participating explicit tasks that specify an **in_reduction** clause (see Section 7.6.12). After the end of the region, the original list item contains the result of the reduction.

Restrictions

Restrictions to the **reduction** clause are as follows:

- All restrictions common to all reduction clauses, as listed in Section 7.6.5 and Section 7.6.6, apply to this clause.
- For a given construct on which the clause appears, the lifetime of all original list items must extend at least until after the synchronization point at which the completion of the corresponding region by all participants in the reduction can be observed by all participants.
- If the **inscan** *reduction-modifier* is specified on a **reduction** clause that appears on a worksharing construct and an original list item is private in the enclosing context of the construct, the private copies must all have identical values when the construct is encountered.
- If the **reduction** clause appears on a worksharing construct and the *original-sharing-modifier* specifies **default** as its *sharing* argument, each original list item must be shared in the enclosing context unless it is determined not to be shared according to the rules specified in Section 7.1.

- If the **reduction** clause appears on a worksharing construct and the *original-sharing-modifier* specifies **shared** or **private** as its *sharing* argument, the original list items must be shared or private, respectively, in the enclosing context.
- Each list item specified with the **inscan** *reduction-modifier* must appear as a list item in an **inclusive** or **exclusive** clause on a **scan** directive enclosed by the construct.
- If the **inscan** *reduction-modifier* is specified, a **reduction** clause without the **inscan** *reduction-modifier* must not appear on the same construct.
- A list item that appears in a **reduction** clause on a work-distribution construct for which the corresponding region binds to a **teams** region must be shared in the **teams** region.
- A **reduction** clause with the **task** *reduction-modifier* may only appear on a **parallel** construct or a worksharing construct, or a compound construct for which any of the aforementioned constructs is a constituent construct and neither **simd** nor **loop** are constituent constructs.
- A **reduction** clause with the **inscan** *reduction-modifier* may only appear on a worksharing-loop construct or a **simd** construct, or a compound construct for which any of the aforementioned constructs is a constituent construct and neither **distribute** nor **taskloop** is a constituent construct.
- The **inscan** *reduction-modifier* must not be specified on a construct for which the **ordered** or **schedule** clause is specified.
- A list item that appears in a **reduction** clause of the innermost enclosing worksharing construct or **parallel** construct must not be accessed in an explicit task generated by a construct unless an **in_reduction** clause with the same list item appears on that construct.
- The **task** *reduction-modifier* must not appear in a **reduction** clause if the **nowait** clause is specified on the same construct.

Fortran

- If the *original-sharing-modifier* for a **reduction** clause on a worksharing construct specifies **default** *sharing* and a list item in the clause either has a base pointer or is a dummy argument without the **VALUE** attribute, the original list item must refer to the same object for all threads of the team that execute the corresponding region.

Fortran

C / C++

- If the *original-sharing-modifier* specifies **default** as it *sharing* argument and a list item in a **reduction** clause on a worksharing construct has a reference type then that list item must bind to the same object for all threads of the team.
- A variable of class type (or array thereof) that appears in a **reduction** clause with the **inscan** *reduction-modifier* requires an accessible, unambiguous default constructor and copy assignment operator for the class type; the number of calls to them while performing the scan computation is unspecified.

C / C++

Cross References

- **do** Construct, see Section 13.6.2
- **for** Construct, see Section 13.6.1
- List Item Privatization, see Section 7.4
- **loop** Construct, see Section 13.8
- **ordered** Clause, see Section 6.4.6
- **parallel** Construct, see Section 12.1
- **private** Clause, see Section 7.5.3
- **scan** Directive, see Section 7.7
- **schedule** Clause, see Section 13.6.3
- **scope** Construct, see Section 13.2
- **sections** Construct, see Section 13.3
- **simd** Construct, see Section 12.4
- **taskloop** Construct, see Section 14.2
- **teams** Construct, see Section 12.2

7.6.11 task_reduction Clause

Name: **task_reduction**	**Properties:** data-environment attribute, data-sharing attribute, original list-item updating, privatization, reduction scoping

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
reduction-identifier	*all arguments*	An OpenMP reduction identifier	required, ultimate
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

taskgroup

Semantics

The **task_reduction** clause is a reduction-scoping clause, as described in Section 7.6.7, that specifies a task reduction. For each list item, the number of copies is unspecified. Any copies associated with the reduction are initialized before they are accessed by the tasks that participate in the reduction. After the end of the region, the original list item contains the result of the reduction.

Restrictions

Restrictions to the **task_reduction** clause are as follows:

- All restrictions common to all reduction clauses, as listed in Section 7.6.5 and Section 7.6.6, apply to this clause.

Cross References

- **taskgroup** Construct, see Section 17.4

7.6.12 `in_reduction` Clause

Name: `in_reduction`	**Properties:** data-environment attribute, data-sharing attribute, privatization, reduction participating

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
reduction-identifier	*all arguments*	An OpenMP reduction identifier	required, ultimate
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

target, **target_data**, **task**, **taskloop**

Semantics

The **in_reduction** clause is a reduction-participating clause, as described in Section 7.6.8, that specifies that a task participates in a reduction. For a given list item, the **in_reduction** clause defines a task to be a participant in a task reduction that is defined by an enclosing region for a matching list item that appears in a **task_reduction** clause or a **reduction** clause with the **task** *reduction-modifier*, where either:

1. The matching list item has the same storage location as the list item in the **in_reduction** clause; or
2. A private copy, derived from the matching list item, that is used to perform the task reduction has the same storage location as the list item in the **in_reduction** clause.

For the **task** construct, the generated task becomes the participating task. For each list item, a private copy may be created as if the **private** clause had been used.

For the **target** construct, the target task becomes the participating task. For each list item, a private copy may be created in the data environment of the target task as if the **private** clause had been used. This private copy will be implicitly mapped into the device data environment of the target device, if the target device is not the parent device.

At the end of the task region, if a private copy was created its value is combined with a copy created by a reduction-scoping clause or with the original list item.

When specified on the **target_data** directive, the **in_reduction** clause has the all-data-environments property.

Restrictions

Restrictions to the **in_reduction** clause are as follows:

- All restrictions common to all reduction clauses, as listed in Section 7.6.5 and Section 7.6.6, apply to this clause.
- For each list item, a matching list item must exist that appears in a **task_reduction** clause or a **reduction** clause with the **task** *reduction-modifier* that is specified on a construct that corresponds to a region in which the region of the participating task is closely nested. The construct that corresponds to the innermost enclosing region that meets this condition must specify the same *reduction-identifier* for the matching list item as the **in_reduction** clause.

Cross References

- **target** Construct, see Section 15.8
- **target_data** Construct, see Section 15.7
- **task** Construct, see Section 14.1
- **taskloop** Construct, see Section 14.2

7.6.13 induction Clause

Name: **induction**	**Properties:** data-environment attribute, data-sharing attribute, original list-item updating, privatization

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
induction-identifier	*list*	OpenMP induction identifier	required, ultimate
step-modifier	*list*	Complex, name: **step** Arguments: ***induction-step*** expression of induction-step type (region-invariant)	required
induction-modifier	*list*	Keyword: **relaxed**, **strict**	*default*
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

distribute, **do**, **for**, **simd**, **taskloop**

Semantics

The **induction** clause provides a superset of the functionality provided by the **private** clause. A list item that appears in an **induction** clause is subject to the **private** clause semantics described in Section 7.5.3, except as otherwise specified. The new list items have the induction attribute.

When an **induction** clause is specified on a loop-nest-associated directive and the **strict** *induction-modifier* is present, the value of the new list item at the beginning of each collapsed iteration is determined by the closed form of the induction operation. The value of the original list item at the end of the last collapsed iteration is the result of applying the inductor expression to the value of the new list item at the beginning of that collapsed iteration. When the **relaxed** *induction-modifier* is present, the implementation may assume that the value of the new list item at the end of the previous collapsed iteration, if executed by the same task or SIMD lane, is the value determined by the closed form of the induction operation. When an *induction-modifier* is not specified, the behavior is as if the **relaxed** *induction-modifier* is present.

The value of the new list item at the end of the last collapsed iteration is assigned to the original list item.

C++

For class types, the copy assignment operator is invoked. The order in which copy assignment operators for different variables of the same class type are invoked is unspecified.

C++

C / C++

For an array of elements of non-array type, each element is assigned to the corresponding element of the original array.

C / C++

Fortran

If the original list item does not have the **POINTER** attribute, its update occurs as if by intrinsic assignment unless it has a type bound procedure as a defined assignment.

If the original list item has the **POINTER** attribute, its update occurs as if by pointer assignment.

Fortran

If the construct is a worksharing-loop construct with the **nowait** clause present and the original list item is shared in the enclosing context, access to the original list item after the construct may create a data race. To avoid this data race, user code must insert synchronization.

The *induction-identifier* must match a previously declared induction identifier of the same name and type for each of the list items and for the *induction-step-expr*. This match is done by means of a name lookup in the base language.

Restrictions

Restrictions to the **induction** clause are as follows:

- All restrictions listed in Section 7.6.5 apply to this clause.
- The *induction-step* must not be an array or array section.
- If an array section or array element appears as a list item in an **induction** clause on a worksharing construct, all threads of the team must specify the same storage location.
- None of the affected loops of a loop-nest-associated construct that has an **induction** clause may be a non-rectangular loop.

C / C++

- If a list item in an **induction** clause on a worksharing construct has a reference type and the original list item is shared in the enclosing context then it must bind to the same object for all threads of the team.
- If a list item in an **induction** clause on a worksharing construct is an array section or an array element that has a base pointer and the original list item is shared in the enclosing context, the base pointer must point to the same variable for all threads of the team.

C / C++

Cross References

- **distribute** Construct, see Section 13.7
- **do** Construct, see Section 13.6.2
- **for** Construct, see Section 13.6.1
- List Item Privatization, see Section 7.4
- **private** Clause, see Section 7.5.3
- **simd** Construct, see Section 12.4

- **taskloop** Construct, see Section 14.2

7.6.14 declare_reduction Directive

Name: declare_reduction Category: declarative	Association: unassociated Properties: pure

Arguments

declare_reduction(*reduction-specifier***)**

Name	Type	Properties
reduction-specifier	OpenMP reduction specifier	*default*

Clauses

combiner, **initializer**

Additional information

The **declare_reduction** directive may alternatively be specified with **declare reduction** as the *directive-name*.

The syntax *reduction-identifier* **:** *typename-list* **:** *combiner-expr*, where *combiner* is an OpenMP combiner expression, may alternatively be used for *reduction-specifier*. The **combiner** clause must not be specified if this syntax is used. This syntax has been deprecated.

Semantics

The **declare_reduction** directive declares a reduction identifier that can be used in a reduction clause as a user-defined reduction. The directive argument *reduction-specifier* uses the following syntax:

reduction-identifier **:** *typename-list*

where *reduction-identifier* is a reduction identifier and *typename-list* is a type-name list.

The specified reduction identifier and type-name list identify the **declare_reduction** directive. The reduction identifier can later be used in a reduction clause that uses variables of the types specified in the type-name list. If the directive specifies several types then the behavior is as if a **declare_reduction** directive was specified for each type. The visibility and accessibility of a user-defined reduction are the same as those of a variable declared at the same location in the program.

C++

The **declare_reduction** directive can also appear at the locations in a program where a static data member could be declared. In this case, the visibility and accessibility of the declaration are the same as those of a static data member declared at the same location in the program.

C++

The enclosing context of the combiner expression specified by the **combiner** clause and of the initializer expression specified by the **initializer** clause is that of the **declare_reduction** directive. The combiner expression and the initializer expression must be correct in the base language, as if they were the body of a procedure defined at the same location in the program.

Fortran

If a type with a deferred or assumed length type parameter is specified in a **declare_reduction** directive, the reduction identifier of that directive can be used in a reduction clause with any variable of the same type and the same kind parameter, regardless of the length type parameters with which the variable is declared.

If the specified reduction identifier is the same as the name of a user-defined operator or an extended operator, or the same as a generic name that is one of the allowed intrinsic procedures, and if the operator or procedure name appears in an accessibility statement in the same module, the accessibility of the corresponding **declare_reduction** directive is determined by the accessibility attribute of the statement.

If the specified reduction identifier is the same as a generic name that is one of the allowed intrinsic procedures and is accessible, and if it has the same name as a derived type in the same module, the accessibility of the corresponding **declare_reduction** directive is determined by the accessibility of the generic name according to the base language.

Fortran

Restrictions

Restrictions to the **declare_reduction** directive are as follows:

- A reduction identifier must not be re-declared in the current scope for the same type or for a type that is compatible according to the base language rules.
- The type-name list must not declare new types.

C / C++

- A type name in a **declare_reduction** directive must not be a function type, an array type, a reference type, or a type qualified with **const**, **volatile** or **restrict**.

C / C++

Fortran

- If the length type parameter is specified for a type, it must be a constant, a colon (`:`) or an asterisk (`*`).
- If a type with a deferred or assumed length parameter is specified in a **declare_reduction** directive, no other **declare_reduction** directive with the same type, the same kind parameters and the same reduction identifier is allowed in the same scope.

Fortran

Cross References

- **combiner** Clause, see Section 7.6.15
- OpenMP Combiner Expressions, see Section 7.6.2.1
- OpenMP Initializer Expressions, see Section 7.6.2.2
- OpenMP Reduction and Induction Identifiers, see Section 7.6.1
- **initializer** Clause, see Section 7.6.16

7.6.15 combiner Clause

Name: **combiner**	**Properties:** unique, required

Arguments

Name	Type	Properties
combiner-expr	expression of combiner type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

declare_reduction

Semantics

This clause specifies *combiner-expr* as the combiner expression for a user-defined reduction.

Cross References

- **declare_reduction** Directive, see Section 7.6.14
- OpenMP Combiner Expressions, see Section 7.6.2.1

7.6.16 initializer Clause

Name: **initializer**	**Properties:** unique

Arguments

Name	Type	Properties
initializer-expr	expression of initializer type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`declare_reduction`

Semantics

This clause specifies *initializer-expr* as the initializer expression for a user-defined reduction.

Cross References

- **`declare_reduction`** Directive, see Section 7.6.14
- OpenMP Initializer Expressions, see Section 7.6.2.2

7.6.17 `declare_induction` Directive

Name: **`declare_induction`** **Category:** declarative	**Association:** unassociated **Properties:** pure

Arguments

`declare_induction(`*induction-specifier***`)`**

Name	Type	Properties
induction-specifier	OpenMP induction specifier	*default*

Clauses

`collector`, **`inductor`**

Semantics

The **`declare_induction`** directive declares an induction identifier that can be used in an **`induction`** clause as a user-defined induction. The directive argument *induction-specifier* uses the following syntax:

induction-identifier **:** *type-specifier-list*

where *type-specifier-list* is defined as follows:

type-specifier-list **:=** *type-specifier* **|** *type-specifier* **,** *type-specifier-list*

type-specifier **:=** *typename-list-item* **|** *typename-pair*

typename-pair **:=** **(** *typename-list-item* **,** *typename-list-item* **)**

and where *induction-identifier* is the specified induction identifier and *typename-list-item* is a type-name list item.

The induction identifier identifies the **declare_induction** directive. The induction identifier can be used in an **induction** clause that lists induction variables of the types specified in the *type-specifier-list*, with corresponding step expressions of the same type if the *type-specifier-list* does not specify a *typename-pair*. If the *type-specifier-list* specifies a *typename-pair* then the induction identifier can be used in an **induction** clause that lists that pair, in which case the induction variable and **omp_var** must be of the first type specified in the *typename-pair* while the corresponding step expression and **omp_step** must be of the second type in the *typename-pair*. The type of **omp_idx** is the type used for the iteration count of the collapsed iteration space of the collapsed loops of the construct on which the **induction** clause appears.

The visibility and accessibility of a user-defined induction are the same as those of a variable declared at the same location in the program.

C++

The **declare_induction** directive can also appear at the locations in a program where a static data member could be declared. In this case, the visibility and accessibility of the declaration are the same as those of a static data member declared at the same location in the program.

C++

The enclosing context of the inductor expression specified by the **inductor** clause and of the collector expression specified by the **collector** clause is that of the **declare_induction** directive. The inductor expression and the collector expression must be correct in the base language, as if they were the body of a procedure defined at the same location in the program.

Fortran

If the induction identifier is the same as the name of a user-defined operator or an extended operator, or the same as a generic name that is one of the allowed intrinsic procedures, and if the operator or procedure name appears in an accessibility statement in the same module, the accessibility of the corresponding **declare_induction** directive is determined by the accessibility attribute of the statement.

If the induction identifier is the same as a generic name that is one of the allowed intrinsic procedures and is accessible, and if it has the same name as a derived type in the same module, the accessibility of the corresponding **declare_induction** directive is determined by the accessibility of the generic name according to the base language.

Fortran

Restrictions

Restrictions to the **declare_induction** directive are as follows:

- An induction identifier must not be re-declared in the current scope for the same type or for a type that is compatible according to the base language rules.
- A type-name list item in the *type-specifier-list* must not declare a new type.

C / C++

- A type name in a **declare_induction** directive must not be a function type, an array type, a reference type, or a type qualified with **const**, **volatile** or **restrict**.

C / C++

Fortran

- A type name in a **declare_induction** directive must not be an enum type or an enumeration type.

Fortran

Cross References

- **collector** Clause, see Section 7.6.19
- OpenMP Collector Expressions, see Section 7.6.2.4
- OpenMP Inductor Expressions, see Section 7.6.2.3
- OpenMP Loop-Iteration Spaces and Vectors, see Section 6.4.3
- OpenMP Reduction and Induction Identifiers, see Section 7.6.1
- **inductor** Clause, see Section 7.6.18

7.6.18 inductor Clause

Name: inductor	Properties: unique, required

Arguments

Name	Type	Properties
inductor-expr	expression of inductor type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

declare_induction

Semantics

This clause specifies *inductor-expr* as the inductor expression for a user-defined induction.

Cross References

- **`declare_induction`** Directive, see Section 7.6.17
- OpenMP Inductor Expressions, see Section 7.6.2.3

7.6.19 `collector` Clause

Name: `collector`	Properties: unique, required

Arguments

Name	Type	Properties
collector-expr	expression of collector type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`declare_induction`

Semantics

This clause specifies *collector-expr* as the collector expression for a user-defined induction, which ensures that a collector is available for use in the closed form of the induction operation.

Cross References

- **`declare_induction`** Directive, see Section 7.6.17
- OpenMP Collector Expressions, see Section 7.6.2.4

7.7 `scan` Directive

Name: `scan`	Association: separating
Category: subsidiary	Properties: pure

Separated directives

`do`, **`for`**, **`simd`**

Clauses

`exclusive`, **`inclusive`**, **`init_complete`**

Clause set

Properties: unique, required, exclusive	Members: **`exclusive`**, **`inclusive`**, **`init_complete`**

Semantics

The **scan** directive is a subsidiary directive that separates the *final-loop-body* of an enclosing **simd** construct or worksharing-loop construct (or a composite construct that combines them) into structured block sequences that represent different phases of a scan computation. The use of **scan** directives results in a structured block sequence that serves as an input phase, a structured block sequence that serves as a scan phase, and, optionally, a structured block sequence that serves as an initialization phase. The optional initialization phase begins the collapsed iteration by initializing private variables that can be used in the input phase, the input phase contains all computations that update the list item in the collapsed iteration, and the scan phase ensures that any statement that reads the list item uses the result of the scan computation for that collapsed iteration. Thus, the **scan** directive specifies that a scan computation updates each list item on each collapsed iteration of the enclosing canonical loop nest that is associated with the separated construct.

The clause that is specified on the **scan** directive determines the phases of the scan computation that correspond to the structured block sequences that precede and follow the directive.

The result of a scan computation for a given collapsed iteration is calculated according to the last generalized prefix sum ($\texttt{PRESUM}_{\texttt{last}}$) applied over the sequence of values given by the value of the original list item prior to the affected loops and all preceding updates to the new list item in the collapsed iteration space. The operation $\texttt{PRESUM}_{\texttt{last}}(op, a_1, \ldots, a_N)$ is defined for a given binary operator *op* and a sequence of N values $a_1, \ldots, a_N$ as follows:

- if $N = 1$, a_1
- if $N > 1$, $op(\ \texttt{PRESUM}_{\texttt{last}}(op, a_1, \ldots, a_j), \texttt{PRESUM}_{\texttt{last}}(op, a_k, \ldots, a_N)\ )$, $1 \le j + 1 = k \le N$.

At the beginning of the input phase of each collapsed iteration, the new list item is either initialized with the value of the initializer expression of the *reduction-identifier* specified by the **reduction** clause on the separated construct or with the value of the list item in the scan phase of some collapsed iteration. The update value of a new list item is, for a given collapsed iteration, the value the new list item would have on completion of its input phase if it were initialized with the value of the initializer expression.

Let *orig-val* be the value of the original list item on entry to the separated construct. Let *combiner* be the combiner expression for the *reduction-identifier* specified by the **reduction** clause on the construct. Let u_i be the update value of a list item for collapsed iteration i. For list items that appear in an **inclusive** clause on the **scan** directive, at the beginning of the scan phase for collapsed iteration i the new list item is assigned the result of the operation $\texttt{PRESUM}_{\texttt{last}}(\ combiner, orig\text{-}val, u_0, \ldots, u_i)$. For list items that appear in an **exclusive** clause on the **scan** directive, at the beginning of the scan phase for collapsed iteration $i = 0$ the list item is assigned the value *orig-val*, and at the beginning of the scan phase for collapsed iteration $i > 0$ the list item is assigned the result of the operation $\texttt{PRESUM}_{\texttt{last}}(\ combiner, orig\text{-}val, u_0, \ldots, u_{i-1})$.

For list items that appear in an **inclusive** clause, at the end of the separated construct, the original list item is assigned the value of the private copy from the last collapsed iteration of the affected loops of the separated construct. For list items that appear in an **exclusive** clause, let k

be the last collapsed iteration of the affected loops of the separated construct. At the end of the separated construct, the original list item is assigned the result of the operation $\text{PRESUM}_{\text{last}}($ *combiner*, *orig-val*, $u_0, \ldots, u_k)$.

Restrictions

Restrictions to the **scan** directive are as follows:

- The separated construct must have at most one **scan** directive with an **inclusive** or **exclusive** clause as a separating directive.
- The separated construct must have at most one **scan** directive with an **init_complete** clause as a separating directive.
- If specified, a **scan** directive with an **init_complete** clause must precede a **scan** directive with an **exclusive** clause that is a subsidiary directive of the same construct.
- The affected loops of the separated construct must all be perfectly nested loops.
- Each list item that appears in the **inclusive** or **exclusive** clause must appear in a **reduction** clause with the **inscan** modifier on the separated construct.
- Each list item that appears in a **reduction** clause with the **inscan** modifier on the separated construct must appear in a clause on the **scan** separating directive.
- Cross-iteration dependences across different collapsed iterations of the separated construct must not exist, except for dependences for the list items specified in an **inclusive** or **exclusive** clause.
- Intra-iteration dependences from a statement in the structured block sequence that immediately precedes a **scan** directive with an **inclusive** or **exclusive** clause to a statement in the structured block sequence that follows that **scan** directive must not exist, except for dependences for the list items specified in that clause.
- The private copy of a list item that appears in the **inclusive** or **exclusive** clause must not be modified in the scan phase.
- Any list item that appears in an **exclusive** clause must not be modified or used in the initialization phase.
- Statements in the initialization phase must only modify private variables. Any private variables modified in the initialization phase must not be used in the scan phase.

Cross References

- **do** Construct, see Section 13.6.2
- **exclusive** Clause, see Section 7.7.2
- **for** Construct, see Section 13.6.1
- **inclusive** Clause, see Section 7.7.1

- **init_complete** Clause, see Section 7.7.3
- **reduction** Clause, see Section 7.6.10
- **simd** Construct, see Section 12.4

7.7.1 inclusive Clause

Name: inclusive	**Properties:** innermost-leaf, unique

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

scan

Semantics

The **inclusive** clause is used on a **scan** directive to specify that an inclusive scan computation is performed for each list item of the argument list. The structured block sequence that precedes the directive serves as the input phase of the inclusive scan computation while the structured block sequence that follows the directive serves as the scan phase of the inclusive scan computation. The list items that appear in an **inclusive** clause may include array sections and array elements.

Cross References

- **scan** Directive, see Section 7.7

7.7.2 exclusive Clause

Name: exclusive	**Properties:** innermost-leaf, unique

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

scan

Semantics

The **exclusive** clause is used on a **scan** directive to specify an exclusive scan computation is performed for each list item of the argument list. The structured block sequence that follows the directive serves as the input phase of the exclusive scan computation while the structured block sequence that precedes the directive serves as the scan phase of the exclusive scan computation. The list items that appear in an **exclusive** clause may include array sections and array elements.

Cross References

- **scan** Directive, see Section 7.7

7.7.3 init_complete Clause

Name: init_complete	Properties: innermost-leaf, unique

Arguments

Name	Type	Properties
create_init_phase	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

scan

Semantics

The **init_complete** clause is used on a **scan** directive to demarcate the end of the initialization phase of an exclusive scan computation. The structured block sequence that precedes the directive serves as the initialization phase of the exclusive scan computation while the structured block sequence that follows the directive serves as the scan phase of the exclusive scan computation. If *create_init_phase* is not specified, the effect is as if *create_init_phase* evaluates to *true*.

Cross References

- **scan** Directive, see Section 7.7

7.8 Data Copying Clauses

This section describes the **copyin** clause and the **copyprivate** clause. These two clauses support copying data values from private variables or threadprivate variables of an implicit task or thread to the corresponding variables of other implicit tasks or threads in the team.

7.8.1 `copyin` Clause

Name: `copyin`	**Properties:** outermost-leaf, data copying

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`parallel`

Semantics

The **`copyin`** clause provides a mechanism to copy the value of a threadprivate variable of the primary thread to the threadprivate variable of each other member of the team that is executing the **`parallel`** region.

C / C++

The copy is performed after the team is formed and prior to the execution of the associated structured block. For variables of non-array type, the copy is by copy assignment. For an array of elements of non-array type, each element is copied as if by assignment from an element of the array of the primary thread to the corresponding element of the array of all other threads.

C / C++

C++

For class types, the copy assignment operator is invoked. The order in which copy assignment operators for different variables of the same class type are invoked is unspecified.

C++

Fortran

The copy is performed, as if by assignment, after the team is formed and prior to the execution of the associated structured block.

Named variables that appear in a threadprivate common block may be specified. The whole common block does not need to be specified.

On entry to any **`parallel`** region, the copy of each thread of a variable that is affected by a **`copyin`** clause for the **`parallel`** region will acquire the type parameters, allocation, association, and definition status of the copy of the primary thread, according to the following rules:

- If the original list item has the **`POINTER`** attribute, each copy receives the same association status as that of the copy of the primary thread as if by pointer assignment.

- If the original list item does not have the **POINTER** attribute, each copy becomes defined with the value of the copy of the primary thread as if by intrinsic assignment unless the list item has a type bound procedure as a defined assignment. If the original list item does not have the **POINTER** attribute but has the allocation status of unallocated, each copy will have the same status.
- If the original list item is unallocated or unassociated, each copy inherits the declared type parameters and the default type parameter values from the original list item.

Fortran

Restrictions

Restrictions to the **copyin** clause are as follows:

- A list item that appears in a **copyin** clause must be threadprivate.

C++

- A variable of class type (or array thereof) that appears in a **copyin** clause requires an accessible, unambiguous copy assignment operator for the class type.

C++

Fortran

- A common block name that appears in a **copyin** clause must be declared to be a common block in the same scoping unit in which the **copyin** clause appears.

Fortran

Cross References

- **parallel** Construct, see Section 12.1

7.8.2 copyprivate Clause

Name: **copyprivate**	**Properties:** innermost-leaf, end-clause, data copying

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

single

Semantics

The **copyprivate** clause provides a mechanism to use a private variable to broadcast a value from the data environment of one implicit task to the data environments of the other implicit tasks that belong to the innermost enclosing parallel region. The effect of the **copyprivate** clause on the specified list items occurs after the execution of the structured block associated with the construct on which the clause is specified, and before any of the threads in the team have left the barrier at the end of the construct. To avoid data races, concurrent reads or updates of the list item must be synchronized with the update of the list item that occurs as a result of the **copyprivate** clause if, for example, the **nowait** clause is used to remove the barrier.

C / C++

In all other implicit tasks that belong to the parallel region, each specified list item becomes defined with the value of the corresponding list item in the implicit task associated with the thread that executed the structured block. For variables of non-array type, the definition occurs by copy assignment. For an array of elements of non-array type, each element is copied by copy assignment from an element of the array in the data environment of the implicit task that is associated with the thread that executed the structured block to the corresponding element of the array in the data environment of the other implicit tasks.

C / C++

C++

For class types, a copy assignment operator is invoked. The order in which copy assignment operators for different variables of class type are called is unspecified.

C++

Fortran

If a list item does not have the **POINTER** attribute then, in all other implicit tasks that belong to the parallel region, the list item becomes defined as if by intrinsic assignment with the value of the corresponding list item in the implicit task that is associated with the thread that executed the structured block. If the list item has a type bound procedure as a defined assignment, the assignment is performed by the defined assignment.

If the list item has the **POINTER** attribute then, in all other implicit tasks that belong to the parallel region, the list item receives, as if by pointer assignment, the same association status as the corresponding list item in the implicit task that is associated with the thread that executed the structured block.

The order in which any final subroutines for different variables of a finalizable type are called is unspecified.

Fortran

Restrictions

Restrictions to the **copyprivate** clause are as follows:

- All list items that appear in a **copyprivate** clause must be either threadprivate or private in the enclosing context.

C++

- A variable of class type (or array thereof) that appears in a **copyprivate** clause requires an accessible unambiguous copy assignment operator for the class type.

C++

Fortran

- A common block that appears in a **copyprivate** clause must be threadprivate.
- Pointers with the **INTENT(IN)** attribute must not appear in a **copyprivate** clause.
- Any list item with the **ALLOCATABLE** attribute must have the allocation status of allocated when the intrinsic assignment is performed.

Fortran

Cross References

- List Item Privatization, see Section 7.4
- **single** Construct, see Section 13.1
- **threadprivate** Directive, see Section 7.3

7.9 Data-Mapping Control

This section describes the available mechanisms for controlling how data are mapped to device data environments. It covers implicitly determined data-mapping attribute rules for variables referenced in **target** constructs, clauses that support explicitly determined data-mapping attributes, and clauses for mapping variables with static storage duration and making procedures available on other devices. It also describes how mappers may be defined and referenced to control the mapping of data with user-defined types. When storage is mapped, the programmer must ensure, by adding proper synchronization or by explicit unmapping, that the storage does not reach the end of its lifetime before it is unmapped.

7.9.1 *map-type* Modifier

Modifiers

Name	Modifies	Type	Properties
map-type	*all arguments*	Keyword: **from**, **storage**, **to**, **tofrom**	*default*

Clauses

map

Additional information

The value **alloc** may be used on map-entering constructs and the value **release** may be used on map-exiting constructs with identical meaning to the value **storage**.

Semantics

The *map-type* modifier determines the type of mapping operations that are performed as a result of the clause on which it appears. All mapping operations update the reference count of corresponding storage in a device data environment, which may entail creation or removal of that storage. The **storage** *map-type* never includes an assignment operation. If the *map-type* is **to**, **from**, or **tofrom**, the *map-type* is an assigning map type and may include an assignment operation to or from the target device.

The *map-type* is a map-entering map type if it is **to**, **tofrom**, or **storage**. The *map-type* is a map-exiting map type if it is **from**, **tofrom**, or **storage**. If the *map-type* is a map-entering map type, the clause on which the *map-type* appears is a map-entering clause. If the *map-type* is a map-exiting map type, the clause on which the *map-type* appears is a map-exiting clause.

When a *map-type* is not specified for a clause on which it may be specified, the *map-type* defaults to **storage** if the *delete-modifier* is present on the clause or if the list item for which the *map-type* is not specified is an assumed-size array. Otherwise, the *map-type* defaults to **tofrom** if a *map-type* is not specified for a clause on which it may be specified, unless otherwise specified.

Fortran

When a *map-type* is not specified for a clause on which it may be specified, the *map-type* defaults to **storage** if the list item for which the *map-type* is not specified is an assumed-type variable.

Fortran

Restrictions

Restrictions to the *map-type* modifier are as follows:

- If the clause on which the *map-type* appears is specified on a construct that is map-entering but not map-exiting, the *map-type* must be map-entering.
- If the clause on which the *map-type* appears is specified on a construct that is map-exiting but not map-entering, the *map-type* must be map-exiting.

Cross References

- **map** Clause, see Section 7.9.6

7.9.2 Map Type Decay

Map-type decay is a process that derives an output map type from a given input map type according to an underlying map type. This process is defined by Table 7.5, where the output map type is shown at the row and column that corresponds to the underlying map type and input map type, respectively. When map-type decay determines the *map-type* modifier to apply for a **map** clause on a data-mapping constituent directive of a composite construct, the input map type is given by the *map-type* modifier specified by the **map** clause on the composite construct and the underlying map type is respectively **to** or **from** for a map-entering constituent directive or a map-exiting constituent directive. When map-type decay is applied by an invoked mapper, the underlying map

TABLE 7.5: Map-Type Decay of Map Type Combinations

	storage	to	from	tofrom
storage	storage	storage	storage	storage
to	storage	to	storage	to
from	storage	storage	from	from
tofrom	storage	to	from	tofrom

type is given by the *map-type* modifier of the **map** clause specified by the mapper and the input map type is given by the *map-type* modifier of the **map** clause that invokes the mapper.

7.9.3 Implicit Data-Mapping Attribute Rules

When specified, data-mapping attribute clauses on **target** directives determine the data-mapping attributes for variables referenced in a **target** construct. Otherwise, the first matching rule from the following list determines the implicitly determined data-mapping attribute (or implicitly determined data-sharing attribute) for variables referenced in a **target** construct that do not have a predetermined data-sharing attribute according to Section 7.1.1. References to structure elements or array elements are treated as references to the structure or array, respectively, for the purposes of implicitly determined data-mapping attributes or implicitly determined data-sharing attributes of variables referenced in a **target** construct.

- If a variable appears in an **enter** or **link** clause on a declare target directive that does not have a **device_type** clause with the **nohost** *device-type-description* then it is treated as if it had appeared in a **map** clause with a *map-type* of **tofrom**.
- If a variable is the base variable of a list item in a **reduction**, **lastprivate** or **linear** clause on a compound target construct then the list item is treated as if it had appeared in a **map** clause with a *map-type* of **tofrom** if Section 19.2 specifies this behavior.
- If a variable is the base variable of a list item in an **in_reduction** clause on a **target** construct then it is treated as if the list item had appeared in a **map** clause with a *map-type* of **tofrom** and an *always-modifier*.
- If a **defaultmap** clause is present for the category of the variable and specifies an implicit behavior other than **default**, the data-mapping attribute or data-sharing attribute is determined by that clause.

C++

- If the **target** construct is within a class non-static member function, and a variable is an accessible data member of the object for which the non-static member function is invoked, the variable is treated as if the **this[:1]** expression had appeared in a **map** clause with a *map-type* of **tofrom**. Additionally, if the variable is of type pointer or reference to pointer, it is also treated as if it is the array base of a zero-offset assumed-size array that appears in a **map** clause with the **storage** *map-type*.

- If the **this** keyword is referenced inside a **target** construct within a class non-static member function, it is treated as if the **this[:1]** expression had appeared in a **map** clause with a *map-type* of **tofrom**.

C++

C / C++

- A variable that is of type pointer, but is neither a pointer to function nor (for C++) a pointer to a member function, is treated as if it is the array base of a zero-offset assumed-size array that appears in a **map** clause with the **storage** *map-type*.

C / C++

C++

- A variable that is of type reference to pointer, but is neither a reference to pointer to function nor a reference to a pointer to a member function, is treated as if it is the array base of a zero-offset assumed-size array that appears in a **map** clause with the **storage** *map-type*.

C++

Fortran

- If a compound target construct is associated with a **DO CONCURRENT** loop, a variable that has **REDUCE** or **SHARED** locality in the loop is treated as if it had appeared in a **map** clause with a *map-type* of **tofrom**.

Fortran

- If a variable is not a scalar variable then it is treated as if it had appeared in a **map** clause with a *map-type* of **tofrom**.

Fortran

- If a scalar variable has the **TARGET**, **ALLOCATABLE** or **POINTER** attribute then it is treated as if it had appeared in a **map** clause with a *map-type* of **tofrom**.
- If a variable is an assumed-type variable then it is treated as if it had appeared in a **map** clause with a *map-type* of **storage**.
- A procedure pointer is treated as if it had appeared in a **firstprivate** clause.

Fortran

- If the above rules do not apply then a scalar variable is not mapped but instead has an implicitly determined data-sharing attribute of firstprivate (see Section 7.1.1).

7.9.4 Mapper Identifiers and `mapper` Modifiers

Modifiers

Name	Modifies	Type	Properties
mapper	*locator-list*	Complex, name: **`mapper`** Arguments: ***mapper-identifier*** OpenMP identifier (*default*)	unique

Clauses

`from`, **`map`**, **`to`**

Semantics

Mapper identifiers can be used to identify uniquely the mapper used in a **`map`** or data-motion clause through a *mapper* modifier, which is a unique, complex modifier. A **`declare_mapper`** directive defines a mapper identifier that can later be specified in a *mapper* modifier as its *modifier-parameter-specification*. Each mapper identifier is a base language identifier or **`default`** where **`default`** is the default mapper for all types.

A non-structure type *T* has a predefined default mapper that is defined as if by the following **`declare_mapper`** directive:

C / C++

```
#pragma omp declare_mapper(T v) map(tofrom: v)
```

C / C++

Fortran

```
!$omp declare_mapper(T :: v) map(tofrom: v)
```

Fortran

A structure type *T* has a predefined default mapper that is defined as if by a **`declare_mapper`** directive that specifies *v* in a **`map`** clause with the **`storage`** *map-type* and each structure element of *v* in a **`map`** clause with the **`tofrom`** *map-type*.

A **`declare_mapper`** directive that uses the **`default`** *mapper* identifier overrides the predefined default mapper for the given type, making it the default mapper for variables of that type.

Cross References

- **`declare_mapper`** Directive, see Section 7.9.10
- **`from`** Clause, see Section 7.10.2
- Data-Motion Clauses, see Section 7.10
- **`map`** Clause, see Section 7.9.6
- **`to`** Clause, see Section 7.10.1

7.9.5 `ref-modifier` Modifier

Modifiers

Name	Modifies	Type	Properties
ref-modifier	*all arguments*	Keyword: **`ref_ptee`**, **`ref_ptr`**, **`ref_ptr_ptee`**	unique

Clauses

`map`

Semantics

The *ref-modifier* for a given clause indicates how to interpret the identity of a list item argument of that clause. If the **`ref_ptr`** or **`ref_ptr_ptee`** *ref-modifier* is specified, the semantics of the clause apply to the referring pointer of the referencing variable. If the **`ref_ptee`** or **`ref_ptr_ptee`** *ref-modifier* is specified and a referenced pointee of the referencing variable exists, the semantics of the clause apply to the referenced pointee.

Restrictions

Restrictions to the *ref-modifier* are as follows:

- A list item that appears in a clause with the *ref-modifier* must be a referencing variable.

C / C++

- A list item that appears in a clause for which the *ref-modifier* is specified must have a containing structure.

C / C++

Cross References

- **`map`** Clause, see Section 7.9.6

7.9.6 `map` Clause

Name: **`map`**	**Properties:** data-environment attribute, data-mapping attribute

Arguments

Name	Type	Properties
locator-list	list of locator list item type	*default*

Modifiers

Name	Modifies	Type	Properties
always-modifier	*locator-list*	Keyword: **always**	map-type-modifying
close-modifier	*locator-list*	Keyword: **close**	map-type-modifying
present-modifier	*locator-list*	Keyword: **present**	map-type-modifying
self-modifier	*locator-list*	Keyword: **self**	map-type-modifying
ref-modifier	*all arguments*	Keyword: **ref_ptee**, **ref_ptr**, **ref_ptr_ptee**	unique
delete-modifier	*locator-list*	Keyword: **delete**	map-type-modifying
mapper	*locator-list*	Complex, name: **mapper** Arguments: ***mapper-identifier*** OpenMP identifier (*default*)	unique
iterator	*locator-list*	Complex, name: **iterator** Arguments: ***iterator-specifier*** list of iterator specifier list item type (*default*)	unique
map-type	*all arguments*	Keyword: **from**, **storage**, **to**, **tofrom**	*default*
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

declare_mapper, **target**, **target_data**, **target_enter_data**, **target_exit_data**

Semantics

The **map** clause specifies how an original list item is mapped from the data environment of the current task to a corresponding list item in the device data environment of the device identified by the construct. The list items that appear on a **map** clause may include array sections, assumed-size arrays, and structure elements. A list item in a **map** clause may reference any *iterator-identifier* defined in its *iterator* modifier. A list item may appear more than once in the **map** clauses that are specified on the same directive.

C / C++

If a list item is a zero-length array section that has a single array subscript, the behavior is as if the list item is an assumed-size array that is instead mapped with the **storage** *map-type*.

C / C++

When a list item in a **map** clause that is not an assumed-size array is mapped on a map-entering construct and corresponding storage is created in the device data environment on entry to the region, the list item becomes a matchable candidate with an associated starting address, ending address, and base address that define its mapped address range and extended address range. The current set of matchable candidates consists of any **map** clause list item on the construct that is a matchable candidate and all matchable candidates that were previously mapped and are still mapped.

A list item in a **map** clause that is an assumed-size array is treated as if an array section, with an array base, lower bound and length determined as follows, is substituted in its place if a matched candidate is found. If the assumed-size array is an array section, the array base of the substitute array section is the same as for the assumed-size array; otherwise, the array base is the assumed-size array. If the mapped address range of a matchable candidate includes the first storage location of the assumed-size array, it is a matched candidate. If a matchable candidate does not exist for which the mapped address range includes the first storage location of the assumed-size array then a matchable candidate is a matched candidate if its extended address range includes the first storage location of the assumed-size array. If multiple matched candidates exist, an arbitrary one of them is the found matched candidate. The lower bound and length of the substitute array section are set such that its storage is identical to the storage of the found matched candidate. If a matched candidate is not found then a substitute array section is not formed and no further actions that are described in this section are performed for the list item.

Fortran

The list items may include assumed-type variables and procedure pointers.

If a list item in a **map** clause is an assumed-type variable for which the storage location is included in the mapped address range of a matchable candidate, the list item is treated as if it refers to the storage of that matchable candidate. Otherwise, no further actions that are described in this section are performed for the list item.

Fortran

If a list item is an array or array section, the array elements become implicit list items with the same modifiers (including the *map-type*) specified in the clause. If the array or array section is implicitly mapped and corresponding storage exists in the device data environment prior to a task encountering the construct on which the **map** clause appears, only those array elements that have corresponding storage are implicitly mapped.

If a *mapper* modifier is not present, the behavior is as if a *mapper* modifier was specified with the **default** parameter. The map behavior of a list item in a **map** clause is modified by a visible user-defined mapper (see Section 7.9.10) if the *mapper-identifier* of the *mapper* modifier is defined for a base language type that matches the type of the list item. Otherwise, the predefined default mapper for the type of the list item applies. The effect of the *mapper* modifier is to remove the list item from the **map** clause and to apply the clauses specified in the declared mapper to the construct on which the **map** clause appears. In the clauses applied by the *mapper*, references to *var* are replaced with references to the list item and the *map-type* is replaced with the output map type that is determined according to the rules of map-type decay. If any modifier with the map-type-modifying property appears in the **map** clause then the effect is as if that modifier appears in each **map** clause specified in the declared mapper.

Unless otherwise specified, if a list item is a referencing variable then the effect of the **map** clause is applied to its referring pointer and, if a referenced pointee exists, its referenced pointee. For the purposes of the **map** clause, the referenced pointee is treated as if its referring pointer is the referring pointer of the referencing variable.

C++

If a list item is a reference and it does not have a containing structure then the **map** clause is applied only to its referenced pointee.

C++

Fortran

If a component of a derived type list item is a **map** clause list item that results from the predefined default mapper for that derived type, and if the derived type component is not an explicit list item or the array base of an explicit list item in a **map** clause on the construct then:

- If it has the **POINTER** attribute, it is attach-ineligible; and
- If it has the **ALLOCATABLE** attribute and an allocated allocation status, and it is present in the device data environment when the construct is encountered, the **map** clause may treat its allocation status as if it is unallocated if the corresponding component does not have allocated storage.

If a list item in a **map** clause is an associated pointer that is attach-ineligible or the pointer is the base pointer of another list item in a **map** clause on the same construct then the effect of the **map** clause does not apply to its pointer target.

If a list item is a procedure pointer, it is attach-ineligible.

Fortran

C++

If a list item has a closure type that is associated with a lambda expression, it is mapped as if it has a structure type. For each variable that is captured by reference by the lambda expression, the behavior is as if the closure type contains a non-static data member that is a reference to that variable unless otherwise specified. If a variable that is captured by reference is a reference that binds to an object with static storage duration, a corresponding non-static data member might not exist in the closure type. For the corresponding list item of closure type, references in the body of the lambda expression to a variable that is captured by reference refer to the corresponding storage of the variable in the device data environment. For each pointer, that is not a function pointer, that is captured by the lambda expression, the behavior is as if the pointer or, if a corresponding pointer member exists, the corresponding pointer member of the closure object is the base pointer of a zero-offset assumed-size array that appears as a list item in a **map** clause with the **storage** *map-type*.

If the **this** pointer is captured by a lambda expression in class scope, and a variable of the associated closure type is later mapped explicitly or implicitly with its full static type, the behavior is as if the object to which **this** points is also mapped as an array section, of length one, for which the base pointer is the non-static data member that corresponds to the **this** pointer in the closure object.

C++

If a **map** clause with a *present-modifier* appears on a construct and on entry to the region the corresponding list item is not present in the device data environment, runtime error termination is performed.

If a map-entering clause has the *self-modifier*, the resulting mapping operations are self maps.

The effective **map** clause set of a data-mapping construct is the set of all **map** clauses that apply to that construct, including implicit **map** clauses and **map** clauses applied by mappers. The effective **map** clause set of a construct determines the set of mappable storage blocks for that construct. All **map** clause list items that share storage or have the same containing structure or containing array result in a single mappable storage block that contains the storage of the list items, unless otherwise specified. The storage for each other **map** clause list item becomes a distinct mappable storage block. If a list item is a referencing variable that has a containing structure, the behavior is as if only the storage for its referring pointer is part of that structure. In general, if a list item is a referencing variable then the storage for its referring pointer and its referenced pointee occupy distinct mappable storage blocks.

For each mappable storage block that is determined by the effective **map** clause set of a map-entering construct, on entry to the region the following sequence of steps occurs as if performed as a single atomic operation:

1. If a corresponding storage block is not present in the device data environment then:
   a) A corresponding storage block, which may share storage with the original storage block, is created in the device data environment of the target device;
   b) The corresponding storage block receives a reference count that is initialized to zero. This reference count also applies to any part of the corresponding storage block.
2. The reference count of the corresponding storage block is incremented by one.
3. For each **map** clause list item in the effective **map** clause set that is contained by the mappable storage block:
   a) If the reference count of the corresponding storage block is one, a new list item with language-specific attributes derived from the original list item is created in the corresponding storage block. The reference count of the new list item is always equal to the reference count of its storage.
   b) If the reference count of the corresponding list item is one or if the *always-modifier* is specified, and if the map type is **to**, the corresponding list item is updated as if the list item appeared in a **to** clause on a **target_update** directive.

If the effect of the **map** clauses on a construct would assign the value of an original list item to a corresponding list item more than once then an implementation is allowed to ignore additional assignments of the same value to the corresponding list item.

In all cases on entry to the region, concurrent reads or updates of any part of the corresponding list item must be synchronized with any update of the corresponding list item that occurs as a result of the **map** clause to avoid data races.

For **map** clauses on map-entering constructs, if any list item has a base pointer or referring pointer for which a corresponding pointer exists in the device data environment after all mappable storage blocks are mapped, and either a new list item or the corresponding pointer is created in the device data environment on entry to the region, then pointer attachment is performed and the corresponding pointer becomes an attached pointer to the corresponding list item via corresponding pointer initialization.

The original list item and corresponding list item may share storage such that writes to either item by one task followed by a read or write of the other list item by another task without intervening synchronization can result in data races. They are guaranteed to share storage if the mapping operation is a self map, if the **map** clause appears on a data-mapping construct for which the target device is the encountering device, or if the corresponding list item has an attached pointer that shares storage with its original pointer.

For each mappable storage block that is determined by the effective **map** clause set of a map-exiting construct, and for which corresponding storage is present in the device data environment, on exit from the region the following sequence of steps occurs as if performed as a single atomic operation:

1. For each **map** clause list item in the effective **map** clause set that is contained by the mappable storage block:

    a) If the reference count of the corresponding list item is one or if the *always-modifier* or *delete-modifier* is specified, and if the map type is **from**, the original list item is updated as if the list item appeared in a **from** clause on a **target_update** directive.

2. If the *delete-modifier* is not present and the reference count of the corresponding storage block is finite then the reference count is decremented by one.

3. If the *delete-modifier* is present and the reference count of the corresponding storage block is finite then the reference count is set to zero.

4. If the reference count of the corresponding storage block is zero, all storage to which that reference count applies is removed from the device data environment.

If the effect of the **map** clauses on a construct would assign the value of a corresponding list item to an original list item more than once, then an implementation is allowed to ignore additional assignments of the same value to the original list item.

In all cases on exit from the region, concurrent reads or updates of any part of the original list item must be synchronized with any update of the original list item that occurs as a result of the **map** clause to avoid data races.

If a single contiguous part of the original storage of a list item that results from an implicitly determined data-mapping attribute has corresponding storage in the device data environment prior to a task encountering the construct on which the **map** clause appears, only that part of the original storage will have corresponding storage in the device data environment as a result of the **map** clause.

If a list item with an implicitly determined data-mapping attribute does not have any corresponding storage in the device data environment prior to a task encountering the construct associated with the **map** clause, and one or more contiguous parts of the original storage are either list items or base pointers to list items that are explicitly mapped on the construct, only those parts of the original storage will have corresponding storage in the device data environment as a result of the **map** clauses on the construct.

C / C++

If a new list item is created then the new list item will have the same static type as the original list item, and language-specific attributes of the new list item, including size and alignment, are determined by that type.

C / C++

C++

If corresponding storage that differs from the original storage is created in a device data environment, all new list items that are created in that corresponding storage are default initialized. Default initialization for new list items of class type, including their data members, is performed as if with an implicitly-declared default constructor and as if non-static data member initializers are ignored.

C++

Fortran

If a new list item is created then the new list item will have the same type, type parameter, and rank as the original list item. The new list item inherits all default values for the type parameters from the original list item.

Fortran

The *close-modifier* is a hint that the corresponding storage should be close to the target device.

If a map-entering clause specifies a self map for a list item then runtime error termination is performed if any of the following is true:

- The original list item is not accessible and cannot be made accessible from the target device;
- The corresponding list item is present prior to a task encountering the construct on which the clause appears, and the corresponding storage differs from the original storage; or
- The list item is a pointer that would be assigned a different value as a result of pointer attachment.

Execution Model Events

The *target-map* event occurs in a thread that executes the outermost region that corresponds to an encountered device construct with a **map** clause, after the *target-task-begin* event for the device construct and before any mapping operations are performed. The *target-data-op-begin* event occurs before a thread initiates a data operation on the target device that is associated with a **map** clause, in the outermost region that corresponds to the encountered construct. The *target-data-op-end* event occurs after a thread initiates a data operation on the target device that is associated with a **map** clause, in the outermost region that corresponds to the encountered construct.

Tool Callbacks

A thread dispatches one or more registered **target_map_emi** callbacks for each occurrence of a *target-map* event in that thread. The callback occurs in the context of the target task. A thread dispatches a registered **target_data_op_emi** callback with **ompt_scope_begin** as its endpoint argument for each occurrence of a *target-data-op-begin* event in that thread. Similarly, a thread dispatches a registered **target_data_op_emi** callback with **ompt_scope_end** as its endpoint argument for each occurrence of a *target-data-op-end* event in that thread.

Restrictions

Restrictions to the **map** clause are as follows:

- Two list items of the **map** clauses on the same construct must not share original storage unless one of the following is true: they are the same list item, one is the containing structure of the other, at least one is an assumed-size array, or at least one is implicitly mapped due to the list item also appearing in a **use_device_addr** clause.
- If the same list item appears more than once in **map** clauses on the same construct, the **map** clauses must specify the same *mapper* modifier.
- A variable that is a groupprivate variable or a device-local variable must not appear as a list item in a **map** clause.
- If a list item is an array or an array section, it must specify contiguous storage.
- If an expression that is used to form a list item in a **map** clause contains an iterator identifier that is defined by an *iterator* modifier, the list item instances that would result from different values of the iterator must not have the same containing array and must not have base pointers that share original storage.
- If multiple list items are explicitly mapped on the same construct and have the same containing array or have base pointers that share original storage, and if any of the list items do not have corresponding list items that are present in the device data environment prior to a task encountering the construct, then the list items must refer to the same array elements of either the containing array or the implicit array of the base pointers.
- If any part of the original storage of a list item that is explicitly mapped by a **map** clause has corresponding storage in the device data environment prior to a task encountering the construct associated with the **map** clause, all of the original storage must have corresponding storage in the device data environment prior to the task encountering the construct.

- If a list item in a **map** clause has corresponding storage in the device data environment, all corresponding storage must correspond to a single mappable storage block that was previously mapped.
- If a list item is an element of a structure, and a different element of the structure has a corresponding list item in the device data environment prior to a task encountering the construct associated with the **map** clause, then the list item must also have a corresponding list item in the device data environment prior to the task encountering the construct.
- Each list item must have a mappable type.
- If a *mapper* modifier appears in a **map** clause, the type on which the specified mapper operates must match the type of the list items in the clause.
- Handles for memory spaces and memory allocators must not appear as list items in a **map** clause.
- If a list item is an assumed-size array, multiple matched candidates must not exist unless they are subobjects of the same containing structure.
- If a list item is an assumed-size array, the *map-type* must be **storage**.
- If a list item appears in a **map** clause with the *self-modifier*, any other list item in a **map** clause on the same construct that has the same base variable or base pointer must also be specified with the *self-modifier*.

C++

- If a list item has a polymorphic class type and its static type does not match its dynamic type, the behavior is unspecified if the **map** clause is specified on a map-entering construct and a corresponding list item is not present in the device data environment prior to a task encountering the construct.
- No type mapped through a reference may contain a reference to its own type, or any references to types that could produce a cycle of references.
- If a given variable is captured by reference by the associated lambda expression of a list item that has a closure type and that variable is a reference that binds to a variable with static storage duration, the variable to which it binds must appear in an **enter** clause or a **link** clause on a declare target directive and must have corresponding storage in the device data environment prior to a task encountering the construct.

C++

C / C++

- A list item cannot be a variable that is a member of a structure of a union type.
- A bit-field cannot appear in a **map** clause.
- A pointer that has a corresponding pointer that is an attached pointer must not be modified for the duration of the lifetime of the list item to which the corresponding pointer is attached in the device data environment.

C / C++

Fortran

- The association status of a list item that is a pointer must not be undefined unless it is a structure component and it results from a predefined default mapper.
- If a list item of a **map** clause is an allocatable variable or is the subobject of an allocatable variable, the original list item must not be allocated, deallocated or reshaped while the corresponding list item has allocated storage.
- A pointer that has a corresponding pointer that is an attached pointer and is associated with a given pointer target must not become associated with a different pointer target for the duration of the lifetime of the list item to which the corresponding pointer is attached in the device data environment.
- If a list item has polymorphic type, the behavior is unspecified.
- If an array section is mapped and the size of the array section is smaller than that of the whole array, the behavior of referencing the whole array in a **target** region is unspecified.
- A list item must not be a complex part designator.
- If a list item is an assumed-type variable, the *map-type* must be **storage**.

Fortran

Cross References

- **declare_mapper** Directive, see Section 7.9.10
- Array Sections, see Section 5.2.5
- **iterator** Modifier, see Section 5.2.6
- Mapper Identifiers and **mapper** Modifiers, see Section 7.9.4
- *map-type* Modifier, see Section 7.9.1
- OMPT **scope_endpoint** Type, see Section 33.27
- **target** Construct, see Section 15.8
- **target_data** Construct, see Section 15.7
- **target_data_op_emi** Callback, see Section 35.7
- **target_enter_data** Construct, see Section 15.5
- **target_exit_data** Construct, see Section 15.6
- **target_map_emi** Callback, see Section 35.9
- **target_update** Construct, see Section 15.9

7.9.7 `enter` Clause

Name: `enter`	Properties: data-environment attribute, data-mapping attribute

Arguments

Name	Type	Properties
list	list of extended list item type	*default*

Modifiers

Name	Modifies	Type	Properties
automap-modifier	*list*	Keyword: `automap`	*default*
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`declare_target`

Semantics

The `enter` clause is a data-mapping attribute clause.

If a procedure name appears in an `enter` clause in the same compilation unit in which the definition of the procedure occurs then a device-specific version of the procedure is created for all devices to which the directive of the clause applies.

C / C++

If a variable appears in an `enter` clause in the same compilation unit in which the definition of the variable occurs then a corresponding list item to the original list item is created in the device data environment of all devices to which the directive of the clause applies.

C / C++

Fortran

If a variable that is host associated appears in an `enter` clause then a corresponding list item to the original list item is created in the device data environment of all devices to which the directive of the clause applies.

Fortran

If a variable appears in an `enter` clause then the corresponding list item in the device data environment of each device to which the directive of the clause applies is initialized once, in the manner specified by the OpenMP program, but at an unspecified point in the OpenMP program prior to the first reference to that list item. The list item is never removed from those device data environments, as if its reference count was initialized to positive infinity, unless otherwise specified.

If a list item is a referencing variable, the effect of the `enter` clause applies to its referring pointer.

Fortran

If a list item is an allocatable variable, the *automap-modifier* is present, and the variable is allocated by an **ALLOCATE** statement or deallocated by a **DEALLOCATE** statement where the **enter** clause is visible, the behavior is as follows:

- Upon allocation due to the **ALLOCATE** statement, the list item is mapped to the device data environment of the default device as if it appeared as a list item in a **map** clause on a **target_enter_data** directive; and
- Immediately prior to the deallocation due to the **DEALLOCATE** statement, the list item is removed from the device data environment of the default device as if it appeared as a list item in a **map** clause with the *delete-modifier* on a **target_exit_data** directive.

Fortran

Restrictions

Restrictions to the **enter** clause are as follows:

- Each list item must have a mappable type.
- Each list item must have static storage duration.

C / C++

- The *automap-modifier* must not be present.

C / C++

Fortran

- If the *automap-modifier* is present, each list item must be an allocatable variable.

Fortran

Cross References

- **declare_target** Directive, see Section 9.9.1

7.9.8 link Clause

Name: link	**Properties:** data-environment attribute

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

declare_target

Semantics

The **link** clause supports compilation of device procedures that refer to variables with static storage duration that appear as list items in the clause. The **declare_target** directive on which the clause appears does not map the list items. Instead, they are mapped according to the data-mapping rules described in Section 7.9.3.

Restrictions

Restrictions to the **link** clause are as follows:

- Each list item must have a mappable type.
- Each list item must have static storage duration.

Cross References

- **declare_target** Directive, see Section 9.9.1
- Data-Mapping Control, see Section 7.9

7.9.9 defaultmap Clause

Name: **defaultmap**	**Properties:** unique, post-modified

Arguments

Name	Type	Properties
implicit-behavior	Keyword: **default**, **firstprivate**, **from**, **none**, **present**, **private**, **self**, **storage**, **to**, **tofrom**	*default*

Modifiers

Name	Modifies	Type	Properties
variable-category	*implicit-behavior*	Keyword: **aggregate**, **all**, **allocatable**, **pointer**, **scalar**	*default*
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

target

Additional information

The value **alloc** may also be specified as *implicit-behavior* with identical meaning to the value **storage**.

Semantics

The **defaultmap** clause controls the implicitly determined data-mapping attributes or implicitly determined data-sharing attributes of certain variables that are referenced in a **target** construct, in accordance with the rules given in Section 7.9.3. The *variable-category* specifies the variables for which the attribute may be set, and the attribute is specified by *implicit-behavior*. If no *variable-category* is specified in the clause then the effect is as if **all** was specified for the *variable-category*.

C / C++

The **scalar** *variable-category* specifies non-pointer scalar variables.

C / C++

Fortran

The **scalar** *variable-category* specifies non-pointer and non-allocatable scalar variables. The **allocatable** *variable-category* specifies variables with the **ALLOCATABLE** attribute.

Fortran

The **pointer** *variable-category* specifies variables of pointer type. The **aggregate** *variable-category* specifies aggregate variables. Finally, the **all** *variable-category* specifies all variables.

If *implicit-behavior* corresponds to a *map-type*, the attribute is a data-mapping attribute determined by an implicit **map** clause with the specified *map-type*. If *implicit-behavior* is **firstprivate**, the attribute is a data-sharing attribute of firstprivate. If *implicit-behavior* is **present**, the attribute is a data-mapping attribute determined by an implicit **map** clause with a *map-type* of **storage** and the *present-modifier*. If *implicit-behavior* is **self**, the attribute is a data-mapping attribute determined by an implicit **map** clause with a *map-type* of **storage** and the *self-modifier*. If *implicit-behavior* is **none** then no implicitly determined data-mapping attributes or implicitly determined data-sharing attributes are defined for variables in *variable-category*, except for variables that appear in the **enter** or **link** clause of a **declare_target** directive. If *implicit-behavior* is **default** then the clause has no effect.

Restrictions

Restrictions to the **defaultmap** clause are as follows:

- A given *variable-category* may be specified in at most one **defaultmap** clause on a construct.
- If a **defaultmap** clause specifies the **all** *variable-category*, no other **defaultmap** clause may appear on the construct.
- If *implicit-behavior* is **none**, each variable that is specified by *variable-category* and is referenced in the construct but does not have a predetermined data-sharing attribute and does not appear in an **enter** or **link** clause on a **declare_target** directive must be explicitly listed in a data-environment attribute clause on the construct.

C / C++

- The specified *variable-category* must not be **allocatable**.

C / C++

Cross References

- Implicit Data-Mapping Attribute Rules, see Section 7.9.3
- **target** Construct, see Section 15.8

7.9.10 declare_mapper Directive

Name: declare_mapper **Category:** declarative	**Association:** unassociated **Properties:** pure

Arguments

declare_mapper(*mapper-specifier***)**

Name	Type	Properties
mapper-specifier	OpenMP mapper specifier	*default*

Clauses

map

Additional information

The **declare_mapper** directive may alternatively be specified with **declare mapper** as the *directive-name*.

Semantics

User-defined mappers can be defined using the **declare_mapper** directive. The *mapper-specifier* argument declares the mapper using the following syntax:

C / C++

```
[ mapper-identifier : ] type var
```

C / C++

Fortran

```
[ mapper-identifier : ] type :: var
```

Fortran

where *mapper-identifier* is a mapper identifier, *type* is a type that is permitted in a type-name list, and *var* is a base language identifier.

The *type* and an optional *mapper-identifier* uniquely identify the mapper for use in a **map** clause or data-motion clause later in the OpenMP program.

If *mapper-identifier* is not specified, the behavior is as if *mapper-identifier* is **default**.

The variable declared by *var* is available for use in all **map** clauses on the directive, and no part of the variable to be mapped is mapped by default.

The effect that a user-defined mapper has on either a **map** clause that maps a list item of the given base language type or a data-motion clause that invokes the mapper and updates a list item of the given base language type is to replace the map or update with a set of **map** clauses or updates derived from the **map** clauses specified by the mapper, as described in Section 7.9.6 and Section 7.10.

A list item in a **map** clause that appears on a **declare_mapper** directive may include array sections.

All **map** clauses that are introduced by a mapper are further subject to mappers that are in scope, except a **map** clause with list item *var* maps *var* without invoking a mapper.

C++

The **declare_mapper** directive can also appear at locations in the OpenMP program at which a static data member could be declared. In this case, the visibility and accessibility of the declaration are the same as those of a static data member declared at the same location in the OpenMP program.

C++

Restrictions

Restrictions to the **declare_mapper** directive are as follows:

- No instance of *type* can be mapped as part of the mapper, either directly or indirectly through another base language type, except the instance *var* that is passed as the list item. If a set of **declare_mapper** directives results in a cyclic definition then the behavior is unspecified.
- The *type* must not declare a new base language type.
- At least one **map** clause that maps *var* or at least one element of *var* is required.
- List items in **map** clauses on the **declare_mapper** directive may only refer to the declared variable *var* and entities that could be referenced by a procedure defined at the same location.
- If a *mapper* modifier is specified for a **map** clause, its parameter must be **default**.
- Multiple **declare_mapper** directives that specify the same *mapper-identifier* for the same base language type or for compatible base language types, according to the base language rules, must not appear in the same scope.

C

- *type* must be a **struct** or **union** type.

C

C++

- *type* must be a **struct**, **union**, or **class** type.
- If *type* is a **struct** or **class** type, it must not be derived from any virtual base class.

C++

Fortran

- *type* must not be an intrinsic type, a parameterized derived type, an enum type, or an enumeration type.

Fortran

Cross References

- **map** Clause, see Section 7.9.6

7.10 Data-Motion Clauses

A data-motion clause specifies data movement between devices in a device set that is specified by the construct on which the clause appears, where one of the devices in the set is the encountering device and the remaining devices are target devices of the construct. Each data-motion clause specifies a data-motion attribute relative to the target devices.

A data-motion clause specifies an OpenMP locator list as its argument. A corresponding list item and an original list item exist for each list item. If the corresponding list item is not present in the device data environment then no assignment occurs between the corresponding list item and the original list item. Otherwise, each corresponding list item in the device data environment has an original list item in the data environment of the encountering task. Assignment is performed to either the original list item or the corresponding list item as specified with the specific data-motion clauses. List items may reference any *iterator-identifier* defined in an *iterator* modifier on the clause. The list items may include array sections with *stride* expressions.

C / C++

The list items may use shape-operators.

C / C++

If a list item is an array or array section then it is treated as if it is replaced by each of its array elements in the clause.

If the *mapper* modifier is not specified, the behavior is as if the modifier was specified with the **default** mapper identifier. The effect of a data-motion clause on a list item is modified by a visible user-defined mapper if a *mapper* modifier is specified with a mapper identifier for a type that matches the type of the list item. Otherwise, the predefined default mapper for the type of the list item applies. Each list item is replaced with the list items that the given mapper specifies are to be mapped with a compatible map type with respect to the data-motion attribute of the clause.

If a *present-modifier* is specified and the corresponding list item is not present in the device data environment then runtime error termination is performed. For a list item that is replaced with a set of list items as a result of a user-defined mapper, the *present-modifier* only applies to those mapper list items that share storage with the original list item.

If a list item is a referencing variable then the effect of the data-motion clause is applied only to its referenced pointee and only if the referenced pointee exists.

Fortran

If a list item is an associated procedure pointer, the corresponding list item on the device is associated with the target procedure of the host device.

Fortran

C / C++

On exit from the associated region, if the corresponding list item is an attached pointer, the original list item will have the value it had on entry to the region and the corresponding list item will have the value it had on entry to the region.

C / C++

For each list item that is not an attached pointer, the value of the assigned list item is assigned the value of the other list item. To avoid data races, concurrent reads or updates of the assigned list item must be synchronized with the update of an assigned list item that occurs as a result of a data-motion clause.

Restrictions

Restrictions to data-motion clauses are as follows:

- Each list item of *locator-list* must have a mappable type.
- If an array appears as a list item in a data-motion clause and it has corresponding storage in the device data environment, the corresponding storage must correspond to a single mappable storage block that was previously mapped.
- If a list item in a data-motion clause has corresponding storage in the device data environment, all corresponding storage must correspond to a single mappable storage block that was previously mapped.
- If a *mapper* modifier appears in a data-motion clause, the specified mapper must operate on a type that matches either the type or array element type of each list item in the clause.

Fortran

- The association status of a list item that is a pointer must not be undefined unless it is a structure component and it results from a predefined default mapper.

Fortran

Cross References

- **declare_mapper** Directive, see Section 7.9.10
- **device** Clause, see Section 15.2
- **from** Clause, see Section 7.10.2
- Array Sections, see Section 5.2.5
- Array Shaping, see Section 5.2.4
- **iterator** Modifier, see Section 5.2.6
- **target_update** Construct, see Section 15.9
- **to** Clause, see Section 7.10.1

7.10.1 to Clause

Name: to	**Properties:** data-motion attribute

Arguments

Name	Type	Properties
locator-list	list of locator list item type	*default*

Modifiers

Name	Modifies	Type	Properties
present-modifier	*locator-list*	Keyword: **present**	*default*
mapper	*locator-list*	Complex, name: **mapper** Arguments: ***mapper-identifier*** OpenMP identifier (*default*)	unique
iterator	*locator-list*	Complex, name: **iterator** Arguments: ***iterator-specifier*** list of iterator specifier list item type (*default*)	unique
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

target_update

Semantics

The **to** clause is a data-motion clause that specifies data movement to the target devices from the encountering device so the corresponding list items are the assigned list items and the compatible map types are **to** and **tofrom**.

C++

A list item for which a mapper does not exist is ignored if it has static storage duration and either it has the **constexpr** specifier or it is a non-mutable member of a structure that has the **constexpr** specifier.

C++

Cross References

- **iterator** Modifier, see Section 5.2.6
- **target_update** Construct, see Section 15.9

7.10.2 from Clause

Name: from	**Properties:** data-motion attribute

Arguments

Name	Type	Properties
locator-list	list of locator list item type	*default*

Modifiers

Name	Modifies	Type	Properties
present-modifier	*locator-list*	Keyword: **present**	*default*
mapper	*locator-list*	Complex, name: **mapper** Arguments: ***mapper-identifier*** OpenMP identifier (*default*)	unique
iterator	*locator-list*	Complex, name: **iterator** Arguments: ***iterator-specifier*** list of iterator specifier list item type (*default*)	unique
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

target_update

Semantics

The **from** clause is a data-motion clause that specifies data movement from the target devices to the encountering device so the original list items are the assigned list items and the compatible map types are **from** and **tofrom**.

C

A list item for which a mapper does not exist is ignored if it has the **const** specifier or if it is a member of a structure that has the **const** specifier.

C

C++

A list item for which a mapper does not exist is ignored if it has the **const** or **constexpr** specifier or if it is a non-mutable member of a structure that has the **const** or **constexpr** specifier.

C++

Cross References

- **iterator** Modifier, see Section 5.2.6
- **target_update** Construct, see Section 15.9

7.11 uniform Clause

Name: uniform	**Properties:** data-environment attribute

Arguments

Name	Type	Properties
parameter-list	list of parameter list item type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

declare_simd

Semantics

The **uniform** clause declares one or more arguments to have an invariant value for all concurrent invocations of the function in the execution of a single SIMD loop.

Restrictions

Restrictions to the **uniform** clause are as follows:

- Only named parameter list items can be specified in the *parameter-list*.

Cross References

- **declare_simd** Directive, see Section 9.8

7.12 `aligned` Clause

Name: `aligned`	**Properties:** data-environment attribute, post-modified

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
alignment	*list*	OpenMP integer expression	positive, region invariant, ultimate, unique
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`declare_simd`, **`simd`**

Semantics

C / C++

The **`aligned`** clause declares that the object to which each list item points is aligned to the number of bytes expressed in *alignment*.

C / C++

Fortran

The **`aligned`** clause declares that the target of each list item is aligned to the number of bytes expressed in *alignment*.

Fortran

The *alignment* modifier specifies the alignment that the program ensures related to the list items. If the *alignment* modifier is not specified, implementation defined default alignments for SIMD instructions on the target platforms are assumed.

Restrictions

Restrictions to the **`aligned`** clause are as follows:

- If the clause appears on a **`declare_simd`** directive, each list item must be a named parameter list item of the associated procedure.

C

- The type of each list item must be an array or pointer type.

C

C++

- The type of each list item must be an array, pointer, reference to array, or reference to pointer type.

C++

Fortran

- Each list item must be an array.

Fortran

Cross References

- **declare_simd** Directive, see Section 9.8
- **simd** Construct, see Section 12.4

7.13 groupprivate Directive

Name: groupprivate Category: declarative	Association: explicit Properties: pure

Arguments

groupprivate (*list*)

Name	Type	Properties
list	list of variable list item type	*default*

Clauses

device_type

Semantics

The **groupprivate** directive specifies that list items have the groupprivate attribute and therefore they are replicated such that each contention group receives its own copy. Each copy of the list item is uninitialized upon creation. The lifetime of a groupprivate variable is limited to the lifetime of all tasks in the contention group.

For a **device_type** clause that is specified implicitly or explicitly on the directive, the behavior is as if the list items appear in a **local** clause on a declare target directive on which the same **device_type** clause is specified and at the same program point.

All references to a variable in *list* in any task will refer to the groupprivate copy of that variable that is created for the contention group of the innermost enclosing implicit parallel region.

Restrictions

Restrictions to the **groupprivate** directive are as follows:

- A task that executes in a particular contention group must not access the storage of a groupprivate copy of the list item that is created for a different contention group.
- A variable that is declared with an initializer must not appear in a **groupprivate** directive.

C / C++

- Each list item must be a file-scope, namespace-scope, or static block-scope variable.
- No list item may have an incomplete type.
- The address of a groupprivate variable must not be an address constant.
- If any list item is a file-scope variable, the directive must appear outside any definition or declaration, and must lexically precede all references to any of the variables in the *list*.
- If any list item is a namespace-scope variable, the directive must appear outside any definition or declaration other than the namespace definition itself and must lexically precede all references to any of the variables in the *list*.
- Each variable in the *list* of a **groupprivate** directive at file, namespace, or class scope must refer to a variable declaration at file, namespace, or class scope that lexically precedes the directive.
- If any list item is a static block-scope variable, the directive must appear in the scope of the variable and not in a nested scope and must lexically precede all references to any of the variables in the *list*.
- Each variable in the *list* of a **groupprivate** directive in block scope must have static storage duration and must refer to a variable declaration in the same scope that lexically precedes the directive.
- If a variable is specified in a **groupprivate** directive in one compilation unit, it must be specified in a **groupprivate** directive in every compilation unit in which it is declared.

C / C++

C++

- If any list item is a static class member variable, the directive must appear in the class definition, in the same scope in which the member variable is declared, and must lexically precede all references the variable.
- A groupprivate variable must not have an incomplete type or a reference type.

C++

Fortran

- Each list item must be a named variable or a named common block; a named common block must appear between slashes.
- The *list* argument must not include any coarrays or associate names.
- The **groupprivate** directive must appear in the declaration section of a scoping unit in which the common block or variable is declared.
- If a **groupprivate** directive that specifies a common block name appears in one compilation unit, then such a directive must also appear in every other compilation unit that contains a **COMMON** statement that specifies the same name. Each such directive must appear after the last such **COMMON** statement in that compilation unit.

- If a groupprivate variable or a groupprivate common block is declared with the **BIND** attribute, the corresponding C entities must also be specified in a **groupprivate** directive in the C program.
- A variable may only appear as an argument in a **groupprivate** directive in the scope in which it is declared. It must not be an element of a common block or appear in an **EQUIVALENCE** statement.
- A variable that appears as a list item in a **groupprivate** directive must be declared in the scope of a module or have the **SAVE** attribute, either explicitly or implicitly.
- The effect of an access to a groupprivate variable in a **DO CONCURRENT** construct is unspecified.

Fortran

Cross References

- **device_type** Clause, see Section 15.1
- **local** Clause, see Section 7.14

7.14 local Clause

Name: local	**Properties:** data-environment attribute

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

declare_target

Semantics

The **local** clause specifies that each list item has the device-local attribute. A reference to a list item on a given device will refer to a copy of the list item that is a device-local variable and is in memory associated with the device.

Cross References

- **declare_target** Directive, see Section 9.9.1

8 Memory Management

This chapter defines directives, clauses and related concepts for managing memory used by OpenMP programs.

8.1 Memory Spaces

OpenMP memory spaces represent storage resources where variables can be stored and retrieved. Table 8.1 shows the list of predefined memory spaces. The selection of a given memory space expresses an intent to use storage with certain traits for the allocations. The actual storage resources that each memory space represents are implementation defined.

TABLE 8.1: Predefined Memory Spaces

Memory space name	Storage selection intent
omp_default_mem_space	Represents the system default storage
omp_large_cap_mem_space	Represents storage with large capacity
omp_const_mem_space	Represents storage optimized for variables with constant values
omp_high_bw_mem_space	Represents storage with high bandwidth
omp_low_lat_mem_space	Represents storage with low latency

Variables allocated in the **omp_const_mem_space** memory space may be initialized through the **firstprivate** clause or with compile-time constants for static and constant variables. Implementation defined mechanisms to provide the constant value of these variables may also be supported.

Restrictions

Restrictions to OpenMP memory spaces are as follows:

- Variables in the **omp_const_mem_space** memory space may not be written.

8.2 Memory Allocators

OpenMP memory allocators can be used by an OpenMP program to make allocation requests. When a memory allocator receives a request to allocate storage of a certain size, an allocation of logically contiguous memory in the resources of its associated memory space of at least the size that was requested will be returned if possible. This allocation will not overlap with any other existing allocation from a memory allocator.

If an allocator is used to allocate memory for a variable with static storage duration that is not a local static variable then the task that requested the allocation is unspecified. If an allocator is used to allocate memory for a local static variable then the task that requested the allocation is considered to be the current task of the first thread that executes code in which the variable is visible.

The behavior of the allocation process can be affected by the allocator traits that the user specifies. Table 8.2 shows the allowed allocator traits, their possible values and the default value of each trait.

TABLE 8.2: Allocator Traits

Allocator Trait	Allowed Values	Default Value
sync_hint	**contended**, **uncontended**, **serialized**, **private**	**contended**
alignment	Non-negative integer powers of 2	1 byte
access	**all**, **memspace**, **device**, **cgroup**, **pteam**, **thread**	**memspace**
pool_size	Any positive integer	Implementation defined
fallback	**default_mem_fb**, **null_fb**, **abort_fb**, **allocator_fb**	See below
fb_data	An allocator handle	(none)
pinned	*true*, *false*	*false*
partition	**environment**, **nearest**, **blocked**, **interleaved**, **partitioner**	**environment**
pin_device	Conforming device number	(none)
preferred_device	Conforming device number	(none)
target_access	**single**, **multiple**	**single**
atomic_scope	**all**, **device**	**device**

table continued on next page

table continued from previous page

Allocator Trait	Allowed Values	Default Value
part_size	Positive integer value	Implementation defined
partitioner	A memory partitioner handle	(none)
partitioner_arg	An integer value	0

The **sync_hint** trait describes the expected manner in which multiple threads may use the allocator. The values and their descriptions are:

- **contended**: high contention is expected on the allocator; that is, many tasks are expected to request allocations simultaneously;
- **uncontended**: low contention is expected on the allocator; that is, few tasks are expected to request allocations simultaneously;
- **serialized**: one task at a time will request allocations with the allocator. Requesting two allocations simultaneously when specifying **serialized** results in unspecified behavior; and
- **private**: the same thread will execute all tasks that request allocations with the allocator. Requesting an allocation from tasks that different threads execute, simultaneously or not, when specifying **private** results in unspecified behavior.

Allocated memory will be byte aligned to at least the value specified for the **alignment** trait of the allocator. Some directives and routines can specify additional requirements on alignment beyond those described in this section.

The **access** trait defines the *access group* of tasks that may access memory that is allocated by a memory allocator. If the value is **all**, the access group consists of all tasks that execute on all available devices. If the value is **memspace**, the access group consists of all tasks that execute on all devices that are associated with the allocator. If the value is **device**, the access group consists of all tasks that execute on the device where the allocation was requested. If the value is **cgroup**, the access group consists of all tasks in the same contention group as the task that requested the allocation. If the value is **pteam**, the access group consists of all current team tasks of the innermost enclosing parallel region in which the allocation was requested. If the value is **thread**, the access group consists of all tasks that are executed by the same thread that executed the allocation request. Memory returned by the allocator will be memory accessible by all tasks in the same access group as the task that requested the allocation. Attempts to access this memory from a task that is not in same access group results in unspecified behavior.

The total amount of storage in bytes that an allocator can use for allocation requests from tasks in the same access group is limited by the **pool_size** trait. Requests that would result in using more storage than **pool_size** will not be fulfilled by the allocator.

The **fallback** trait specifies how the memory allocator behaves when it cannot fulfill an allocation request. If the **fallback** trait is set to **null_fb**, the allocator returns the value zero if it fails to allocate the memory. If the **fallback** trait is set to **abort_fb**, the behavior is as if an **error** directive for which *sev-level* is **fatal** and *action-time* is **execution** is encountered if the allocation fails. If the **fallback** trait is set to **allocator_fb** then when an allocation fails the request will be delegated to the allocator specified in the **fb_data** trait. If the **fallback** trait is set to **default_mem_fb** then when an allocation fails another allocation will be tried in **omp_default_mem_space**, which assumes all allocator traits to be set to their default values except for **fallback** trait, which will be set to **null_fb**. The default value for the **fallback** trait is **null_fb** for any allocator that is associated with a target memory space. Otherwise, the default value is **default_mem_fb**.

All memory that is allocated with an allocator for which the **pinned** trait is specified as *true* must remain in the same storage resource at the same location for its entire lifetime. If **pin_device** is also specified then the allocation must be allocated in that device.

The **partition** trait describes the partitioning of allocated memory over the storage resources represented by the memory space associated with the allocator. The partitioning will be done in parts with a minimum size that is implementation defined. The values are:

- **environment**: the placement of allocated memory is determined by the execution environment;
- **nearest**: allocated memory is placed in the storage resource that is nearest to the thread that requests the allocation;
- **blocked**: allocated memory is partitioned into parts of approximately the same size with at most one part per storage resource; and
- **interleaved**: allocated memory parts are distributed in a round-robin fashion across the storage resources such that the size of each part is the value of the **part_size** trait except possibly the last part, which can be smaller.
- **partitioner**: the number of memory parts and how they are distributed across the storage are defined by the memory partition object created by the memory partitioner specified by the **partitioner** trait.

The **part_size** trait specifies the size of the parts allocated over the storage resources for some of the memory partition trait policies. The actual value of the trait might be rounded up to an implementation defined value to comply with hardware restrictions of the storage resources.

If the **preferred_device** trait is specified then storage resources of the specified device are preferred to fulfill the allocation.

If the value of the **target_access** trait is **single** then data from this allocator cannot be accessed on two different devices unless, for any given host device access, the entry and exit of the **target** region in which any accesses occur either both precede or both follow the host device access in happens-before order. Additionally, for any two **target** regions that may access data from this allocator and execute on distinct devices, the entry and exit of one of the regions must precede those of the other in happens-before order. If the value of the **target_access** trait is **multiple** then accesses of data from this allocator from different devices may be arbitrarily interleaved, provided that synchronization ensures data races do not occur.

If the value of the **atomic_scope** trait is **all** then all storage locations of data from this allocator have an atomic scope that consists of all threads on the devices associated with the allocator. If the value is **device** then all storage locations have an atomic scope that consists of all threads on the device on which the atomic operation is performed.

Table 8.3 shows the list of predefined memory allocators and their associated memory spaces. The predefined memory allocators have default values for their allocator traits unless otherwise specified.

TABLE 8.3: Predefined Allocators

Allocator Name	Associated Memory Space	Non-Default Trait Values
omp_default_mem_alloc	**omp_default_mem_space**	**fallback:null_fb**
omp_large_cap_mem_alloc	**omp_large_cap_mem_space**	(none)
omp_const_mem_alloc	**omp_const_mem_space**	(none)
omp_high_bw_mem_alloc	**omp_high_bw_mem_space**	(none)
omp_low_lat_mem_alloc	**omp_low_lat_mem_space**	(none)
omp_cgroup_mem_alloc	Implementation defined	**access:cgroup**
omp_pteam_mem_alloc	Implementation defined	**access:pteam**
omp_thread_mem_alloc	Implementation defined	**access:thread**

Fortran

If any operation of the base language causes a reallocation of a variable that is allocated with a memory allocator then that memory allocator will be used to deallocate the current memory and to allocate the new memory. For any allocatable subcomponents, the allocator that is used for the deallocation and allocation is unspecified.

Fortran

Restrictions

- If the **pin_device** trait is specified, its value must be the device number of a device associated with the memory allocator.
- If the **preferred_device** trait is specified, its value must be the device number of a device associated with the memory allocator.

- The **omp_cgroup_mem_alloc**, **omp_pteam_mem_alloc**, and **omp_thread_mem_alloc** predefined memory allocators must not be used to allocate a variable with static storage duration unless the variable is a local static variable.

8.3 align Clause

Name: align	**Properties:** unique

Arguments

Name	Type	Properties
alignment	expression of integer type	constant, positive

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

allocate

Semantics

The **align** clause is used to specify the byte alignment to use for allocations associated with the construct on which the clause appears. Specifically, each allocation is byte aligned to at least the maximum of the value to which *alignment* evaluates, the **alignment** trait of the allocator being used for the allocation, and the alignment required by the base language for the type of the variable that is allocated. On constructs on which the clause may appear, if it is not specified then the effect is as if it was specified with the **alignment** trait of the allocator being used for the allocation.

Restrictions

Restrictions to the **align** clause are as follows:

- *alignment* must evaluate to a power of two.

Cross References

- **allocate** Directive, see Section 8.5
- Memory Allocators, see Section 8.2

8.4 `allocator` Clause

Name: `allocator`	**Properties:** ICV-defaulted, unique

Arguments

Name	Type	Properties
allocator	expression of allocator_-handle type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`allocate`

Semantics

The **`allocator`** clause specifies the memory allocator to be used for allocations associated with the construct on which the clause appears. Specifically, the allocator to which *allocator* evaluates is used for the allocations. On constructs on which the clause may appear, if it is not specified then the effect is as if it was specified with the value of the *def-allocator-var* ICV.

Cross References

- **`allocate`** Directive, see Section 8.5
- Memory Allocators, see Section 8.2
- *def-allocator-var* ICV, see Table 3.1

8.5 `allocate` Directive

Name: `allocate` **Category:** declarative	**Association:** explicit **Properties:** pure

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Clauses

`align`, **`allocator`**

Semantics

The storage for each list item that appears in the **allocate** directive is provided an allocation through the memory allocator as determined by the **allocator** clause with an alignment as determined by the **align** clause. The scope of this allocation is that of the list item in the base language. At the end of the scope for a given list item the memory allocator used to allocate that list item deallocates the storage.

For allocations that arise from this directive the **null_fb** value of the fallback allocator trait behaves as if the **abort_fb** had been specified.

Restrictions

Restrictions to the **allocate** directive are as follows:

- An **allocate** directive must appear in the same scope as the declarations of each of its list items and must follow all such declarations.
- A declared variable may appear as a list item in at most one **allocate** directive in a given compilation unit.
- **allocate** directives that appear in a **target** region must specify an **allocator** clause unless a **requires** directive with the **dynamic_allocators** clause is present in the same compilation unit.

C / C++

- If a list item has static storage duration, the **allocator** clause must be specified and the *allocator* expression in the clause must be a constant expression that evaluates to one of the predefined memory allocator values.
- A variable that is declared in a namespace or global scope may only appear as a list item in an **allocate** directive if an **allocate** directive that lists the variable follows a declaration that defines the variable and if all **allocate** directives that list it specify the same allocator.
- A list item must not be a function parameter.

C / C++

C

- After a list item has been allocated, the scope that contains the **allocate** directive must not end abnormally, such as through a call to the **longjmp** function.

C

C++

- After a list item has been allocated, the scope that contains the **allocate** directive must not end abnormally, such as through a call to the **longjmp** function, other than through C++ exceptions.
- A variable that has a reference type must not appear as a list item in an **allocate** directive.

C++

Fortran

- A list item that is specified in an **allocate** directive must not be a coarray or have a coarray as an ultimate component, or have the **ALLOCATABLE**, or **POINTER** attribute.
- If a list item has the **SAVE** attribute, either explicitly or implicitly, or is a common block name then the **allocator** clause must be specified and only predefined memory allocator parameters can be used in the clause.
- A variable that is part of a common block must not be specified as a list item in an **allocate** directive, except implicitly via the named common block.
- A named common block may appear as a list item in at most one **allocate** directive in a given compilation unit.
- If a named common block appears as a list item in an **allocate** directive, it must appear as a list item in an **allocate** directive that specifies the same allocator in every compilation unit in which the common block is used.
- An associate name must not appear as a list item in an **allocate** directive.
- A list item must not be a dummy argument.

Fortran

Cross References

- **align** Clause, see Section 8.3
- **allocator** Clause, see Section 8.4
- Memory Allocators, see Section 8.2

8.6 allocate Clause

Name: allocate	**Properties:** all-privatizing

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
allocator-simple-modifier	*list*	expression of OpenMP allocator_handle type	exclusive, unique
allocator-complex-modifier	*list*	Complex, name: **allocator** Arguments: ***allocator*** expression of allocator_handle type (*default*)	unique
align-modifier	*list*	Complex, name: **align** Arguments: ***alignment*** expression of integer type (constant, positive)	unique
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

allocators, **distribute**, **do**, **for**, **parallel**, **scope**, **sections**, **single**, **target**, **target_data**, **task**, **taskgroup**, **taskloop**, **teams**

Semantics

The **allocate** clause specifies the memory allocator to be used to obtain storage for a variable list. If a list item in the clause also appears in a data-sharing attribute clause on the same directive that privatizes the list item, allocations that arise from that list item in the clause will be provided by the memory allocator. If the *allocator-simple-modifier* is specified, the behavior is as if the *allocator-complex-modifier* is instead specified with *allocator-simple-modifier* as its *allocator* argument. The *allocator-complex-modifier* and *align-modifier* have the same syntax and semantics for the **allocate** clause as the **allocator** and **align** clauses have for the **allocate** directive. For allocations that arise from this clause, the **null_fb** value of the fallback allocator trait behaves as if the **abort_fb** value had been specified.

Restrictions

Restrictions to the **allocate** clause are as follows:

- For any list item that is specified in the **allocate** clause on a directive other than the **allocators** directive, a data-sharing attribute clause that may create a private copy of that list item must be specified on the same directive.
- For **task**, **taskloop** or **target** directives, allocation requests to memory allocators with the **access** trait set to **thread** result in unspecified behavior.
- **allocate** clauses that appear on a **target** construct or on constructs in a **target** region must specify an *allocator-simple-modifier* or *allocator-complex-modifier* unless a

`requires` directive with the `dynamic_allocators` clause is present in the same compilation unit.

Cross References

- `align` Clause, see Section 8.3
- `allocator` Clause, see Section 8.4
- `allocators` Construct, see Section 8.7
- `distribute` Construct, see Section 13.7
- `do` Construct, see Section 13.6.2
- `for` Construct, see Section 13.6.1
- Memory Allocators, see Section 8.2
- `parallel` Construct, see Section 12.1
- `scope` Construct, see Section 13.2
- `sections` Construct, see Section 13.3
- `single` Construct, see Section 13.1
- `target` Construct, see Section 15.8
- `target_data` Construct, see Section 15.7
- `task` Construct, see Section 14.1
- `taskgroup` Construct, see Section 17.4
- `taskloop` Construct, see Section 14.2
- `teams` Construct, see Section 12.2

Fortran

8.7 allocators Construct

Name: **allocators**	Association: block : allocator
Category: executable	**Properties:** *default*

Clauses

allocate

Semantics

The **allocators** construct specifies that if a variable that is to be allocated by the associated *allocate-stmt*, appears as a list item in an **allocate** clause on the directive an allocator is used to allocate storage for the variable according to the semantics of the **allocate** clause. If a variable that is to be allocated does not appear as a list item in an **allocate** clause, the allocation is performed according to the base language implementation. The list items that appear in an **allocate** clause may include structure elements.

Restrictions

Restrictions to the **allocators** construct are as follows:

- A list item that appears in an **allocate** clause must appear as one of the variables that is allocated by the *allocate-stmt* in the associated allocator structured block.
- A list item must not be a coarray or have a coarray as an ultimate component.

Cross References

- **allocate** Clause, see Section 8.6
- Memory Allocators, see Section 8.2
- OpenMP Allocator Structured Blocks, see Section 6.3.1

Fortran

8.8 uses_allocators Clause

Name: **uses_allocators**	**Properties:** data-environment attribute, data-sharing attribute

Arguments

Name	Type	Properties
allocator	expression of allocator_-handle type	*default*

Modifiers

Name	Modifies	Type	Properties
mem-space	*allocator*	Complex, name: **memspace** Arguments: ***memspace-handle*** expression of memspace_handle type (*default*)	*default*
traits-array	*allocator*	Complex, name: **traits** Arguments: ***traits*** variable of alloctrait array type (*default*)	*default*
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

target

Semantics

The **uses_allocators** clause enables the use of the specified *allocator* in the region associated with the directive on which the clause appears. The clause has no effect for an *allocator* argument value of **omp_null_allocator**. If *allocator* is an identifier that matches the name of a predefined allocator (see Table 8.3), that predefined allocator will be available for use in the region. Otherwise, the effect is as if *allocator* is specified on a **private** clause. The resulting corresponding list item is assigned the result of a call to **omp_init_allocator** at the beginning of the associated region with arguments *memspace-handle*, the number of traits in the *traits* array, and *traits*. If *mem-space* is not specified or **omp_null_mem_space** is specified, the effect is as if *memspace-handle* is specified as **omp_default_mem_space**. If *traits-array* is not specified, the effect is as if *traits* is specified as an empty array. Further, at the end of the associated region, the effect is as if this allocator is destroyed as if by a call to **omp_destroy_allocator**.

More than one *clause-argument-specification* may be specified.

Restrictions

- The *allocator* expression must be a base language identifier.
- If *allocator* is an identifier that matches the name of a predefined allocator, no modifiers may be specified.
- If *allocator* is not the name of a predefined allocator and is not **omp_null_allocator**, it must be a variable.
- The *allocator* argument must not appear in other data-sharing attribute clauses or data-mapping attribute clauses on the same construct.

C / C++

- The *traits* argument for the *traits-array* modifier must be a constant array, have constant values and be defined in the same scope as the construct on which the clause appears.

C / C++

Fortran

- The *traits* argument for the *traits-array* modifier must be a named constant of rank one.

Fortran

- The *memspace-handle* argument for the *mem-space* modifier must be an identifier that matches one of the predefined memory space names.

Cross References

- OpenMP **allocator_handle** Type, see Section 20.8.1
- OpenMP **alloctrait** Type, see Section 20.8.2
- Memory Allocators, see Section 8.2
- Memory Spaces, see Section 8.1
- OpenMP **memspace_handle** Type, see Section 20.8.11
- **omp_destroy_allocator** Routine, see Section 27.7
- **omp_init_allocator** Routine, see Section 27.6
- **target** Construct, see Section 15.8

9 Variant Directives

This chapter defines directives and related concepts to support the seamless adaption of OpenMP programs to OpenMP contexts.

9.1 OpenMP Contexts

At any point in an OpenMP program, an OpenMP context exists that defines traits that describe the active constructs, the execution devices, functionality supported by the implementation and available dynamic values. The traits are grouped into trait sets. The defined trait sets are: the construct trait set; the device trait set; the target device trait set; the implementation trait set; and the dynamic trait set. Traits are categorized as name-list traits, clause-list traits, non-property traits and extension traits. This categorization determines the syntax that is used to match the trait, as defined in Section 9.2.

The construct trait set is composed of the directive names, each being a trait, of all enclosing constructs at that point in the OpenMP program up to a **`target`** construct. Compound constructs are added to the set as their leaf constructs in the same nesting order specified by the original constructs. The **`dispatch`** construct is added to the construct trait set only for the *target-call* of the associated function-dispatch structured block. The construct trait set is ordered by nesting level in ascending order. Specifically, the ordering of the set of constructs is $c_1, \ldots, c_N$, where c_1 is the construct at the outermost nesting level and c_N is the construct at the innermost nesting level. In addition, if the point in the OpenMP program is not enclosed by a **`target`** construct, the following rules are applied in order:

1. For procedures with a **`declare_simd`** directive, the *simd* trait is added to the beginning of the construct trait set as c_1 for any generated SIMD versions so the total size of the trait set is increased by one.
2. For procedures that are determined to be function variants by a declare variant directive, the trait selectors $c_1, \ldots, c_M$ of the **`construct`** selector set are added in the same order to the beginning of the construct trait set as $c_1, \ldots, c_M$ so the total size of the trait set is increased by M.
3. For procedures that are determined to be target variants by a declare target directive, the *target* trait is added to the beginning of the construct trait set as c_1 so the total size of the trait set is increased by one.

The *simd* trait is a clause-list trait that is defined with properties that match the clauses that can be specified on the **`declare_simd`** directive with the same names and semantics. The *simd* trait defines at least the *simdlen* property and one of the *inbranch* or *notinbranch* properties. Traits in the construct trait set other than *simd* are non-property traits.

The device trait set includes traits that define the characteristics of the device that the compiler determines will be the current device during program execution at a given point in the OpenMP program. A trait in the device trait set is considered to be active at program points that fall outside a defined procedure if it defines a characteristic of some available device, including the host device. For each target device that the implementation supports, a target device trait set exists that defines the characteristics of that device. At least the following traits must be defined for the device trait set and all target device trait sets:

- The *kind(kind-list)* name-list trait specifies the general kind of the device. Each member of *kind-list* is a *kind-name*, for which the following values are defined:
  - *host*, which specifies that the device is the host device;
  - *nohost*, which specifies that the device is not the host device; and
  - the values defined in the OpenMP Additional Definitions document.
- The *isa(isa-list)* name-list trait specifies the Instruction Set Architectures supported by the device. Each member of *isa-list* is an *isa-name*, for which the accepted values are implementation defined.
- The *arch(arch-list)* name-list trait specifies the architectures supported by the device. Each member of *arch-list* is an *arch-name*, for which the accepted values are implementation defined.

The target device trait set also defines the following traits:

- The *device_num* trait specifies the *device number* of the device.
- The *uid* trait specifies a unique identifier string of the device, for which the accepted values are implementation defined.

The implementation trait set includes traits that describe the functionality supported by the OpenMP implementation at that point in the OpenMP program. At least the following traits can be defined:

- The *vendor(vendor-list)* name-list trait, which specifies the vendor identifiers of the implementation. Each member of *vendor-list* is a *vendor-name*, for which the defined values are in the OpenMP Additional Definitions document.
- The *extension(extension-list)* name-list trait, which specifies vendor-specific extensions to the OpenMP specification. Each member of *extension-list* is an *extension-name*, for which the accepted values are implementation defined.
- A *requires(requires-list)* clause-list trait, for which the properties are the clauses that have been supplied to the **`requires`** directive prior to the program point as well as implementation defined implicit requirements.

Implementations can define additional traits in the device trait set, target device trait set and implementation trait set; these traits are extension traits.

The dynamic trait set includes traits that define the dynamic properties of an OpenMP program at a point in its execution. The *data state* trait in the dynamic trait set refers to the complete data state of the OpenMP program that may be accessed at runtime.

9.2 Context Selectors

Context selectors are used to define the properties that can match an OpenMP context. OpenMP defines different trait selector sets, each of which contains different trait selectors.

The syntax for a context selector is *context-selector-specification* as described in the following grammar:

```
context-selector-specification:
    trait-set-selector[,trait-set-selector[,...]]

trait-set-selector:
    trait-set-selector-name={trait-selector[, trait-selector[, ...]]}

trait-selector:
    trait-selector-name[([trait-score: ] trait-property[, trait-property[, ...]])]

trait-property:
    trait-property-name
    trait-property-clause
    trait-property-expression
    trait-property-extension

trait-property-clause:
    clause

trait-property-name:
    identifier
    string-literal

trait-property-expression
    scalar-expression (for C/C++)
    scalar-logical-expression (for Fortran)
    scalar-integer-expression (for Fortran)

trait-score:
    score(score-expression)

trait-property-extension:
    trait-property-name
```

identifier **(***trait-property-extension*[**,** *trait-property-extension*[**,** *...*]]**)**
constant integer expression

For trait selectors that correspond to name-list traits, each *trait-property* should be *trait-property-name* and, for any value that is a valid identifier, both the identifier and the corresponding string literal (for C/C++) and the corresponding *char-literal-constant* (for Fortran) representation are considered representations of the same value.

For trait selectors that correspond to clause-list traits, each *trait-property* should be *trait-property-clause*. The syntax is the same as for the matching clause.

The **construct** selector set defines the traits in the construct trait set that should be active in the OpenMP context. Each trait selector that can be defined in the **construct** selector set is the *directive-name* of a context-matching construct. Each *trait-property* of the **simd** trait selector is a *trait-property-clause*. The syntax is the same as for a valid clause of the **declare_simd** directive and the restrictions on the clauses from that directive apply. The **construct** selector set is an ordered list $c_1, \ldots, c_N$.

The **device** selector set and **implementation** selector set define the traits that should be active in the corresponding trait set of the OpenMP context. The **target_device** selector set defines the traits that should be active in the target device trait set for the device that the specified **device_num** trait selector identifies. The same traits that are defined in the corresponding trait sets can be used as trait selectors with the same properties. The **kind** trait selector of the **device** selector set and **target_device** selector set can also specify the value **any**, which is as if no **kind** trait selector was specified. If a **device_num** trait selector does not appear in the **target_device** selector set then a **device_num** trait selector that specifies the value of the *default-device-var* ICV is implied. For the **device_num** trait selector of the **target_device** selector set, a single *trait-property-expression* must be specified. The **device_num** trait selector can be *true* only if that *trait-property-expression* evaluates to a conforming device number other than **omp_invalid_device**. For the **atomic_default_mem_order** trait selector of the **implementation** selector set, a single *trait-property* must be specified as an identifier equal to one of the valid arguments to the **atomic_default_mem_order** clause on the **requires** directive. For the **requires** trait selector of the **implementation** selector set, each *trait-property* is a *trait-property-clause*. The syntax is the same as for a valid clause of the **requires** directive and the restrictions on the clauses from that directive apply.

The **user** selector set defines the **condition** trait selector that provides additional user-defined conditions. The **condition** trait selector contains a single *trait-property-expression* that must evaluate to *true* for the trait selector to be *true*. Any non-constant *trait-property-expression* that is evaluated to determine the suitability of a variant is evaluated according to the *data state* trait in the dynamic trait set of the OpenMP context. The **user** selector set is dynamic if the **condition** trait selector is present and the expression in the **condition** trait selector is not a constant expression; otherwise, it is static.

All parts of a context selector define the static part of the context selector except the following parts, which define the dynamic part of the context selector:

- Its **user** selector set if it is dynamic; and
- Its **target_device** selector set.

For the **match** clause of a **declare_variant** directive, any argument of the base function that is referenced in an expression that appears in the context selector is treated as a reference to the expression that is passed into that argument at the call to the base function. Otherwise, a variable or procedure reference in an expression that appears in a context selector is a reference to the variable or procedure of that name that is visible at the location of the directive on which the context selector appears.

C++

Each occurrence of the **this** pointer in an expression in a context selector that appears in the **match** clause of a **declare_variant** directive is treated as an expression that is the address of the object on which the associated base function is invoked.

C++

Implementations can allow further trait selectors to be specified. Each specified *trait-property* for these implementation defined trait selectors should be a *trait-property-extension*. Implementations can ignore specified trait selectors that are not those described in this section.

Restrictions

Restrictions to context selectors are as follows:

- Each *trait-property* may only be specified once in a trait selector other than those in the **construct** selector set.
- Each *trait-set-selector-name* may only be specified once in a context selector.
- Each *trait-selector-name* may only be specified once in a trait selector set.
- A *trait-score* cannot be specified in traits from the **construct** selector set, the **device** selector set or the **target_device** selector sets.
- A *score-expression* must be a non-negative constant integer expression.
- The expression of a **device_num** trait must evaluate to a conforming device number.
- A variable or procedure that is referenced in an expression that appears in a context selector must be visible at the location of the directive on which the context selector appears unless the directive is a **declare_variant** directive and the variable is an argument of the associated base function.
- If *trait-property* **any** is specified in the **kind** *trait-selector* of the **device** selector set or the **target_device** selector sets, no other *trait-property* may be specified in the same selector set.
- For a *trait-selector* that corresponds to a name-list trait, at least one *trait-property* must be specified.

- For a *trait-selector* that corresponds to a non-property trait, no *trait-property* may be specified.
- For the **requires** trait selector of the **implementation** selector set, at least one *trait-property* must be specified.

9.3 Matching and Scoring Context Selectors

A compatible context selector for an OpenMP context satisfies the following conditions:

- All trait selectors in its **user** selector set are true;
- All traits and trait properties that are defined by trait selectors in the **target_device** selector set are active in the target device trait set for the device that is identified by the **device_num** trait selector;
- All traits and trait properties that are defined by trait selectors in its **construct** selector set, its **device** selector set and its **implementation** selector set are active in the corresponding trait sets of the OpenMP context;
- For each trait selector in the context selector, its properties are a subset of the properties of the corresponding trait of the OpenMP context; and
- Trait selectors in its **construct** selector set appear in the same relative order as their corresponding traits in the construct trait set of the OpenMP context;

Some properties of the **simd** trait selector have special rules to match the properties of the *simd* trait:

- The **simdlen(** *N* **)** property of the trait selector matches the *simdlen(M)* trait of the OpenMP context if M is a multiple of N; and
- The **aligned(** *list:N* **)** property of the trait selector matches the *aligned(list:M)* trait of the OpenMP context if N is a multiple of M.

Among compatible context selectors, a score is computed using the following algorithm:

1. Each trait selector for which the corresponding trait appears in the construct trait set in the OpenMP context is given the value 2^{p-1} where p is the position of the corresponding trait, c_p, in the construct trait set; if the traits that correspond to the **construct** selector set appear multiple times in the OpenMP context, the highest valued subset of context traits that contains all trait selectors in the same order are used;
2. The **kind**, **arch**, and **isa** trait selectors, if specified, are given the values 2^l, 2^{l+1} and 2^{l+2}, respectively, where l is the number of traits in the construct trait set;
3. Trait selectors for which a *trait-score* is specified are given the value specified by the *trait-score score-expression*;

4. The values given to any additional trait selectors allowed by the implementation are implementation defined;
5. Other trait selectors are given a value of zero; and
6. A context selector that is a strict subset of another compatible context selector has a score of zero. For other context selectors, the final score is the sum of the values of all specified trait selectors plus 1.

9.4 Metadirectives

A metadirective is a directive that can specify multiple directive variants of which one may be conditionally selected to replace the metadirective based on the enclosing context. A metadirective is replaced by a **nothing** directive or one of the directive variants specified by the **when** clauses or the **otherwise** clause. If no **otherwise** clause is specified the effect is as if one was specified without an associated directive variant.

The OpenMP context for a given metadirective is defined according to Section 9.1. The order of clauses that appear on a metadirective is significant and, if specified, **otherwise** must be the last clause specified on a metadirective.

Replacement candidates for a metadirective are ordered according to the following rules in decreasing precedence:

- A candidate is before another one if the score associated with the context selector of the corresponding **when** clause is higher.
- A candidate that was explicitly specified is before one that was implicitly specified.
- Candidates are ordered according to the order in which they lexically appear on the metadirective.

The list of dynamic replacement candidates is the prefix of the sorted list of replacement candidates up to and including the first candidate for which the corresponding **when** or **otherwise** clause has a static context selector. The first dynamic replacement candidate for which the corresponding **when** or **otherwise** clause has a compatible context selector, according to the matching rules defined in Section 9.3, replaces the metadirective.

Restrictions

Restrictions to metadirectives are as follows:

- Replacement of the metadirective with the directive variant associated with any of the dynamic replacement candidates must result in a conforming program.
- Insertion of user code at the location of a metadirective must be allowed if the first dynamic replacement candidate does not have a static context selector.
- If the list of dynamic replacement candidates has multiple items then all items must be executable directives.

Fortran

- A metadirective that appears in the specification part of a subprogram must follow all variant-generating directives that appear in the same specification part.
- A metadirective is pure if and only if all directive variants specified for it are pure.

Fortran

9.4.1 `when` Clause

Name: `when`	**Properties:** *default*

Arguments

Name	Type	Properties
directive-variant	directive-specification	optional, unique

Modifiers

Name	Modifies	Type	Properties
context-selector	*directive-variant*	An OpenMP context-selector-specification	required, unique
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`begin metadirective`, **`metadirective`**

Semantics

The specified *directive-variant* is a replacement candidate for the metadirective on which the clause is specified if the static part of the context selector specified by *context-selector* is compatible with the OpenMP context according to the matching rules defined in Section 9.3. If a **`when`** clause does not explicitly specify a directive variant, it implicitly specifies a **`nothing`** directive as the directive variant.

Expressions that appear in the context selector of a **`when`** clause are evaluated if no prior dynamic replacement candidate has a compatible context selector, and the number of times each expression is evaluated is implementation defined. All variables referenced by these expressions are considered to be referenced by the metadirective.

A directive variant that is associated with a **`when`** clause can only affect the OpenMP program if the directive variant is a dynamic replacement candidate.

Restrictions

Restrictions to the **`when`** clause are as follows:

- *directive-variant* must not specify a metadirective.
- *context-selector* must not specify any properties for the **`simd`** trait selector.

C / C++

- *directive-variant* must not specify a **begin declare_variant** directive.

C / C++

Cross References

- **begin metadirective**, see Section 9.4.4
- Context Selectors, see Section 9.2
- **metadirective**, see Section 9.4.3
- **nothing** Directive, see Section 10.7

9.4.2 otherwise Clause

Name: otherwise	Properties: unique, ultimate

Arguments

Name	Type	Properties
directive-variant	directive-specification	optional, unique

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

begin metadirective, **metadirective**

Semantics

The **otherwise** clause is treated as a **when** clause with the specified directive variant, if any, and a static context selector that is always compatible and has a score lower than the scores associated with any other directive variant.

Restrictions

Restrictions to the **otherwise** clause are as follows:

- *directive-variant* must not specify a metadirective.

C / C++

- *directive-variant* must not specify a **begin declare_variant** directive.

C / C++

Cross References

- **begin metadirective**, see Section 9.4.4
- **metadirective**, see Section 9.4.3
- **when** Clause, see Section 9.4.1

9.4.3 metadirective

Name: metadirective Category: meta	Association: unassociated Properties: pure

Clauses

otherwise, **when**

Semantics

The **metadirective** specifies metadirective semantics.

Cross References

- Metadirectives, see Section 9.4
- **otherwise** Clause, see Section 9.4.2
- **when** Clause, see Section 9.4.1

9.4.4 begin metadirective

Name: begin metadirective Category: meta	Association: delimited Properties: pure

Clauses

otherwise, **when**

Semantics

The **begin metadirective** is a metadirective that is a delimited directive and for which the specified directive variants other than the **nothing** directive must accept a paired end directive. For any directive variant that is selected to replace the **begin metadirective** directive, the required paired end directive is implicitly replaced by the end directive of the directive variant to demarcate the statements that are associated with the directive variant. If the **nothing** directive is selected to replace the **begin metadirective** directive, the end directive is ignored.

Restrictions

The restrictions to **begin metadirective** are as follows:

- Any *directive-variant* that is specified by a **when** or **otherwise** clause must be a directive that has a paired end directive or must be the **nothing** directive.

Cross References

- Metadirectives, see Section 9.4
- **nothing** Directive, see Section 10.7
- **otherwise** Clause, see Section 9.4.2
- **when** Clause, see Section 9.4.1

9.5 Semantic Requirement Set

The semantic requirement set of each task is a logical set of elements that can be added to or removed from the set by different directives in the scope of the task region, as well as affect the semantics of those directives.

A directive can add the following elements to the set:

- *depend*, which specifies that a construct requires enforcement of the synchronization relationship expressed by the **depend** clause;
- *nowait*, which specifies that a construct is asynchronous;
- *is_device_ptr(list-item)*, which specifies that the *list-item* is a device pointer in a construct;
- *has_device_addr(list-item)*, which specifies that the *list-item* has a device address in a construct; and
- *interop(list-item)*, which specifies that the *list-item* is a user-provided interoperability object to be used in a construct. The order in which the *interop* elements are added is relevant.

If an implementation supports the **unified_address** requirement then:

- Adding an *is_device_ptr* element for a list item also adds a *has_device_addr* element for any data entity for which the list item is a base pointer; and
- Adding a *has_device_addr* element for a list item that has a base pointer also adds an *is_device_ptr* element for that base pointer if the base pointer is an identifier.

The following directives may add elements to the set:

- **dispatch**.

The following directives may remove elements from the set:

- **declare_variant**

Cross References

- **dispatch** Construct, see Section 9.7
- Declare Variant Directives, see Section 9.6

9.6 Declare Variant Directives

Declare variant directives declare base functions to have the specified function variant. The context selector specified by *context-selector* in the **match** clause is associated with the function variant. The OpenMP context for a direct call to a given base function is defined according to Section 9.1.

For a function variant to be a replacement candidate to be called instead of the base function, its declare variant directive for the base function must be visible at the call site and the static part of its associated context selector must be compatible with the OpenMP context of the call according to the matching rules defined in Section 9.3. In addition, if the base function is called from a non-host device, the declare variant directive must not specify an **append_args** clause or an **adjust_args** clause with a **need_device_ptr** or **need_device_addr** *adjust-op*.

Replacement candidates are ordered in decreasing order of the score associated with the context selector. If two replacement candidates have the same score then their order is implementation defined.

The list of dynamic replacement candidates is the prefix of the sorted list of replacement candidates up to and including the first candidate for which the corresponding **match** clause has a static context selector.

The first dynamic replacement candidate for which the corresponding **match** clause has a compatible context selector is called instead of the base function. If no compatible candidate exists then the base function is called.

Expressions that appear in the context selector of a **match** clause are evaluated if no prior dynamic replacement candidate has a compatible context selector, and the number of times each expression is evaluated is implementation defined. All variables referenced by these expressions are considered to be referenced at the call site.

C++

For calls to **constexpr** base functions that are evaluated in constant expressions, whether variant substitution occurs is implementation defined.

C++

For indirect function calls that can be determined to call a particular base function, whether variant substitution occurs is unspecified.

Any differences that the specific OpenMP context requires in the prototype of the function variant from the base function prototype are implementation defined.

Different declare variant directives may be specified for different declarations of the same base function.

Restrictions

Restrictions to declare variant directives are as follows:

- Calling procedures that a declare variant directive determined to be a function variant directly in an OpenMP context that is different from the one that the **construct** selector set of the context selector specifies is non-conforming.
- If a procedure is determined to be a function variant through more than one declare variant directive then the **construct** selector set of their context selectors must be the same.
- A procedure determined to be a function variant may not be specified as a base function in another declare variant directive.

- An **adjust_args** clause or **append_args** clause may only be specified if the **dispatch** trait selector of the **construct** selector set appears in the **match** clause.

C / C++

- The type of the function variant must be compatible with the type of the base function after the implementation defined transformation for its OpenMP context.

C / C++

C++

- Declare variant directives may not be specified for virtual, defaulted or deleted functions.
- Declare variant directives may not be specified for constructors or destructors.
- Declare variant directives may not be specified for immediate functions.
- The procedure that a declare variant directive determined to be a function variant may not be an immediate function.

C++

Fortran

- The characteristic of the function variant must be compatible with the characteristic of the base function after the implementation defined transformation for its OpenMP context.

Fortran

Cross References

- Context Selectors, see Section 9.2
- OpenMP Contexts, see Section 9.1

9.6.1 match Clause

Name: match	**Properties:** unique, required

Arguments

Name	Type	Properties
context-selector	An OpenMP context-selector-specification	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

begin declare_variant, **declare_variant**

Semantics

The *context-selector* argument of the **match** clause specifies the context selector to use to determine if a specified function variant is a replacement candidate for the specified base function in a given OpenMP context.

Restrictions

Restrictions to the **match** clause are as follows:

- All variables that are referenced in an expression that appears in the context selector of a **match** clause must be accessible at each call site to the base function according to the base language rules.

Cross References

- **begin declare_variant** Directive, see Section 9.6.5
- **declare_variant** Directive, see Section 9.6.4
- Context Selectors, see Section 9.2

9.6.2 adjust_args Clause

Name: **adjust_args**	**Properties:** *default*

Arguments

Name	Type	Properties
parameter-list	list of parameter list item type	*default*

Modifiers

Name	Modifies	Type	Properties
adjust-op	*parameter-list*	Keyword: **need_device_addr**, **need_device_ptr**, **nothing**	required
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

declare_variant

Semantics

The **adjust_args** clause specifies how to adjust the arguments of the base function when a specified function variant is selected for replacement in the context of a function-dispatch structured block. For each **adjust_args** clause that is present on the selected function variant, the adjustment operation specified by the *adjust-op* modifier is applied to each argument specified in the clause before being passed to the selected function variant. Any argument specified in the clause that does not exist at a given function call site is ignored.

If the *adjust-op* modifier is **nothing**, the argument is passed to the selected function variant without being modified.

If the *adjust-op* modifier is **need_device_ptr**, the arguments are converted to corresponding device pointers of the default device if they are not already device pointers. If the current task has the *is_device_ptr* element for a given argument in its semantic requirement set (as added by the **dispatch** construct that encloses the call to the base function), the argument is not adjusted. Otherwise, the argument is converted in the same manner that a **use_device_ptr** clause on a **target_data** construct converts its pointer list items into device pointers, except that if the argument cannot be converted into a device pointer then NULL is passed as the argument.

If the *adjust-op* modifier is **need_device_addr**, the arguments are replaced with references to the corresponding objects in the device data environment of the default device if they do not already have device addresses. If the current task has a *has_device_addr* element for a given argument in its semantic requirement set, as added by the **dispatch** construct that encloses the call to the base function, the argument is not adjusted. Otherwise, the argument is converted in the same manner that a **use_device_addr** clause on a **target_data** construct replaces references to the list items.

Restrictions

- If the **need_device_addr** *adjust-op* modifier is present and the *has-device-addr* element does not exist for a specified argument in the semantic requirement set of the current task, all restrictions that apply to a list item in a **use_device_addr** clause also apply to the corresponding argument that is passed by the call.

C

- If the **need_device_ptr** *adjust-op* modifier is present, each list item that appears in the clause that refers to a specific named argument in the declaration of the function variant must be of pointer type.
- The **need_device_addr** *adjust-op* modifier must not be specified in the clause.

C

C++

- If the **need_device_ptr** *adjust-op* modifier is present, each list item that appears in the clause that refers to a specific named argument in the declaration of the function variant must be of pointer type or reference to pointer type.
- If the **need_device_addr** *adjust-op* modifier is present, each list item that appears in the clause must refer to an argument in the declaration of the function variant that has a reference type.

C++

Fortran

- If the **need_device_ptr** *adjust-op* modifier is present, each list item that appears in the clause must refer to a dummy argument of **C_PTR** type in the declaration of the function variant.
- If the **need_device_addr** *adjust-op* modifier is present, each list item that appears in the clause must refer to a dummy argument in the declaration of the function variant that does not have the **VALUE** attribute.
- If the **need_device_addr** *adjust-op* modifier is present, the corresponding actual argument for each specified argument must be contiguous.

Fortran

Cross References

- **declare_variant** Directive, see Section 9.6.4
- **use_device_addr** Clause, see Section 7.5.10
- **use_device_ptr** Clause, see Section 7.5.8

9.6.3 append_args Clause

Name: **append_args**	**Properties:** unique

Arguments

Name	Type	Properties
append-op-list	list of OpenMP operation list item type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

declare_variant

Semantics

The **append_args** clause specifies additional arguments to pass in the call when a specified function variant is selected for replacement in the context of a function-dispatch structured block. The arguments are formed according to each specified list item in *append-op-list*, in the order those list items appear. The arguments are passed to the function variant after any named arguments of the base function in the same order in which they are formed. If the base function is variadic, the formed arguments are passed before any variadic arguments.

The supported OpenMP operations in *append-op-list* are:

interop

The **interop** operation accepts as its *operator-parameter-specification* any *modifier-specification-list* that is accepted by the **init** clause on the **interop** construct.

For each **interop** operation specified, an argument is formed and appended as follows. If the semantic requirement set contains one or more *interop* elements, the first of those elements that was added to the set is removed and the associated interoperability object of that removed element is appended as an argument. Otherwise, the **interop** operation constructs an argument of **interop** OpenMP type using the semantic requirement set of the encountering task. The argument is constructed as if by an **interop** construct with an **init** clause that specifies the *modifier-specification-list* specified in the **interop** operation. If the semantic requirement set contains one or more elements (as added by the **dispatch** construct) that correspond to clauses for an **interop** construct of *interop-type*, the behavior is as if the corresponding clauses are specified on the **interop** construct and those elements are removed from the semantic requirement set.

Any appended arguments that were not obtained from the *interop* elements of the semantic requirement set are destroyed after the call to the selected function variant returns, as if an **interop** construct with a **destroy** clause was used with the same clauses that were used to initialize the argument.

Cross References

- **declare_variant** Directive, see Section 9.6.4
- **destroy** Clause, see Section 5.7
- OpenMP Operations, see Section 5.2.3
- Semantic Requirement Set, see Section 9.5
- **init** Clause, see Section 5.6
- **interop** Construct, see Section 16.1

9.6.4 declare_variant Directive

Name: declare_variant **Category:** declarative	**Association:** declaration **Properties:** pure

Arguments

declare_variant(*[base–name:]variant-name*)

Name	Type	Properties
base-name	identifier of function type	optional
variant-name	identifier of function type	*default*

Clauses

adjust_args, **append_args**, **match**

Additional information

The **declare_variant** directive may alternatively be specified with **declare variant** as the *directive-name*.

Semantics

The **declare_variant** directive specifies declare variant semantics for a single replacement candidate; *variant-name* identifies the function variant while *base-name* identifies the base function.

C

Any expressions in the **match** clause are interpreted as if they appeared in the scope of arguments of the base function.

C

C++

variant-name and any expressions in the **match** clause are interpreted as if they appeared at the scope of the trailing return type of the base function.

The function variant is determined by base language standard name lookup rules ([basic.lookup]) of *variant-name* using the argument types at the call site after implementation defined changes have been made according to the OpenMP context.

C++

Fortran

The procedure to which *base-name* refers is resolved at the location of the directive according to the establishment rules for procedure names in the base language.

If a **declare_variant** directive appears in the specification part of a subprogram or an interface body, its bound procedure is this subprogram or the procedure defined by the interface body, respectively. Otherwise there is no bound procedure.

Fortran

Restrictions

The restrictions to the **declare_variant** directive are as follows:

C / C++

- If *base-name* is specified, it must match the name used in the associated declaration, if any declaration is associated.

C / C++

C++

- If an expression in the context selector that appears in a **match** clause references the **this** pointer, the base function must be a non-static member function.

C++

Fortran

- If the **declare_variant** directive does not have a bound procedure or the base function is not the bound procedure, *base-name* must be specified.
- *base-name* must not be a generic name, an entry name, the name of a procedure pointer, a dummy procedure or a statement function.
- The procedure *base-name* must have an accessible explicit interface at the location of the directive.

Fortran

Cross References

- **adjust_args** Clause, see Section 9.6.2
- **append_args** Clause, see Section 9.6.3
- Declare Variant Directives, see Section 9.6
- **match** Clause, see Section 9.6.1

C / C++

9.6.5 begin declare_variant Directive

Name: begin declare_variant	**Association:** delimited
Category: declarative	**Properties:** *default*

Clauses

match

Additional information

The **begin declare_variant** directive may alternatively be specified with **begin declare variant** as the *directive-name*.

Semantics

The **begin declare_variant** directive associates the context selector in the **match** clause with each function definition in the delimited code region formed by the directive and its paired end directive. The delimited code region is a declaration sequence. For the purpose of call resolution, each function definition that appears in the delimited code region is a function variant for an assumed base function, with the same name and a compatible prototype, that is declared elsewhere without an associated declare variant directive.

If a declare variant directive appears between a **begin declare_variant** directive and its paired end directive, the effective context selectors of the outer directive are appended to the context selector of the inner directive to form the effective context selector of the inner directive. If a *trait-set-selector* is present on both directives, the *trait-selector* list of the outer directive is appended to the *trait-selector* list of the inner directive after equivalent *trait-selectors* have been removed from the outer list. Restrictions that apply to explicitly specified context selectors also apply to effective context selectors constructed through this process.

The symbol name of a function definition that appears between a **begin declare_variant** directive and its paired end directive is determined through the base language rules after the name of the function has been augmented with a string that is determined according to the effective context selector of the **begin declare_variant** directive. The symbol names of two definitions of a function are considered to be equal if and only if their effective context selectors are equivalent.

If the context selector of a **begin declare_variant** directive contains traits in the *device* or *implementation* set that are known never to be compatible with an OpenMP context during the current compilation, the preprocessed code that follows the **begin declare_variant** directive up to its paired end directive is elided.

Any expressions in the **match** clause are interpreted at the location of the directive.

Restrictions

The restrictions to **begin declare_variant** directive are as follows:

- **match** clause must not contain a **simd** trait selector.
- Two **begin declare_variant** directives and their paired end directives must either encompass disjoint source ranges or be perfectly nested.

C++

- A **match** clause must not contain a dynamic context selector that references the **this** pointer.

C++

Cross References

- Declare Variant Directives, see Section 9.6
- **match** Clause, see Section 9.6.1

C / C++

9.7 dispatch Construct

Name: **dispatch** **Category:** executable	**Association:** block : function-dispatch **Properties:** context-matching

Clauses

depend, **device**, **has_device_addr**, **interop**, **is_device_ptr**, **nocontext**, **novariants**, **nowait**

Binding

The binding task set for a **dispatch** region is the generating task. The **dispatch** region binds to the region of the generating task.

Semantics

The **dispatch** construct controls whether variant substitution occurs for *target-call* in the associated function-dispatch structured block. The **dispatch** construct may also modify the semantic requirement set of elements that affect the arguments of the function variant if variant substitution occurs (see Section 9.6.2 and Section 9.6.3).

Elements added to the semantic requirement set by the **dispatch** construct can be removed by the effect of declare variant directives (see Section 9.5) before the **dispatch** region is executed. If one or more **depend** clauses are present on the **dispatch** construct, they are added as *depend* elements of the semantic requirement set. If a **nowait** clause is present on the **dispatch** construct the *nowait* element is added to the semantic requirement set. For each list item specified in an **is_device_ptr** clause, an *is_device_ptr* element for that list item is added to the semantic requirement set. For each list item specified in a **has_device_addr** clause, a *has_device_addr* element for that list item is added to the semantic requirement set. For each list item specified in an **interop** clause, an *interop* element for that list item is added to the semantic requirement set in the same order that they were specified on the directive.

If the **dispatch** directive adds one or more *depend* element to the semantic requirement set, and those element are not removed by the effect of a declare variant directive, the behavior is as if those elements were applied as **depend** clauses to a **taskwait** construct that is executed before the **dispatch** region is executed.

The addition of the *nowait* and *interop* elements to the semantic requirement set by the **dispatch** directive has no effect on the **dispatch** construct apart from the effect it may have on the arguments that are passed when calling a function variant.

If the **device** clause is present, the value of the *default-device-var* ICV is set to the value of the expression in the clause on entry to the **dispatch** region and is restored to its previous value at the end of the region.

If the **interop** clause is present and has only one *interop-var*, and the **device** clause is not specified, the behavior is as if the **device** clause is present with a *device-description* equivalent to the *device_num* property of the *interop-var*.

Restrictions

Restrictions to the **dispatch** construct are as follows:

- If the **interop** clause is present and has more than one *interop-var* then the **device** clause must also be present.

Cross References

- **depend** Clause, see Section 17.9.5

- **device** Clause, see Section 15.2
- OpenMP Function Dispatch Structured Blocks, see Section 6.3.2
- Semantic Requirement Set, see Section 9.5
- **has_device_addr** Clause, see Section 7.5.9
- **interop** Clause, see Section 9.7.1
- **is_device_ptr** Clause, see Section 7.5.7
- **nocontext** Clause, see Section 9.7.3
- **novariants** Clause, see Section 9.7.2
- **nowait** Clause, see Section 17.6
- **taskwait** Construct, see Section 17.5

9.7.1 interop Clause

Name: interop	**Properties:** unique

Arguments

Name	Type	Properties
interop-var-list	list of variable of interop OpenMP type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

dispatch

Semantics

The **interop** clause specifies interoperability objects to be added to the semantic requirement set of the encountering task. They are added to the semantic requirement set in the same order in which they are specified in the **interop** clause.

Restrictions

Restrictions to the **interop** clause are as follows:

- If the **interop** clause is specified on a **dispatch** construct, the matching **declare_variant** directive for the *target-call* must have an **append_args** clause with a number of list items that equals or exceeds the number of list items in the **interop** clause.

Cross References

- **dispatch** Construct, see Section 9.7

9.7.2 novariants Clause

Name: novariants	**Properties:** unique

Arguments

Name	Type	Properties
do-not-use-variant	expression of OpenMP logical type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

dispatch

Semantics

If *do-not-use-variant* evaluates to *true*, no function variant is selected for the *target-call* of the **dispatch** region associated with the **novariants** clause even if one would be selected normally. The use of a variable in *do-not-use-variant* causes an implicit reference to the variable in all enclosing constructs. *do-not-use-variant* is evaluated in the enclosing context.

Cross References

- **dispatch** Construct, see Section 9.7

9.7.3 nocontext Clause

Name: nocontext	**Properties:** unique

Arguments

Name	Type	Properties
do-not-update-context	expression of OpenMP logical type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

dispatch

Semantics

If *do-not-update-context* evaluates to *true*, the construct on which the **nocontext** clause appears is not added to the construct trait set of the OpenMP context. The use of a variable in *do-not-update-context* causes an implicit reference to the variable in all enclosing constructs. *do-not-update-context* is evaluated in the enclosing context.

Cross References

- **dispatch** Construct, see Section 9.7

9.8 declare_simd Directive

Name: declare_simd Category: declarative	Association: declaration Properties: pure, variant-generating

Arguments

declare_simd[**(***proc-name***)**]

Name	Type	Properties
proc-name	identifier of function type	optional

Clause groups

branch

Clauses

aligned, **linear**, **simdlen**, **uniform**

Additional information

The **declare_simd** directive may alternatively be specified with **declare simd** as the *directive-name*.

Semantics

The association of one or more **declare_simd** directives with a procedure declaration or definition enables the creation of corresponding SIMD versions of the associated procedure that can be used to process multiple arguments from a single invocation in a SIMD loop concurrently.

If a SIMD version is created and the **simdlen** clause is not specified, the number of concurrent arguments for the function is implementation defined.

For purposes of the **linear** clause, any integer-typed parameter that is specified in a **uniform** clause on the directive is considered to be constant and so may be used in a *step-complex-modifier* as *linear-step*.

C / C++

The expressions that appear in the clauses of each directive are evaluated in the scope of the arguments of the procedure declaration or definition.

C / C++

C++

The special **this** pointer can be used as if it was one of the arguments to the procedure in any of the **linear**, **aligned**, or **uniform** clauses.

C++

Restrictions

Restrictions to the **declare_simd** directive are as follows:

- The procedure body must be a structured block.
- The execution of the procedure, when called from a SIMD loop, must not result in the execution of any constructs except for **atomic** constructs and **ordered** constructs on which the **simd** clause is specified.
- The execution of the procedure must not have any side effects that would alter its execution for concurrent iterations of a SIMD chunk.

C / C++

- If a **declare_simd** directive is specified for a declaration of a procedure then the definition of the procedure must have a **declare_simd** directive with identical clauses with identical arguments and modifiers.
- The procedure must not contain calls to the **longjmp** or **setjmp** functions.

C / C++

C++

- The procedure must not contain **throw** statements.

C++

Fortran

- *proc-name* must not be a generic name, procedure pointer, or entry name.
- If *proc-name* is omitted, the **declare_simd** directive must appear in the specification part of a subroutine subprogram or a function subprogram for which creation of the SIMD versions is enabled.
- Any **declare_simd** directive must appear in the specification part of a subroutine subprogram, function subprogram, or interface body to which it applies.
- If a procedure is declared via a procedure declaration statement, the procedure *proc-name* should appear in the same specification.

- If a **declare_simd** directive is specified for a procedure then the definition of the procedure must contain a **declare_simd** directive with identical clauses with identical arguments and modifiers.
- Procedures pointers may not be used to access versions created by the **declare_simd** directive.

Fortran

Cross References

- **aligned** Clause, see Section 7.12
- **linear** Clause, see Section 7.5.6
- **simdlen** Clause, see Section 12.4.3
- **uniform** Clause, see Section 7.11

9.8.1 *branch* Clauses

Clause groups

Properties: exclusive, unique	Members: Clauses **inbranch**, **notinbranch**

Directives

declare_simd

Semantics

The *branch* clause group defines a set of clauses that indicate if a procedure can be assumed to be or not to be encountered in a branch. If neither clause is specified, then the procedure may or may not be called from inside a conditional statement of the calling context.

Cross References

- **declare_simd** Directive, see Section 9.8

9.8.1.1 inbranch Clause

Name: **inbranch**	Properties: unique

Arguments

Name	Type	Properties
inbranch	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`declare_simd`

Semantics

If *inbranch* evaluates to *true*, the **`inbranch`** clause specifies that the procedure will always be called from inside a conditional statement of the calling context. If *inbranch* evaluates to *false*, the procedure may be called other than from inside a conditional statement. If *inbranch* is not specified, the effect is as if *inbranch* evaluates to *true*.

Cross References

- **`declare_simd`** Directive, see Section 9.8

9.8.1.2 `notinbranch` Clause

Name: **`notinbranch`**	**Properties:** unique

Arguments

Name	Type	Properties
notinbranch	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`declare_simd`

Semantics

If *notinbranch* evaluates to *true*, the **`notinbranch`** clause specifies that the procedure will never be called from inside a conditional statement of the calling context. If *notinbranch* evaluates to *false*, the procedure may be called from inside a conditional statement. If *notinbranch* is not specified, the effect is as if *notinbranch* evaluates to *true*.

Cross References

- **`declare_simd`** Directive, see Section 9.8

9.9 Declare Target Directives

Declare target directives apply to procedures and/or variables to ensure that they can be executed or accessed on a device. Variables are either replicated as device-local variables for each device through a **local** clause, are mapped for all device executions through an **enter** clause, or are mapped for specific device executions through a **link** clause. An implementation may generate different versions of a procedure to be used for **target** regions that execute on different devices. Whether it generates different versions, and whether it calls a different version in a **target** region from the version that it calls outside a **target** region, are implementation defined.

To facilitate device usage, OpenMP defines rules that implicitly specify declare target directives for procedures and variables. The remainder of this section defines those rules as well as restrictions that apply to all declare target directives.

C++

If a variable with static storage duration has the **constexpr** specifier and is not a groupprivate variable then the variable is treated as if it had appeared as a list item in an **enter** clause on a declare target directive.

C++

If a variable with static storage duration that is not a device-local variable (including that it is not a groupprivate variable) is declared in a device procedure then the variable is treated as if it had appeared as a list item in an **enter** clause on a declare target directive.

If a procedure is referenced outside of any reverse-offload region in a procedure that appears as a list item in an **enter** clause on a non-host declare target directive then the name of the referenced procedure is treated as if it had appeared in an **enter** clause on a declare target directive.

C / C++

If a variable with static storage duration or a function (except *lambda* for C++) is referenced in the initializer expression list of a variable with static storage duration that appears as a list item in an **enter** or **local** clause on a declare target directive then the name of the referenced variable or procedure is treated as if it had appeared in an **enter** clause on a declare target directive.

C / C++

Fortran

If a **declare_target** directive has a **device_type** clause then any enclosed internal procedure cannot contain any **declare_target** directives. The enclosing **device_type** clause implicitly applies to internal procedures.

Fortran

A reference to a device-local variable that has static storage duration inside a device procedure is replaced with a reference to the copy of the variable for the device. Otherwise, a reference to a variable that has static storage duration in a device procedure is replaced with a reference to a corresponding variable in the device data environment. If the corresponding variable does not exist or the variable does not appear in an **enter** or **link** clause on a declare target directive, the behavior is unspecified.

Execution Model Events

The *target-global-data-op* event occurs when an original list item is associated with a corresponding list item on a device as a result of a declare target directive; the event occurs before the first access to the corresponding list item.

Tool Callbacks

A thread dispatches a registered **target_data_op_emi** callback with **ompt_scope_beginend** as its *endpoint* argument for each occurrence of a *target-global-data-op* event in that thread.

Restrictions

Restrictions to any declare target directive are as follows:

- The same list item must not explicitly appear in both an **enter** clause on one declare target directive and a **link** or **local** clause on another declare target directive.
- The same list item must not explicitly appear in both a **link** clause on one declare target directive and a **local** clause on another declare target directive.
- If a variable appears in a **enter** clause on a declare target directive, its initializer must not refer to a variable that appears in a **link** clause on a declare target directive.

Cross References

- **begin declare_target** Directive, see Section 9.9.2
- **declare_target** Directive, see Section 9.9.1
- **enter** Clause, see Section 7.9.7
- **link** Clause, see Section 7.9.8
- OMPT **scope_endpoint** Type, see Section 33.27
- **target** Construct, see Section 15.8
- **target_data_op_emi** Callback, see Section 35.7

9.9.1 declare_target Directive

Name: **declare_target** **Category:** declarative	**Association:** explicit **Properties:** declare-target, device, pure, variant-generating

Arguments

declare_target (*extended-list*)

Name	Type	Properties
extended-list	list of extended list item type	optional

Clauses

device_type, **enter**, **indirect**, **link**, **local**

Additional information

The **declare_target** directive may alternatively be specified with **declare target** as the *directive-name*.

Semantics

The **declare_target** directive is a declare target directive. If the *extended-list* argument is specified, the effect is as if any list items from *extended-list* that are not groupprivate variables appear in the *list* argument of an implicit **enter** clause and any list items that are groupprivate variables appear in the *list* argument of an implicit **local** clause.

If neither the *extended-list* argument nor a data-environment attribute clause is specified then the directive is a declaration-associated directive. The effect is as if the name of the associated procedure appears as a list item in an **enter** clause of a declare target directive that otherwise specifies the same set of clauses.

C / C++

If the **declare_target** directive is specified as an attribute specifier with the **decl** attribute and a **decl** attribute is not used on the declaration to specify groupprivate variables, the effect is as if an **enter** clause is specified if a **link** or **local** clause is not specified.

If the **declare_target** directive is specified as an attribute specifier with the **decl** attribute and a **decl** attribute is used on the declaration to specify groupprivate variables, the effect is as if a **local** clause is specified.

C / C++

Restrictions

Restrictions to the **declare_target** directive are as follows:

- If the *extended-list* argument is specified, no clauses may be specified.
- If the directive is not a declaration-associated directive and an *extended-list* argument is not specified, a data-environment attribute clause must be present.
- A variable for which **nohost** is specified must not appear in a **link** clause.
- A groupprivate variable must not appear in any **enter** clauses or **link** clauses.

C / C++

- If the directive is not a declaration-associated directive, it must appear at the same scope as the declaration of every list item in its *extended-list* or in its data-environment attribute clauses.

C / C++

Fortran

- If a list item is a procedure name, it must not be a generic name, procedure pointer, entry name, or statement function name.
- If the directive is a declaration-associated directive, the directive must appear in the specification part of a subroutine subprogram, function subprogram or interface body.
- If a list item is a procedure name that is not declared via a procedure declaration statement, the directive must be in the specification part of the subprogram or interface body of that procedure.
- If a list item in *extended-list* is a variable, the directive must appear in the specification part in which the variable is declared.
- If a **declare_target** directive is specified for a procedure that has an explicit interface then the definition of the procedure must contain a **declare_target** directive with identical clauses with identical arguments and modifiers.
- If an external procedure is a type-bound procedure of a derived type and the directive is specified in the definition of the external procedure, it must appear in the interface block that is accessible to the derived-type definition.
- If any procedure is declared via a procedure declaration statement that is not in the type-bound procedure part of a derived-type definition, any **declare_target** directive with the procedure name must appear in the same specification part.
- If a **declare_target** directive that specifies a common block name appears in one program unit, then such a directive must also appear in every other program unit that contains a **COMMON** statement that specifies the same name, after the last such **COMMON** statement in the program unit.
- If a list item is declared with the **BIND** attribute, the corresponding C entities must also be specified in a **declare_target** directive in the C program.
- A variable can only appear in a **declare_target** directive in the scope in which it is declared. It must not be an element of a common block or appear in an **EQUIVALENCE** statement.

Fortran

Cross References

- **device_type** Clause, see Section 15.1
- **enter** Clause, see Section 7.9.7
- Declare Target Directives, see Section 9.9
- **indirect** Clause, see Section 9.9.3
- **link** Clause, see Section 7.9.8

- **local** Clause, see Section 7.14

C / C++

9.9.2 begin declare_target Directive

Name: **begin declare_target** **Category:** declarative	**Association:** delimited **Properties:** declare-target, device, variant-generating

Clauses

device_type, **indirect**

Additional information

The **begin declare_target** directive may alternatively be specified with **begin declare target** as the *directive-name*.

Semantics

The **begin declare_target** directive is a declare target directive. The directive and its paired end directive form a delimited code region that defines an implicit *extended-list* and implicit *local-list* that is converted to an implicit **enter** clause with the *extended-list* as its argument and an implicit **local** clause with the *local-list* as its argument, respectively. The delimited code region is a declaration sequence.

The implicit *extended-list* consists of the variable and procedure names of any variable or procedure declarations at file scope that appear in the delimited code region, excluding declarations of groupprivate variables. If any groupprivate variables are declared in the delimited code region, the effect is as if the variables appear in the implicit *local-list*.

C++

Additionally, the implicit *extended-list* and *local-list* consist of the variable and procedure names of any variable or procedure declarations at namespace or class scope that appear in the delimited code region, including the **operator()** member function of the resulting closure type of any lambda expression that is defined in the delimited code region.

C++

The delimited code region may contain declare target directives. If a **device_type** clause is present on the contained declare target directive, then its argument determines which versions are made available. If a list item appears both in an implicit and explicit list, the explicit list determines which versions are made available.

Restrictions

Restrictions to the **begin declare_target** directive are as follows:

C++

- The function names of overloaded functions or template functions may only be specified within an implicit *extended-list*.
- If a *lambda declaration and definition* appears between a **begin declare_target** directive and the paired end directive, all variables that are captured by the lambda expression must also appear in an **enter** clause.
- A module **export** or **import** statement may not appear between a **begin declare_target** directive and the paired end directive.

C++

Cross References

- **device_type** Clause, see Section 15.1
- **enter** Clause, see Section 7.9.7
- Declare Target Directives, see Section 9.9
- **indirect** Clause, see Section 9.9.3

C / C++

9.9.3 indirect Clause

Name: **indirect**	**Properties:** unique

Arguments

Name	Type	Properties
invoked-by-fptr	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

begin declare_target, **declare_target**

Semantics

If *invoked-by-fptr* evaluates to *true*, any procedures that appear in an **enter** clause on the directive on which the **indirect** clause is specified may be called with an indirect device invocation. If the *invoked-by-fptr* does not evaluate to *true*, any procedures that appear in an **enter** clause on the directive may not be called with an indirect device invocation. Unless otherwise specified by an **indirect** clause, procedures may not be called with an indirect device invocation. If the **indirect** clause is specified and *invoked-by-fptr* is not specified, the effect of the clause is as if *invoked-by-fptr* evaluates to *true*.

C / C++

If a procedure appears in the implicit **enter** clause of a **begin declare_target** directive and in the **enter** clause of a declare target directive that is contained in the delimited code region of the **begin declare_target** directive, and if an **indirect** clause appears on both directives, then the **indirect** clause on the **begin declare_target** directive has no effect or that procedure.

C / C++

Restrictions

Restrictions to the **indirect** clause are as follows:

- If *invoked-by-fptr* evaluates to *true*, a **device_type** clause must not appear on the same directive unless it specifies **any** for its *device-type-description*.

Cross References

- **begin declare_target** Directive, see Section 9.9.2
- **declare_target** Directive, see Section 9.9.1

10 Informational and Utility Directives

An informational directive conveys information about code properties to the compiler while a utility directive facilitates interactions with the compiler or supports code readability. A utility directive is informational unless the **at** clause implies it is an executable directive.

10.1 `error` Directive

Name: **error** Category: utility	Association: unassociated Properties: pure

Clauses

at, **message**, **severity**

Semantics

The **error** directive instructs the compiler or runtime to perform an error action. The error action displays an implementation defined message. The **severity** clause determines whether the error action is abortive following the display of the message. If *sev-level* is **fatal** and the *action-time* of the **at** clause is **compilation**, the message is displayed and compilation of the current compilation unit is aborted. If *sev-level* is **fatal** and *action-time* is **execution**, the message is displayed and program execution is aborted.

Execution Model Events

The *runtime-error* event occurs when a thread encounters an **error** directive for which the **at** clause specifies **execution**.

Tool Callbacks

A thread dispatches a registered **error** callback for each occurrence of a *runtime-error* event in the context of the encountering task.

Restrictions

Restrictions to the **error** directive are as follows:

- The directive is pure only if *action-time* is **compilation**.

Cross References

- **at** Clause, see Section 10.2
- **error** Callback, see Section 34.2

- **message** Clause, see Section 10.3
- **severity** Clause, see Section 10.4

10.2 at Clause

Name: at	Properties: unique

Arguments

Name	Type	Properties
action-time	Keyword: **compilation**, **execution**	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

error

Semantics

The **at** clause determines when the implementation performs an action that is associated with a utility directive. If *action-time* is **compilation**, the action is performed during compilation if the directive appears in a declarative context or in an executable context that is reachable at runtime. If *action-time* is **compilation** and the directive appears in an executable context that is not reachable at runtime, the action may or may not be performed. If *action-time* is **execution**, the action is performed during program execution when a thread encounters the directive and the directive is considered to be an executable directive. If the **at** clause is not specified, the effect is as if *action-time* is **compilation**.

Cross References

- **error** Directive, see Section 10.1

10.3 message Clause

Name: message	Properties: unique

Arguments

Name	Type	Properties
msg-string	expression of string type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`error`, **`parallel`**

Semantics

The **`message`** clause specifies that *msg-string* is included in the implementation defined message that is associated with the directive on which the clause appears.

Restrictions

- If the *action-time* is **`compilation`**, *msg-string* must be a constant expression.

Cross References

- **`error`** Directive, see Section 10.1
- **`parallel`** Construct, see Section 12.1

10.4 `severity` Clause

Name: **`severity`**	**Properties:** unique

Arguments

Name	Type	Properties
sev-level	Keyword: **`fatal`**, **`warning`**	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`error`, **`parallel`**

Semantics

The **`severity`** clause determines the action that the implementation performs if an error is encountered with respect to the directive on which the clause appears. If *sev-level* is **`warning`**, the implementation takes no action besides displaying the message that is associated with the directive. If *sev-level* is **`fatal`**, the implementation performs the abortive action associated with the directive on which the clause appears. If no **`severity`** clause is specified then the effect is as if *sev-level* is **`fatal`**.

Cross References

- **error** Directive, see Section 10.1
- **parallel** Construct, see Section 12.1

10.5 requires Directive

Name: **requires** Category: informational	Association: unassociated Properties: *default*

Clause groups

requirement

Semantics

The **requires** directive specifies features that an implementation must support for correct execution and requirements for the execution of all code in the current compilation unit. The behavior that a *requirement* clause specifies may override the normal behavior specified elsewhere in this document. Whether an implementation supports the feature that a given *requirement* clause specifies is implementation defined.

The clauses of a **requires** directive are added to the *requires* trait in the OpenMP context for all program points that follow the directive.

Restrictions

Restrictions to the **requires** directive are as follows:

- A **requires** directive must appear lexically after the specification of a context selector in which any clause of that **requires** directive is used, nor may the directive appear lexically after any code that depends on such a context selector.

C

- The **requires** directive must only appear at file scope.

C

C++

- The **requires** directive must only appear at file or namespace scope.

C++

C / C++

- Any **requires** directive that specifies a device global requirement clause must appear lexically before any device constructs or device procedures.

C / C++

Fortran

- The **requires** directive must appear in the specification part of a program unit, either after all **USE** statements, **IMPORT** statements, and **IMPLICIT** statements or by referencing a module. Additionally, it may appear in the specification part of an internal or module subprogram that appears by referencing a module if each clause already appeared with the same arguments in the specification part of the program unit.

Fortran

10.5.1 *requirement* Clauses

Clause groups

Properties: required, unique	**Members:**
	Clauses **atomic_default_mem_order**, **device_safesync**, **dynamic_allocators**, **reverse_offload**, **self_maps**, **unified_address**, **unified_shared_memory**

Directives

requires

Semantics

The *requirement* clause group defines a clause set that indicates the requirements that a program requires the implementation to support. If an implementation supports a given *requirement* clause then the use of that clause on a **requires** directive will cause the implementation to ensure the enforcement of a guarantee represented by the specific member of the clause group. If the implementation does not support the requirement then it must perform compile-time error termination.

Restrictions

- All compilation units of a program that contain declare target directives, device constructs or device procedures must specify the same set of requirements that are defined by clauses with the device global requirement property in the *requirement* clause group.

Cross References

- **requires** Directive, see Section 10.5

10.5.1.1 atomic_default_mem_order Clause

Name: atomic_default_mem_order	**Properties:** unique

Arguments

Name	Type	Properties
memory-order	Keyword: **acq_rel**, **acquire**, **relaxed**, **release**, **seq_cst**	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

requires

Semantics

The **atomic_default_mem_order** clause specifies the default memory ordering behavior for **atomic** constructs that an implementation must provide. The effect is as if its argument appears as a clause on any **atomic** construct that does not specify a *memory-order* clause.

Restrictions

Restrictions to the **atomic_default_mem_order** clause are as follows:

- All **requires** directives in the same compilation unit that specify the **atomic_default_mem_order** requirement must specify the same argument.
- Any directive that specifies the **atomic_default_mem_order** clause must not appear lexically after any **atomic** construct on which a *memory-order* clause is not specified.

Cross References

- **atomic** Construct, see Section 17.8.5
- *memory-order* Clauses, see Section 17.8.1
- **requires** Directive, see Section 10.5

10.5.1.2 dynamic_allocators Clause

Name: dynamic_allocators	**Properties:** unique

Arguments

Name	Type	Properties
required	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

requires

Semantics

If *required* evaluates to *true*, the **dynamic_allocators** clause removes certain restrictions on the use of memory allocators in **target** regions. Specifically, allocators (including the default allocator that is specified by the *def-allocator-var* ICV) may be used in a **target** region or in an **allocate** clause on a **target** construct without specifying the **uses_allocators** clause on the **target** construct. Additionally, the implementation must support calls to the **omp_init_allocator** and **omp_destroy_allocator** API routines in **target** regions. If *required* is not specified, the effect is as if *required* evaluates to *true*.

Cross References

- **allocate** Clause, see Section 8.6
- *def-allocator-var* ICV, see Table 3.1
- **omp_destroy_allocator** Routine, see Section 27.7
- **omp_init_allocator** Routine, see Section 27.6
- **requires** Directive, see Section 10.5
- **target** Construct, see Section 15.8
- **uses_allocators** Clause, see Section 8.8

10.5.1.3 reverse_offload Clause

Name: **reverse_offload**	**Properties:** unique, device global requirement

Arguments

Name	Type	Properties
required	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

requires

Semantics

If *required* evaluates to *true*, the **reverse_offload** clause requires an implementation to guarantee that if a **target** construct specifies a **device** clause in which the **ancestor** *device-modifier* appears, the **target** region can execute on the parent device of an enclosing **target** region. If *required* is not specified, the effect is as if *required* evaluates to *true*.

Cross References

- **device** Clause, see Section 15.2
- **requires** Directive, see Section 10.5
- **target** Construct, see Section 15.8

10.5.1.4 unified_address Clause

Name: **unified_address**	**Properties:** unique, device global requirement

Arguments

Name	Type	Properties
required	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

requires

Semantics

If *required* evaluates to *true*, the **unified_address** clause requires an implementation to guarantee that all devices accessible through OpenMP API routines and directives use a unified address space. In this address space, a pointer will always refer to the same location in memory from all devices accessible through OpenMP. Any OpenMP mechanism that returns a device pointer is guaranteed to return a device address that supports pointer arithmetic, and the **is_device_ptr** clause is not necessary to obtain device addresses from device pointers for use inside **target** regions. Host pointers may be passed as device pointer arguments to device memory routines and device pointers may be passed as host pointer arguments to device memory routines. Non-host devices may still have discrete memories and dereferencing a device pointer on the host device or a host pointer on a non-host device remains unspecified behavior. Memory local to a specific execution context may be exempt from the **unified_address** requirement, following the restrictions of locality to a given execution context, thread or contention group. If *required* is not specified, the effect is as if *required* evaluates to *true*.

Cross References

- **is_device_ptr** Clause, see Section 7.5.7
- **requires** Directive, see Section 10.5
- **target** Construct, see Section 15.8

10.5.1.5 unified_shared_memory Clause

Name: **unified_shared_memory**	**Properties:** unique, device global requirement

Arguments

Name	Type	Properties
required	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

requires

Semantics

If *required* evaluates to *true*, the **unified_shared_memory** clause requires the implementation to guarantee that all devices share memory that is generally accessible to all threads.

The **unified_shared_memory** clause implies the **unified_address** requirement, inheriting all of its behaviors.

The implementation must guarantee that storage locations in memory are accessible to threads on all accessible devices, except for memory that is local to a specific execution context and exempt from the **unified_address** requirement (see Section 10.5.1.4). Every device address that refers to storage allocated through OpenMP API routines is a valid host pointer that may be dereferenced and may be used as a host address. Values stored into memory by one device may not be visible to another device until synchronization establishes a happens-before order between the memory accesses.

The use of declare target directives in an OpenMP program is optional for referencing variables with static storage duration in device procedures.

Any data object that results from the declaration of a variable that has static storage duration is treated as if it is mapped with a persistent self map at the beginning of the program to the device data environments of all target devices if:

- The variable is not a device-local variable;
- The variable is not listed in an **enter** clause on a declare target directive; and
- The variable is referenced in a device procedure.

If *required* is not specified, the effect is as if *required* evaluates to *true*.

Cross References

- **enter** Clause, see Section 7.9.7
- **requires** Directive, see Section 10.5
- **unified_address** Clause, see Section 10.5.1.4

10.5.1.6 self_maps Clause

Name: **self_maps**	**Properties:** unique, device global requirement

Arguments

Name	Type	Properties
required	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

requires

Semantics

If *required* evaluates to *true*, the **self_maps** clause implies the **unified_shared_memory** clause, inheriting all of its behaviors. Additionally, map-entering clauses in the compilation unit behave as if all resulting mapping operations are self maps, and all corresponding list items created by the **enter** clauses specified by declare target directives in the compilation unit share storage with the original list items. If *required* is not specified, the effect is as if *required* evaluates to *true*.

Cross References

- **enter** Clause, see Section 7.9.7
- **requires** Directive, see Section 10.5
- **unified_shared_memory** Clause, see Section 10.5.1.5

10.5.1.7 device_safesync Clause

Name: device_safesync	Properties: unique, device global requirement

Arguments

Name	Type	Properties
required	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

requires

Semantics

If *required* evaluates to *true*, the **device_safesync** clause indicates that any two synchronizing divergent threads in a team that execute on a non-host device must be able to make progress, unless indicated otherwise by the use of a **safesync** clause. If *required* is not specified, the effect is as if *required* evaluates to *true*.

Cross References

- **requires** Directive, see Section 10.5
- **safesync** Clause, see Section 12.1.5

10.6 Assumption Directives

Different assumption directives facilitate definition of assumptions for a scope that is appropriate to each base language. The assumption scope of a particular format is defined in the section that defines that directive. If the invariants specified by the assumption directive do not hold at runtime, the behavior is unspecified.

10.6.1 *assumption* Clauses

Clause groups

Properties: required, unique	**Members:**
	Clauses **absent**, **contains**, **holds**, **no_openmp**, **no_openmp_constructs**, **no_openmp_routines**, **no_parallelism**

Directives

assume, **assumes**, **begin assumes**

Semantics

The *assumption* clause group defines a clause set that indicates the invariants that a program ensures the implementation can exploit.

The **absent** and **contains** clauses accept a *directive-name* list that may match a construct that is encountered within the assumption scope. An encountered construct matches the directive name if it or one of its constituent constructs has the same *directive-name* as one of the list items.

Restrictions

The restrictions to *assumption* clauses are as follows:

- A *directive-name* list item must not specify a directive that is a declarative directive, an informational directive, or a metadirective.

Cross References

- **assume** Directive, see Section 10.6.3
- **assumes** Directive, see Section 10.6.2
- **begin assumes** Directive, see Section 10.6.4

10.6.1.1 absent Clause

Name: absent	**Properties:** unique

Arguments

Name	Type	Properties
directive-name-list	list of directive-name list item type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`assume`, **`assumes`**, **`begin assumes`**

Semantics

The **`absent`** clause specifies that the program guarantees that no construct that matches a *directive-name* list item is encountered in the assumption scope.

Cross References

- **`assume`** Directive, see Section 10.6.3
- **`assumes`** Directive, see Section 10.6.2
- **`begin assumes`** Directive, see Section 10.6.4

10.6.1.2 `contains` Clause

Name: `contains`	**Properties:** unique

Arguments

Name	Type	Properties
directive-name-list	list of directive-name list item type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`assume`, **`assumes`**, **`begin assumes`**

Semantics

The **`contains`** clause specifies that constructs that match the *directive-name* list items are likely to be encountered in the assumption scope.

Cross References

- **`assume`** Directive, see Section 10.6.3
- **`assumes`** Directive, see Section 10.6.2
- **`begin assumes`** Directive, see Section 10.6.4

10.6.1.3 `holds` Clause

Name: `holds`	**Properties:** unique

Arguments

Name	Type	Properties
hold-expr	expression of OpenMP logical type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

assume, **assumes**, **begin assumes**

Semantics

When the **holds** clause appears on an assumption directive, the program guarantees that the listed expression evaluates to *true* in the assumption scope. The effect of the clause does not include any evaluation of the expression that affects the behavior of the program.

Cross References

- **assume** Directive, see Section 10.6.3
- **assumes** Directive, see Section 10.6.2
- **begin assumes** Directive, see Section 10.6.4

10.6.1.4 no_openmp Clause

Name: no_openmp	Properties: unique

Arguments

Name	Type	Properties
can_assume	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

assume, **assumes**, **begin assumes**

Semantics

If *can_assume* evaluates to *true*, the **no_openmp** clause implies the **no_openmp_constructs** clause and the **no_openmp_routines** clause. If *can_assume* is not specified, the effect is as if *can_assume* evaluates to *true*.

C++

The **no_openmp** clause also guarantees that no thread will throw an exception in the assumption scope if it is contained in a region that arises from an exception-aborting directive.

C++

Cross References

- **assume** Directive, see Section 10.6.3
- **assumes** Directive, see Section 10.6.2
- **begin assumes** Directive, see Section 10.6.4

10.6.1.5 no_openmp_constructs Clause

Name: no_openmp_constructs	Properties: unique

Arguments

Name	Type	Properties
can_assume	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

assume, **assumes**, **begin assumes**

Semantics

If *can_assume* evaluates to *true*, the **no_openmp_constructs** clause guarantees that no constructs are encountered in the assumption scope. If *can_assume* is not specified, the effect is as if *can_assume* evaluates to *true*.

Cross References

- **assume** Directive, see Section 10.6.3
- **assumes** Directive, see Section 10.6.2
- **begin assumes** Directive, see Section 10.6.4

10.6.1.6 no_openmp_routines Clause

Name: no_openmp_routines	Properties: unique

Arguments

Name	Type	Properties
can_assume	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

assume, **assumes**, **begin assumes**

Semantics

If *can_assume* evaluates to *true*, the **no_openmp_routines** clause guarantees that no OpenMP API routines are executed in the assumption scope. If *can_assume* is not specified, the effect is as if *can_assume* evaluates to *true*.

Cross References

- **assume** Directive, see Section 10.6.3
- **assumes** Directive, see Section 10.6.2
- **begin assumes** Directive, see Section 10.6.4

10.6.1.7 no_parallelism Clause

Name: no_parallelism	Properties: unique

Arguments

Name	Type	Properties
can_assume	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

assume, **assumes**, **begin assumes**

Semantics

If *can_assume* evaluates to *true*, the **no_parallelism** clause guarantees that no parallelism-generating constructs will be encountered in the assumption scope. If *can_assume* is not specified, the effect is as if *can_assume* evaluates to *true*.

Cross References

- **assume** Directive, see Section 10.6.3
- **assumes** Directive, see Section 10.6.2
- **begin assumes** Directive, see Section 10.6.4

10.6.2 assumes Directive

Name: assumes	Association: unassociated
Category: informational	Properties: pure

Clause groups

assumption

Semantics

The assumption scope of the **assumes** directive is the code executed and reached from the current compilation unit.

Fortran

Referencing a module that has an **assumes** directive in its specification part does not have the effect as if the **assumes** directive appeared in the specification part of the referencing scope.

Fortran

Restrictions

The restrictions to the **assumes** directive are as follows:

C

- The **assumes** directive must only appear at file scope.

C

C++

- The **assumes** directive must only appear at file or namespace scope.

C++

Fortran

- The **assumes** directive must only appear in the specification part of a module or subprogram, after all **USE** statements, **IMPORT** statements, and **IMPLICIT** statements.

Fortran

10.6.3 `assume` Directive

Name: `assume` Category: informational	Association: block **Properties:** pure

Clause groups

assumption

Semantics

The assumption scope of the **`assume`** directive is the corresponding region and any nested region of that region.

C / C++

10.6.4 `begin assumes` Directive

Name: `begin assumes` Category: informational	Association: delimited **Properties:** *default*

Clause groups

assumption

Semantics

The assumption scope of the **`begin assumes`** directive is the code that is executed and reached from any of the declared functions in the delimited code region. The delimited code region is a declaration sequence.

C / C++

10.7 `nothing` Directive

Name: `nothing` Category: utility	Association: unassociated **Properties:** pure, loop-transforming

Clauses

`apply`

Loop Modifiers for the `apply` Clause

loop-modifier	Number of Generated Loops	Description
`identity` *(default)*	1	the copy of the transformation-affected loop

Semantics

The `nothing` directive has no effect on the execution of the OpenMP program unless otherwise specified by the `apply` clause.

If the `nothing` directive immediately precedes a canonical loop nest then it forms a loop-transforming construct. It is associated with the outermost loop and generates one loop that has the same logical iterations in the same order as the transformation-affected loop.

Restrictions

- The `apply` clause can be specified if and only if the `nothing` directive forms a loop-transforming construct.

Cross References

- `apply` Clause, see Section 11.1
- Loop-Transforming Constructs, see Chapter 11

11 Loop-Transforming Constructs

A loop-transforming construct replaces itself, including its associated loop nest (see Section 6.4.1) or associated loop sequence (see Section 6.4.2), with a structured block that may be another loop nest or loop sequence. If the replacement of a loop-transforming construct is another loop nest or sequence, that loop nest or sequence, possibly as part of an enclosing loop nest or sequence, may be associated with another loop-nest-associated directive or loop-sequence-associated directive. A nested loop-transforming construct and any loop-transforming constructs that result from its **`apply`** clauses are replaced before any enclosing loop-transforming construct.

A loop-sequence-transforming construct generates a canonical loop sequence from its associated canonical loop sequence. The canonical loop nests that precede or follow the affected loop nests in the associated canonical loop sequence will respectively precede or follow, in the generated canonical loop sequence, the generated loop nest or generated loop sequence that replaces the affected loop nests.

All generated loops have canonical loop nest form, unless otherwise specified. Loop-iteration variables of generated loops are always private in the innermost enclosing parallelism-generating construct.

At the beginning of each logical iteration, the loop-iteration variable or the variable declared by *range-decl* has the value that it would have if the transformation-affected loop was not associated with any directive. After the execution of the loop-transforming construct, the loop-iteration variables of any of its transformation-affected loops have the values that they would have without the loop-transforming directive.

Restrictions

The following restrictions apply to loop-transforming constructs:

- The replacement of a loop-transforming construct with its generated loop nests or generated loop sequences must result in a conforming program.
- A generated loop of a loop-transforming construct must not be a doacross-affected loop.
- The arguments of any clauses on a loop-transforming construct must not refer to loop-iteration variables of surrounding loops in the same canonical loop nest.
- The *lb* and *ub* expressions of an affected loop (see Section 6.4.1) may only reference the loop-iteration variable of an enclosing loop affected by a loop-transforming construct if that loop-transforming construct has the nonrectangular-compatible property.

- A generated loop of a loop-transforming construct may only be a non-rectangular affected loop of an enclosing loop-nest-associated directive if that loop-transforming construct has the nonrectangular-compatible property.

Cross References

- Canonical Loop Nest Form, see Section 6.4.1

11.1 `apply` Clause

Name: `apply`	Properties: *default*

Arguments

Name	Type	Properties
applied-directives	list of directive specification list item type	*default*

Modifiers

Name	Modifies	Type	Properties
loop-modifier	*applied-directives*	Complex, Keyword: **`fused`**, **`grid`**, **`identity`**, **`interchanged`**, **`intratile`**, **`offsets`**, **`reversed`**, **`split`**, **`unrolled`** Arguments: ***indices*** list of expression of integer type (optional)	optional
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`fuse`, **`interchange`**, **`nothing`**, **`reverse`**, **`split`**, **`stripe`**, **`tile`**, **`unroll`**

Semantics

The **`apply`** clause applies loop-nest-associated constructs, specified by the *applied-directives* list, to generated loops of a loop-transforming construct. The *loop-modifier* specifies to which generated loops the directives are applied. If the loop-transforming construct generates a canonical loop sequence, the generated loops to which the directives are applied are the outermost loops of each generated loop nest. An applied loop-transforming construct may also specify **`apply`** clauses.

The valid *loop-modifier* keywords, the default *loop-modifier* if it exists, the number of *applied-directives* list items, and the target of each *applied-directives* list item is defined by the loop-transforming construct to which it applies. Each of the *indices* in the argument of the *loop-modifier* specifies the position of the generated loop to which the respective *applied-directives* item is applied.

If the *loop-modifier* is specified with no argument, the behavior is as if the list 1, 2, . . . , m is specified, where m is the number of generated loops according to the specification of the *loop-modifier* keyword. If the *loop-modifier* is omitted and a default *loop-modifier* exists for the **apply** clause on the construct, the behavior is as if the default *loop-modifier* with the argument 1, 2, . . . , m is specified.

The list items of the **apply** clause arguments are not required to be directive-wide unique.

Restrictions

Restrictions to the **apply** clause are as follows:

- Each list item in the *applied-directives* list of any **apply** clause must be **nothing** or the *directive-specification* of a loop-nest-associated construct.
- The loop-transforming construct on which the **apply** clause is specified must either have the generally-composable property or every list item in the *applied-directives* list of any **apply** clause must be the *directive-specification* of a loop-transforming directive.
- Every list item in the *applied-directives* list of any **apply** clause that is specified on a loop-transforming construct that is itself specified as a list item in the *applied-directives* list of another **apply** clause must be the *directive-specification* of a loop-transforming directive.
- For a given *loop-modifier* keyword, every *indices* list item may appear at most once in any **apply** clause on the directive.
- Every *indices* list item must be a positive constant less than or equal to m, the number of generated loops according to the specification of the *loop-modifier* keyword.
- The list items in *indices* must be in ascending order.
- If a directive does not define a default *loop-modifier* keyword, a *loop-modifier* is required.

Cross References

- **fuse** Construct, see Section 11.3
- **interchange** Construct, see Section 11.4
- **metadirective**, see Section 9.4.3
- **nothing** Directive, see Section 10.7
- **reverse** Construct, see Section 11.5
- **split** Construct, see Section 11.6
- **stripe** Construct, see Section 11.7
- **tile** Construct, see Section 11.8
- **unroll** Construct, see Section 11.9

11.2 `sizes` Clause

Name: `sizes`	**Properties:** unique, required

Arguments

Name	Type	Properties
size-list	list of OpenMP integer expression type	positive

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`stripe`, **`tile`**

Semantics

For a given loop-transforming directive on which the clause appears, the **`sizes`** clause specifies the manner in which the logical iteration space of the affected canonical loop nest is subdivided into m-dimensional grid cells that are relevant to the loop transformation, where m is the number of list items in *size-list*. Specificially, each list item in *size-list* specifies the size of the grid cells along the corresponding dimension. List items in *size-list* are not required to be unique.

Restrictions

Restrictions to the **`sizes`** clause are as follows:

- The loop nest depth of the associated loop nest of the loop-transforming construct on which the clause is specified must be greater than or equal to m.

Cross References

- **`stripe`** Construct, see Section 11.7
- **`tile`** Construct, see Section 11.8

11.3 `fuse` Construct

Name: `fuse` **Category:** executable	**Association:** loop sequence **Properties:** loop-transforming, order-concurrent-nestable, pure, simdizable, teams-nestable

Clauses

`apply`, **`looprange`**

Loop Modifiers for the `apply` Clause

loop-modifier	Number of Generated Loops	Description
`fused` *(default)*	1	the fused loop

Semantics

The **`fuse`** construct merges the affected loop nests specified by the **`looprange`** clause into a single canonical loop nest where execution of each logical iteration of the generated loop executes a logical iteration of each affected loop nest. Let $\ell^1, \ldots, \ell^n$ be the affected loop nests with $m^1, \ldots, m^n$ logical iterations each, and i_j^k the j^{th} logical iteration of loop ℓ^k. Let i_j^k be an empty iteration if $j \geq m^k$. Let m_{max} be the number of logical iterations of the affected loop nest with the most logical iterations. The loop generated by the **`fuse`** construct has m_{max} logical iterations, where execution of the j^{th} logical iteration executes the logical iterations $i_j^1, \ldots, i_j^n$, in that order.

Cross References

- **`apply`** Clause, see Section 11.1
- **`looprange`** Clause, see Section 6.4.7

11.4 `interchange` Construct

Name: **`interchange`** **Category:** executable	**Association:** loop nest **Properties:** loop-transforming, nonrectangular-compatible, order-concurrent-nestable, pure, simdizable, teams-nestable

Clauses

`apply`, **`permutation`**

Loop Modifiers for the `apply` Clause

loop-modifier	Number of Generated Loops	Description
`interchanged` *(default)*	n	the generated loops, in the new order

Semantics

The **`interchange`** construct has n transformation-affected loops, where $s_1, \ldots, s_n$ are the n items in the *permutation-list* argument of the **`permutation`** clause. Let $\ell_1, \ldots, \ell_n$ be the transformation-affected loops, from outermost to innermost. The original transformation-affected loops are replaced with the loops in the order $\ell_{s_1}, \ldots, \ell_{s_n}$. If the **`permutation`** clause is not specified, the effect is as if **`permutation(2,1)`** was specified.

Restrictions

Restrictions to the **interchange** clause are as follows:

- No transformation-affected loops may be a non-rectangular loop.
- The transformation-affected loops must be perfectly nested loops.

Cross References

- **apply** Clause, see Section 11.1
- **permutation** Clause, see Section 11.4.1

11.4.1 permutation Clause

Name: **permutation**	**Properties:** unique

Arguments

Name	Type	Properties
permutation-list	list of OpenMP integer expression type	constant, positive

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

interchange

Semantics

The **permutation** clause specifies a list of n positive constant expressions of integer OpenMP type.

Restrictions

Restrictions to the **permutation** clause are as follows:

- Every integer from 1 to n must appear exactly once in *permutation-list*.
- n must be at least 2.

Cross References

- **interchange** Construct, see Section 11.4

11.5 `reverse` Construct

Name: `reverse` **Category:** executable	**Association:** loop nest **Properties:** generally-composable, loop-transforming, order-concurrent-nestable, pure, simdizable, teams-nestable

Clauses

`apply`

Loop Modifiers for the `apply` Clause

loop-modifier	Number of Generated Loops	Description
`reversed` *(default)*	1	the reversed loop

Semantics

The **`reverse`** construct has one transformation-affected loop, the outermost loop, where $0, 1, \ldots, n-2, n-1$ are the logical iteration numbers of that loop. The construct transforms that loop into a loop in which iterations occur in the order $n-1, n-2, \ldots, 1, 0$.

Cross References

- **`apply`** Clause, see Section 11.1

11.6 `split` Construct

Name: `split` **Category:** executable	**Association:** loop nest **Properties:** generally-composable, loop-transforming, order-concurrent-nestable, pure, simdizable, teams-nestable

Clauses

`apply`, **`counts`**

Loop Modifiers for the `apply` Clause

loop-modifier	Number of Generated Loop Nests	Description
`split`	m	the loops of each logical iteration space partition

Semantics

The **split** loop-transforming construct implements index-set splitting, which partitions a logical iteration space into a sequence of smaller logical iteration spaces. It has one transformation-affected loop and generates a canonical loop sequence with m loop nests where m is the number of list items in the *count-list* argument of the **counts** clause. Let n be the number of logical iterations of the affected loop and $c_1, \ldots, c_m$ be the list items of the *count-list* argument. Let the k^{th} list item be the list item with the predefined identifier **omp_fill**. c_k is defined as

$$c_k = \max(0, n - \sum_{\substack{t=1 \\ t \neq k}}^{m} c_t)$$

Each generated loop in the sequence contains a copy of the loop body of the affected loop. The i^{th} generated loop executes the next c_i logical iterations except any logical iteration beyond the n original logical iterations.

Restrictions

The following restrictions apply to the **split** construct:

- Exactly one list item in the **counts** clause must be the predefined identifier **omp_fill**.

Cross References

- **apply** Clause, see Section 11.1
- **counts** Clause, see Section 11.6.1

11.6.1 counts Clause

Name: **counts**	**Properties:** unique, required

Arguments

Name	Type	Properties
count-list	list of OpenMP integer expression type	non-negative

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

split

Semantics

For a given loop-transforming directive on which the clause appears, the **counts** clause specifies the manner in which the logical iteration space of the transformation-affected loop is subdivided into n partitions, where m is the number of list items in *count-list* and where each partition is associated with a generated loop of the directive. Specifically, each list item in *count-list* specifies the iteration count of one of the generated loops. List items in *count-list* are not required to be unique.

Restrictions

Restrictions to the **counts** clause are as follows:

- A list item in *count-list* must be constant or **omp_fill**.

Cross References

- **split** Construct, see Section 11.6

11.7 stripe Construct

Name: **stripe** **Category:** executable	**Association:** loop nest **Properties:** loop-transforming, order-concurrent-nestable, pure, simdizable, teams-nestable

Clauses

apply, **sizes**

Loop Modifiers for the apply Clause

loop-modifier	Number of Generated Loops	Description
offsets	m	the offsetting loops $o_1, \ldots, o_m$
grid	m	the grid loops $g_1, \ldots, g_m$

Semantics

The **stripe** construct has m transformation-affected loops, where m is the number of list items in the *size-list* argument of the **sizes** clause, which consists of the list items $s_1, \ldots, s_m$. The construct has the effect of striping the execution order of the logical iterations across the grid cells of the logical iteration space that result from the **sizes** clause. Let $\ell_1, \ldots, \ell_m$ be the transformation-affected loops, from outermost to innermost, which the construct replaces with a canonical loop nest that consists of $2m$ perfectly nested loops. Let $o_1, \ldots, o_m, g_1, \ldots, g_m$ be the generated loops, from outermost to innermost. The loops $o_1, \ldots, o_m$ are the offsetting loops and the loops $g_1, \ldots, g_m$ are the grid loops.

Let $n_1, \ldots, n_m$ be number of logical iterations of each affected loop and $O = \{G_{\alpha_1,\ldots,\alpha_m} \mid \forall k \in \{1, \ldots, m\} : 0 \leq \alpha_1 < s_k\}$ the logical iteration vector space of the offsetting loops. The logical iteration $(i_1, \ldots, i_m)$ is executed in the logical iteration space of $G_{i_1 \bmod s_1, \ldots, i_m \bmod s_m}$.

The offsetting loops iterate over all $G_{\alpha_1, \ldots, \alpha_m}$ in lexicographic order of their indices and the grid loops iterate over the logical iteration space in the lexicographic order of the corresponding logical iteration vectors.

If an offsetting loop and a grid loop that are generated from the same **stripe** construct are affected loops of the same loop-nest-associated construct, the grid loops may execute additional empty logical iterations. The number of empty logical iterations is implementation defined.

Restrictions

Restrictions to the **stripe** construct are as follows:

- The transformation-affected loops must be perfectly nested loops.
- No transformation-affected loops may be a non-rectangular loop.

Cross References

- **apply** Clause, see Section 11.1
- Consistent Loop Schedules, see Section 6.4.4
- **sizes** Clause, see Section 11.2

11.8 tile Construct

Name: tile **Category:** executable	**Association:** loop nest **Properties:** loop-transforming, order-concurrent-nestable, pure, simdizable, teams-nestable

Clauses

apply, **sizes**

Loop Modifiers for the apply Clause

loop-modifier	Number of Generated Loops	Description
grid	m	the grid loops $g_1, \ldots, g_m$
intratile	m	the tile loops $t_1, \ldots, t_m$

Semantics

The **tile** construct has m transformation-affected loops, where m is the number of list items in the *size-list* argument of the **sizes** clause, which consists of list items $s_1, \ldots, s_m$. Let $\ell_1, \ldots, \ell_m$ be the transformation-affected loops, from outermost to innermost, which the construct replaces with a canonical loop nest that consists of $2m$ perfectly nested loops. Let $g_1, \ldots, g_m, t_1, \ldots, t_m$ be the generated loops, from outermost to innermost. The loops $g_1, \ldots, g_m$ are the grid loops and the loops $t_1, \ldots, t_m$ are the tile loops.

Let Ω be the logical iteration vector space of the transformation-affected loops. For any $(\alpha_1, \ldots, \alpha_m) \in \mathbb{N}^m$, define the set of iterations $\{(i_1, \ldots, i_m) \in \Omega \mid \forall k \in \{1, \ldots, m\} : s_k\alpha_k \leq i_k < s_k\alpha_k + s_k\}$ to be tile $T_{\alpha_1,\ldots,\alpha_m}$ and $G = \{T_{\alpha_1,\ldots,\alpha_m} \mid T_{\alpha_1,\ldots,\alpha_m} \neq \emptyset\}$ to be the set of tiles with at least one iteration. Tiles that contain $\prod_{k=1}^{m} s_k$ iterations are complete tile. Otherwise, they are partial tiles.

The grid loops iterate over all tiles $\{T_{\alpha_1,\ldots,\alpha_m} \in G\}$ in lexicographic order with respect to their indices $(\alpha_1, \ldots, \alpha_m)$ and the tile loops iterate over the iterations in $T_{\alpha_1,\ldots,\alpha_m}$ in the lexicographic order of the corresponding iteration vectors. An implementation may reorder the sequential execution of two iterations if at least one is from a partial tile and if their respective logical iteration vectors in *loop-nest* do not have a product order relation.

If a grid loop and a tile loop that are generated from the same **`tile`** construct are affected loops of the same loop-nest-associated construct, the tile loops may execute additional empty logical iterations. The number of empty logical iterations is implementation defined.

Restrictions

Restrictions to the **`tile`** construct are as follows:

- The transformation-affected loops must be perfectly nested loops.
- No transformation-affected loops may be a non-rectangular loop.

Cross References

- **`apply`** Clause, see Section 11.1
- Consistent Loop Schedules, see Section 6.4.4
- **`sizes`** Clause, see Section 11.2

11.9 `unroll` Construct

Name: **`unroll`** **Category:** executable	**Association:** loop nest **Properties:** generally-composable, loop-transforming, order-concurrent-nestable, pure, simdizable, teams-nestable

Clauses

`apply`, **`full`**, **`partial`**

Clause set

Properties: exclusive	**Members:** **`full`**, **`partial`**

Loop Modifiers for the `apply` Clause

loop-modifier	Number of Generated Loops	Description
`unrolled` *(default)*	1	the grid loop g_1 of the tiling step

Semantics

The **`unroll`** construct has one transformation-affected loop, which is unrolled according to its specified clauses. If no clauses are specified, if and how the loop is unrolled is implementation defined. The **`unroll`** construct results in a generated loop that has canonical loop nest form if and only if the **`partial`** clause is specified.

Restrictions

Restrictions to the **`unroll`** directive are as follows:

- The **`apply`** clause can only be specified if the **`partial`** clause is specified.

Cross References

- **`apply`** Clause, see Section 11.1
- **`full`** Clause, see Section 11.9.1
- **`partial`** Clause, see Section 11.9.2

11.9.1 `full` Clause

Name: `full`	**Properties:** unique

Arguments

Name	Type	Properties
fully_unroll	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`unroll`

Semantics

If *fully_unroll* evaluates to *true*, the **`full`** clause specifies that the transformation-affected loop is *fully unrolled*. The construct is replaced by a structured block that only contains n instances of its loop body, one for each of the n affected iterations and in their logical iteration order. If *fully_unroll* evaluates to *false*, the **`full`** clause has no effect. If *fully_unroll* is not specified, the effect is as if *fully_unroll* evaluates to *true*.

Restrictions

Restrictions to the **`full`** clause are as follows:

- The iteration count of the transformation-affected loop must be constant.

Cross References

- **`unroll`** Construct, see Section 11.9

11.9.2 `partial` Clause

Name: `partial`	**Properties:** unique

Arguments

Name	Type	Properties
unroll-factor	expression of integer type	optional, constant, positive

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`unroll`

Semantics

The **`partial`** clause specifies that the transformation-affected loop is first tiled with a tile size of *unroll-factor*. Then, the generated tile loop is fully unrolled. If the **`partial`** clause is used without an *unroll-factor* argument then *unroll-factor* is an implementation defined positive integer.

Cross References

- **`unroll`** Construct, see Section 11.9

12 Parallelism Generation and Control

This chapter defines constructs for generating and controlling parallelism.

12.1 parallel Construct

Name: parallel Category: executable	Association: block Properties: cancellable, context-matching, order-concurrent-nestable, parallelism-generating, team-generating, teams-nestable, thread-limiting

Clauses

allocate, **copyin**, **default**, **firstprivate**, **if**, **message**, **num_threads**, **private**, **proc_bind**, **reduction**, **safesync**, **severity**, **shared**

Binding

The binding thread set for a **parallel** region is the encountering thread. The encountering thread becomes the primary thread of the new team.

Semantics

When a thread encounters a **parallel** construct, a team is formed to execute the **parallel** region. The thread that encountered the **parallel** construct becomes the primary thread of the new team, with a thread number of zero for the duration of the new **parallel** region. All threads in the new team, including the primary thread, execute the region. Once the team is formed, the number of threads in the team is region-invariant and, so, does not change for the duration of that **parallel** region.

Within a **parallel** region, thread numbers uniquely identify each thread. Thread numbers are consecutive non-negative integers ranging from zero for the primary thread up to one less than the number of threads in the team. A thread may obtain its own thread number by a call to the **omp_get_thread_num** library routine.

A set of implicit tasks, equal in number to the number of threads in the team, is generated by the encountering thread. The structured block of the **parallel** construct determines the code that will be executed in each implicit task. Each task is assigned to a different thread in the team and becomes a tied. The task region of the task that the encountering thread is executing is suspended and each thread in the team executes its implicit task. Each thread can execute a path of statements that is different from that of the other threads.

The implementation may cause any thread to suspend execution of its implicit task at a task scheduling point, and to switch to execution of any explicit task generated by any of the threads in the team, before eventually resuming execution of the implicit task.

An implicit barrier occurs at the end of a **parallel** region. After the end of a **parallel** region, only the primary thread of the team resumes execution of the enclosing task region.

If a thread in a team that is executing a **parallel** region encounters another **parallel** directive, it forms a new team and becomes the primary thread of that new team.

If execution of a thread terminates while inside a **parallel** region, execution of all threads in all teams terminates. The order of termination of threads is unspecified. All work done by a team prior to any barrier that the team has passed in the program is guaranteed to be complete. The amount of work done by each thread after the last barrier that it passed and before it terminates is unspecified.

Unless a **requires** directive is specified on which the **device_safesync** clause appears, if the **parallel** construct is encountered on a non-host device and the **safesync** clause is not present then the behavior is as if the **safesync** clause appears on the directive with a *width* value that is implementation defined.

Execution Model Events

The *parallel-begin* event occurs in a thread that encounters a **parallel** construct before any implicit task is generated for the corresponding **parallel** region.

Upon generation of each implicit task, an *implicit-task-begin* event occurs in the thread that executes the implicit task after the implicit task is fully initialized but before the thread begins to execute the structured block of the **parallel** construct.

If a new native thread is created for the team that executes the **parallel** region upon encountering the construct, a *native-thread-begin* event occurs as the first event in the context of the new thread prior to the *implicit-task-begin* event.

Events associated with implicit barriers occur at the end of a **parallel** region. Section 17.3.2 describes events associated with implicit barriers.

When a thread completes an implicit task, an *implicit-task-end* event occurs in the thread after events associated with the implicit barrier synchronization in the implicit task.

The *parallel-end* event occurs in the thread that encounters the **parallel** construct after the thread executes its *implicit-task-end* event but before the thread resumes execution of the encountering task.

If a native thread is destroyed at the end of a **parallel** region, a *native-thread-end* event occurs in the worker thread that uses the native thread as the last event prior to destruction of the native thread.

Tool Callbacks

A thread dispatches a registered **parallel_begin** callback for each occurrence of a *parallel-begin* event in that thread. The callback occurs in the task that encounters the **parallel** construct. In the dispatched callback, **(***flags* **& ompt_parallel_team)** evaluates to *true*.

A thread dispatches a registered **implicit_task** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of an *implicit-task-begin* event in that thread. Similarly, a thread dispatches a registered **implicit_task** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of an *implicit-task-end* event in that thread. The callbacks occur in the context of the implicit task. In the dispatched callback, **(***flags* **& ompt_task_implicit)** evaluates to *true*.

A thread dispatches a registered **parallel_end** callback for each occurrence of a *parallel-end* event in that thread. The callback occurs in the task that encounters the **parallel** construct.

A thread dispatches a registered **thread_begin** callback for any *native-thread-begin* event in that thread. The callback occurs in the context of the thread.

A thread dispatches a registered **thread_end** callback for any *native-thread-end* event in that thread. The callback occurs in the context of the thread.

Cross References

- **allocate** Clause, see Section 8.6
- **copyin** Clause, see Section 7.8.1
- **default** Clause, see Section 7.5.1
- **firstprivate** Clause, see Section 7.5.4
- **if** Clause, see Section 5.5
- **implicit_task** Callback, see Section 34.5.3
- **message** Clause, see Section 10.3
- **num_threads** Clause, see Section 12.1.2
- **omp_get_thread_num** Routine, see Section 21.3
- Determining the Number of Threads for a **parallel** Region, see Section 12.1.1
- **parallel_begin** Callback, see Section 34.3.1
- **parallel_end** Callback, see Section 34.3.2
- OMPT **parallel_flag** Type, see Section 33.22
- **private** Clause, see Section 7.5.3
- **proc_bind** Clause, see Section 12.1.4
- **reduction** Clause, see Section 7.6.10

Algorithm 12.1 Determine Number of Threads

let *ThreadsBusy* be the number of threads currently executing tasks in this contention group;
let *StructuredThreadsBusy* be the number of structured threads currently executing tasks in this contention group;
if an **`if`** clause is specified **then let** *IfClauseValue* be the value of *if-expression*;
else let *IfClauseValue* = *true*;
if a **`num_threads`** clause is specified **then let** *ThreadsRequested* be the value of the first item of the *nthreads* list;
else let *ThreadsRequested* = value of the first element of *nthreads-var*;
let *ThreadsAvailable* = min(*thread-limit-var* - *ThreadsBusy*, *structured-thread-limit-var* - *StructuredThreadsBusy*) + 1;
if (*IfClauseValue* = *false*) **then** number of threads = 1;
else if (*active-levels-var* $\geq$ *max-active-levels-var*) **then** number of threads = 1;
else if (*dyn-var* = *true*) **and** (*ThreadsRequested* $\leq$ *ThreadsAvailable*)
then 1 $\leq$ number of threads $\leq$ *ThreadsRequested*;
else if (*dyn-var* = *true*) **and** (*ThreadsRequested* $>$ *ThreadsAvailable*)
then 1 $\leq$ number of threads $\leq$ *ThreadsAvailable*;
else if (*dyn-var* = *false*) **and** (*ThreadsRequested* $\leq$ *ThreadsAvailable*)
then number of threads = *ThreadsRequested*;
else if (*dyn-var* = *false*) **and** (*ThreadsRequested* $>$ *ThreadsAvailable*)
then behavior is implementation defined

- **`safesync`** Clause, see Section 12.1.5
- OMPT **`scope_endpoint`** Type, see Section 33.27
- **`severity`** Clause, see Section 10.4
- **`shared`** Clause, see Section 7.5.2
- OMPT **`task_flag`** Type, see Section 33.37
- **`thread_begin`** Callback, see Section 34.1.3
- **`thread_end`** Callback, see Section 34.1.4

12.1.1 Determining the Number of Threads for a parallel Region

When execution encounters a **parallel** directive, the value of the **if** clause or the first item of the *nthreads* list of the **num_threads** clause (if any) on the directive, the current parallel context, and the values of the *nthreads-var*, *dyn-var*, *thread-limit-var*, and *max-active-levels-var* ICVs are used to determine the number of threads to use in the region. When a thread encounters a **parallel** construct, the number of threads is determined according to Algorithm 12.1.

Using a variable in an *if-expression* of an **if** clause or in an element of the *nthreads* list of a **num_threads** clause of a **parallel** construct causes an implicit reference to the variable in all enclosing constructs. The *if-expression* and the *nthreads* list items are evaluated in the context outside of the **parallel** construct, and no ordering of those evaluations is specified. In what order or how many times any side effects of the evaluation of the *nthreads* list items or an *if-expression* occur is also unspecified.

Cross References

- *dyn-var* ICV, see Table 3.1
- *max-active-levels-var* ICV, see Table 3.1
- *nthreads-var* ICV, see Table 3.1
- *thread-limit-var* ICV, see Table 3.1
- **if** Clause, see Section 5.5
- **num_threads** Clause, see Section 12.1.2
- **parallel** Construct, see Section 12.1

12.1.2 num_threads Clause

Name: num_threads	**Properties:** unique

Arguments

Name	Type	Properties
nthreads	list of OpenMP integer expression type	positive

Modifiers

Name	Modifies	Type	Properties
prescriptiveness	*nthreads*	Keyword: **strict**	*default*
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

parallel

Semantics

The **num_threads** clause specifies the desired number of threads to execute a **parallel** region. Algorithm 12.1 determines the number of threads that execute the **parallel** region. If *prescriptiveness* is specified as **strict** and an implementation determines that Algorithm 12.1 would always result in a number of threads other than the value of the first item of the *nthreads* list then compile-time error termination may be performed in which case the effect of any **message** clause associated with the directive is implementation defined. Otherwise, if *prescriptiveness* is specified as **strict** and Algorithm 12.1 would result in a number of threads other than the value of the first item of the *nthreads* list then runtime error termination is performed. In both error termination scenarios, the effect is as if an **error** directive has been encountered on which any specified **message** and **severity** clauses and an **at** clause with **execution** as *action-time* are specified.

Cross References

- **at** Clause, see Section 10.2
- **error** Directive, see Section 10.1
- **message** Clause, see Section 10.3
- **parallel** Construct, see Section 12.1

12.1.3 Controlling OpenMP Thread Affinity

When a thread encounters a **parallel** directive without a **proc_bind** clause, the *bind-var* ICV is used to determine the policy for assigning threads to places within the input place partition, as defined in the following paragraph. If the **parallel** directive has a **proc_bind** clause then the thread affinity policy specified by the **proc_bind** clause overrides the policy specified by the first element of the *bind-var* ICV. Once a thread in the team is assigned to a place, the OpenMP implementation should not move it to another place.

If the encountering thread is a free-agent thread that is executing an explicit task that was created in an implicit parallel region, the input place partition for all thread affinity policies is the value of the *place-partition-var* ICV of the initial task. If the encountering thread is a free-agent thread that is executing an explicit task that was created in an explicit parallel region, the input place partition for all thread affinity policies is the input place partition of that parallel region. If the encountering thread is not a free-agent thread, the input place partition for all thread affinity policies is the value of the *place-partition-var* ICV of its binding implicit task.

Under the **primary** and **close** thread affinity policies, the *place-partition-var* ICV of each implicit task is assigned the input place partition. As discussed below, under the **spread** thread affinity policy, the *place-partition-var* ICV of each implicit task is derived from the value of the input place partition.

TABLE 12.1: Affinity-related Symbols used in this Section

Symbol	Symbol Description
L	the value of the *thread-limit-var* ICV
NG	the total number of place-assignment groups
g_i	the i^{th} place-assignment group
P	the number of places in the input place partition
T	the number of threads in the team
AT	$\lceil T/NG \rceil$ ("above-thread" count)
BT	$\lfloor T/NG \rfloor$ ("below-thread" count)
ET	$T \bmod NG$ ("excess-thread" count)

The *place-assignment-var* ICV is a list of L place numbers, where L is the value of the *thread-limit-var* ICV, that defines the place assignment of threads that participate in the execution of tasks bound to a given team. Any such thread corresponds to a position in the list, meaning it will be assigned to the place given by the place number at that position. If a thread is an assigned thread of the team with thread number i, it corresponds to position i in the *place-assignment-var* list. If a thread is a free-agent thread, it corresponds to the first position for which another thread has not yet been assigned to the associated place. If another thread is already assigned to the place associated with that position, the place to which the free-agent thread is assigned is implementation defined.

Each thread affinity policy determines how threads are assigned to places. A policy assigns each place in the input place partition to one of NG place-assignment groups, $g_0, \ldots, g_{NG-1}$; additionally, it assigns each position from the *place-assignment-var* ICV to one of these groups. In a given group, the place number of each place is then assigned to a *place-assignment-var* position, in round robin fashion, starting with the first place. Threads are thus assigned to places according to the resulting *place-assignment-var* of the policy.

Under the **`primary`** thread affinity policy, $NG = 1$ and place-assignment group g_0 is assigned the place to which the encountering thread is assigned, and all positions of *place-assignment-var* are assigned to the same group. Thus, the corresponding threads of all positions of the *place-assignment-var* ICV are assigned to the same place as the primary thread.

For the **`close`** and **`spread`** thread affinity policies, let P be the number of places in the input place partition and let T be the number of assigned threads in the team. The following paragraphs describe how places in the input place partition are subdivided into place-assignment groups for these policies. A general description of how positions in *place-assignment-var* are assigned to these places, and thus how place assignment for threads under the policies is determined, then follows these descriptions.

The **close** thread affinity policy distributes assignment of places evenly across a team of threads, while ensuring threads with consecutive numbers are assigned to the same place or adjacent places. Each place in the input place partition is assigned to one place-assignment group (so, $NG = P$). Place-assignment group g_0 is assigned the place to which the encountering thread is assigned. The place assigned to group g_i is then the next place in the place partition of the one assigned to group g_{i-1}, with wrap around with respect to the input place partition.

The **spread** thread affinity policy creates a sparse distribution for a team of T threads among the P places of the input place partition. A sparse distribution is achieved by first subdividing the input place partition into T subpartitions if $T \leq P$ (in which case $NG = T$), or P subpartitions if $T > P$ (in which case $NG = P$). The subpartitions are determined as follows:

- $T \leq P$: The input place partition is split into T subpartitions, where each subpartition contains $\lfloor P/T \rfloor$ or $\lceil P/T \rceil$ consecutive places; if $P mod T$ is not zero, which subpartitions contain $\lceil P/T \rceil$ places is implementation defined;
- $T > P$: The input place partition is split into P subpartitions, each with a single place.

In either case, the places from each subpartition are assigned to a place-assignment group that corresponds to the subpartition. The subpartition that corresponds to group g_0 is the one that includes the place on which the encountering thread is executing. The subpartition that corresponds to group g_i is the one that includes the next place to those in the subpartition corresponding to group g_{i-1}, with wrap around with respect to the input place partition. For a given implicit task and corresponding *place-assignment-var* position to its assigned thread, the *place-partition-var* ICV of the implicit task is set to the subpartition that corresponds to the group that includes the position. Thus, the subpartitioning is not only a mechanism for achieving a sparse distribution, it also defines a subset of places for a thread to use when creating a nested **parallel** region.

Let AT equal $\lceil T/NG \rceil$, BT equal $\lfloor T/NG \rfloor$, and ET equal $T \bmod NG$. The **close** and the **spread** thread affinity policies assign the positions of the *place-assignment-var* ICV to place-assignment groups as follows.

- For positions from 0 up to $T - 1$: The positions are partitioned into NG sets of consecutive positions, ET of which have AT positions and $NG - ET$ of which have only BT positions (when ET is not zero, which sets have which count is implementation defined unless the thread affinity policy is **close** and $T < P$, in which case the first T groups are assigned the sets with AT positions). The sets are assigned to each group, with the first set, starting at position 0, assigned to group g_0, and with each successive set i, starting at the position immediately after the last position in the set assigned to group g_{i-1}, assigned to the next group g_i;
- If $ET \neq 0$, for the positions from T up to $(AT * NG) - 1$: Each of these positions is assigned to a group g_i that received only BT positions in the above step, such that each such g_i is then assigned AT positions (which positions are assigned to which group is implementation defined);

- For the remaining positions from $AT * NG$ up to L: Each position is assigned to a group in round robin fashion, starting with the first group g_0.

The determination of whether the thread affinity request can be fulfilled is implementation defined. If it cannot be fulfilled, then the affinity of threads in the team is implementation defined.

Note – Wrap around is needed if the end of a place partition is reached before all thread assignments are done. For example, wrap around may be needed in the case of **close** and $T \leq P$, if the primary thread is assigned to a place other than the first place in the place partition. In this case, thread 1 is assigned to the place after the place of the primary thread, thread 2 is assigned to the place after that, and so on. The end of the place partition may be reached before all threads are assigned. In this case, assignment of threads is resumed with the first place in the place partition.

Cross References

- *bind-var* ICV, see Table 3.1
- *place-assignment-var* ICV, see Table 3.1
- *place-partition-var* ICV, see Table 3.1
- *thread-limit-var* ICV, see Table 3.1
- **parallel** Construct, see Section 12.1
- **proc_bind** Clause, see Section 12.1.4

12.1.4 proc_bind Clause

Name: proc_bind	**Properties:** unique

Arguments

Name	Type	Properties
affinity-policy	Keyword: **close**, **primary**, **spread**	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

parallel

Semantics

The **proc_bind** clause specifies the mapping of threads to places within the input place partition. The effect of the possible values for *affinity-policy* are described in Section 12.1.3

Cross References

- Controlling OpenMP Thread Affinity, see Section 12.1.3
- **parallel** Construct, see Section 12.1

12.1.5 safesync Clause

Name: safesync	Properties: unique

Arguments

Name	Type	Properties
width	expression of integer type	positive, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

parallel

Semantics

The **safesync** clause determines whether two synchronizing threads in a team can make progress (see Section 1.2). The clause specifies that threads in the new team are partitioned, in thread number order, into progress groups of size *width*, except for the last progress group, which may contain less than *width* threads. Among threads that are executing tasks in the same contention group in parallel, only threads that are in the same progress group may execute in the same progress unit. If the *width* argument is not specified, the behavior is as if the *width* argument is one.

Restrictions

Restrictions to the **safesync** clause are as follows:

- The *width* argument must be a **safesync**-compatible expression.

Cross References

- **parallel** Construct, see Section 12.1

12.2 teams Construct

Name: teams Category: executable	Association: block Properties: parallelism-generating, team-generating, thread-limiting, context-matching

Clauses

allocate, **default**, **firstprivate**, **if**, **num_teams**, **private**, **reduction**, **shared**, **thread_limit**

Binding

The binding thread set for a **teams** region is the encountering thread.

Semantics

When a thread encounters a **teams** construct, a league of teams is created. Each team is an initial team, and the initial thread in each team executes the **teams** region. The number of teams created is determined by evaluating the **if** and **num_teams** clauses. Once the teams are created, the number of initial teams are region-invariant , thus do not change for the duration of the **teams** region. Within a **teams** region, initial team numbers uniquely identify each initial team. Initial teams numbers are consecutive non-negative integers ranging from zero to one less than the number of initial teams.

When an **if** clause is present on a **teams** construct and the **if** clause expression evaluates to *false*, the number of formed teams is one. The use of a variable in an **if** clause expression of a **teams** construct causes an implicit reference to the variable in all enclosing constructs. The **if** clause expression is evaluated in the context outside of the **teams** construct.

If a **thread_limit** clause is not present on the **teams** construct, but the construct is closely nested inside a **target** construct on which the **thread_limit** clause is specified, the behavior is as if that **thread_limit** clause is also specified for the **teams** construct.

The place list, given by the *place-partition-var* ICV of the encountering thread, is split into subpartitions in an implementation defined manner, and each team is assigned to a subpartition by setting the *place-partition-var* of its initial thread to the subpartition.

The **teams** construct sets the *default-device-var* ICV for each initial thread to an implementation defined value.

After the teams have completed execution of the **teams** region, the encountering task resumes execution of the enclosing task region.

Execution Model Events

The *teams-begin* event occurs in a thread that encounters a **teams** construct before any initial task is generated for the corresponding **teams** region.

Upon generation of each initial task, an *initial-task-begin* event occurs in the thread that executes the initial task after the initial task is fully initialized but before the thread begins to execute the structured block of the **teams** construct.

If a new native thread is created for the league of teams that executes the **teams** region upon encountering the construct, a *native-thread-begin* event occurs as the first event in the context of the new thread prior to the *initial-task-begin* event.

When a thread completes an initial task, an *initial-task-end* event occurs in the thread.

The *teams-end* event occurs in the thread that encounters the **teams** construct after the thread executes its *initial-task-end* event but before it resumes execution of the encountering task.

If a native thread is destroyed at the end of a **teams** region, a *native-thread-end* event occurs in the initial thread that uses the native thread as the last event prior to destruction of the native thread.

Tool Callbacks

A thread dispatches a registered **parallel_begin** callback for each occurrence of a *teams-begin* event in that thread. The callback occurs in the task that encounters the **teams** construct. In the dispatched callback, **(***flags* **& ompt_parallel_league)** evaluates to *true*.

A thread dispatches a registered **implicit_task** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of an *initial-task-begin* event in that thread. Similarly, a thread dispatches a registered **implicit_task** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of an *initial-task-end* event in that thread. The callbacks occur in the context of the initial task. In the dispatched callback, **(***flags* **& ompt_task_initial)** and **(***flags* **& ompt_task_implicit)** evaluate to *true*.

A thread dispatches a registered **parallel_end** callback for each occurrence of a *teams-end* event in that thread. The callback occurs in the task that encounters the **teams** construct.

A thread dispatches a registered **thread_begin** callback for each *native-thread-begin* event in that thread. The callback occurs in the context of the thread.

A thread dispatches a registered **thread_end** callback for each *native-thread-end* event in that thread. The callback occurs in the context of the thread.

Restrictions

Restrictions to the **teams** construct are as follows:

- If a *reduction-modifier* is specified in a **reduction** clause that appears on the directive then the *reduction-modifier* must be **default**.
- A **teams** region must be a strictly nested region of the implicit parallel region that surrounds the whole OpenMP program or a **target** region. If a **teams** region is nested inside a **target** region, the corresponding **target** construct must not contain any statements, declarations or directives outside of the corresponding **teams** construct.
- For a **teams** construct that is an immediately nested construct of a **target** construct, the bounds expressions of any array sections and the index expressions of any array elements

used in any clause on the construct, as well as all expressions of any **target**-consistent clauses on the construct, must be **target**-consistent expressions.

- Only regions that are generated by **teams**-nestable constructs or **teams**-nestable routines may be strictly nested regions of **teams** regions.

Cross References

- **allocate** Clause, see Section 8.6
- **default** Clause, see Section 7.5.1
- **distribute** Construct, see Section 13.7
- **firstprivate** Clause, see Section 7.5.4
- *default-device-var* ICV, see Table 3.1
- *place-partition-var* ICV, see Table 3.1
- **if** Clause, see Section 5.5
- **implicit_task** Callback, see Section 34.5.3
- **num_teams** Clause, see Section 12.2.1
- **omp_get_num_teams** Routine, see Section 22.1
- **omp_get_team_num** Routine, see Section 22.3
- **parallel** Construct, see Section 12.1
- **parallel_begin** Callback, see Section 34.3.1
- **parallel_end** Callback, see Section 34.3.2
- OMPT **parallel_flag** Type, see Section 33.22
- **private** Clause, see Section 7.5.3
- **reduction** Clause, see Section 7.6.10
- OMPT **scope_endpoint** Type, see Section 33.27
- **shared** Clause, see Section 7.5.2
- **target** Construct, see Section 15.8
- OMPT **task_flag** Type, see Section 33.37
- **thread_begin** Callback, see Section 34.1.3
- **thread_end** Callback, see Section 34.1.4
- **thread_limit** Clause, see Section 15.3

12.2.1 `num_teams` Clause

Name: `num_teams`	**Properties:** target-consistent, unique

Arguments

Name	Type	Properties
upper-bound	expression of integer type	positive

Modifiers

Name	Modifies	Type	Properties
lower-bound	*upper-bound*	OpenMP integer expression	positive, ultimate, unique
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`teams`

Semantics

The **`num_teams`** clause specifies the bounds on the number of teams formed by the construct on which it appears. *lower-bound* specifies the lower bound and *upper-bound* specifies the upper bound on the number of teams requested. If *lower-bound* is not specified, the effect is as if *lower-bound* is specified as equal to *upper-bound*. The number of teams formed is implementation defined, but it will be greater than or equal to the lower bound and less than or equal to the upper bound.

If the **`num_teams`** clause is not specified on a construct then the effect is as if *upper-bound* was specified as follows. If the value of the *nteams-var* ICV is greater than zero, the effect is as if *upper-bound* was specified as an implementation defined value greater than zero but less than or equal to the value of the *nteams-var* ICV. Otherwise, the effect is as if *upper-bound* was specified as an implementation defined value greater than or equal to one.

Restrictions

- *lower-bound* must be less than or equal to *upper-bound*.

Cross References

- *nteams-var* ICV, see Table 3.1
- **`teams`** Construct, see Section 12.2

12.3 `order` Clause

Name: `order`	**Properties:** schedule-specification, unique

Arguments

Name	Type	Properties
ordering	Keyword: **concurrent**	*default*

Modifiers

Name	Modifies	Type	Properties
order-modifier	*ordering*	Keyword: **reproducible**, **unconstrained**	*default*
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

distribute, **do**, **for**, **loop**, **simd**

Semantics

The **order** clause specifies an *ordering* of execution for the collapsed iterations of a loop-collapsing construct. If *ordering* is **concurrent**, different collapsed iterations may execute in any order, including in parallel, as if by the binding thread set of the region. The binding thread set may recruit or create additional native threads to participate in the parallel execution of any collapsed iterations.

The *order-modifier* on the **order** clause affects the schedule specification for the purpose of determining its consistency with other schedules (see Section 6.4.4). If *order-modifier* is **reproducible**, the loop schedule for the construct on which the clause appears is reproducible, whereas if *order-modifier* is **unconstrained**, the loop schedule is not reproducible.

Restrictions

Restrictions to the **order** clause are as follows:

- The only routines for which a call may be nested inside a region that corresponds to a construct on which the **order** clause is specified with **concurrent** as the *ordering* argument are **order-concurrent**-nestable routines.
- Only regions that correspond to **order-concurrent**-nestable constructs or **order-concurrent**-nestable routines may be strictly nested regions of regions that correspond to constructs on which the **order** clause is specified with **concurrent** as the *ordering* argument.
- If a threadprivate variable is referenced inside a region that corresponds to a construct with an **order** clause that specifies **concurrent**, the behavior is unspecified.

Cross References

- **distribute** Construct, see Section 13.7
- **do** Construct, see Section 13.6.2

- **for** Construct, see Section 13.6.1
- **loop** Construct, see Section 13.8
- **simd** Construct, see Section 12.4

12.4 simd Construct

Name: simd	**Association:** loop nest
Category: executable	**Properties:** context-matching, order-concurrent-nestable, parallelism-generating, pure, simdizable

Separating directives

scan

Clauses

aligned, **collapse**, **if**, **induction**, **lastprivate**, **linear**, **nontemporal**, **order**, **private**, **reduction**, **safelen**, **simdlen**

Binding

A **simd** region binds to the current task region. The binding thread set of the **simd** region is the current team.

Semantics

The **simd** construct enables the execution of multiple collapsed iterations concurrently by using SIMD instructions. The number of collapsed iterations that are executed concurrently at any given time is implementation defined. Each concurrent iteration will be executed by a different SIMD lane. Each set of concurrent iterations is a SIMD chunk. Lexical forward dependences in the iterations of the original loop must be preserved within each SIMD chunk, unless an **order** clause that specifies **concurrent** is present.

When an **if** clause is present with an *if-expression* that evaluates to *false*, the preferred number of iterations to be executed concurrently is one, regardless of whether a **simdlen** clause is specified.

Restrictions

Restrictions to the **simd** construct are as follows:

- If both **simdlen** and **safelen** clauses are specified, the value of the **simdlen** *length* must be less than or equal to the value of the **safelen** *length*.
- Only SIMDizable constructs may be encountered during execution of a **simd** region.
- If an **order** clause that specifies **concurrent** appears on a **simd** directive, the **safelen** clause must not also appear.

C / C++

- The **simd** region cannot contain calls to the **longjmp** or **setjmp** functions.

C / C++

C++

- No exceptions can be raised in the **simd** region.
- The only random access iterator types that are allowed for the collapsed loops are pointer types.

C++

Cross References

- **aligned** Clause, see Section 7.12
- **collapse** Clause, see Section 6.4.5
- **if** Clause, see Section 5.5
- **induction** Clause, see Section 7.6.13
- **lastprivate** Clause, see Section 7.5.5
- **linear** Clause, see Section 7.5.6
- **nontemporal** Clause, see Section 12.4.1
- **order** Clause, see Section 12.3
- **private** Clause, see Section 7.5.3
- **reduction** Clause, see Section 7.6.10
- **safelen** Clause, see Section 12.4.2
- **scan** Directive, see Section 7.7
- **simdlen** Clause, see Section 12.4.3

12.4.1 nontemporal Clause

Name: **nontemporal**	**Properties:** *default*

Arguments

Name	Type	Properties
list	list of variable list item type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

simd

Semantics

The **nontemporal** clause specifies that accesses to the storage locations to which the list items refer have low temporal locality across the logical iterations in which those storage locations are accessed. The list items of the **nontemporal** clause may also appear as list items of data-environment attribute clauses.

Cross References

- **simd** Construct, see Section 12.4

12.4.2 safelen Clause

Name: safelen	Properties: unique

Arguments

Name	Type	Properties
length	expression of integer type	positive, constant

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

simd

Semantics

The **safelen** clause specifies that no two concurrent logical iterations within a SIMD chunk can have a distance in the collapsed iteration space that is greater than or equal to the *length* argument.

Cross References

- **simd** Construct, see Section 12.4

12.4.3 simdlen Clause

Name: simdlen	Properties: unique

Arguments

Name	Type	Properties
length	expression of integer type	positive, constant

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

declare_simd, **simd**

Semantics

When the **simdlen** clause appears on a **simd** construct, *length* is treated as a hint that specifies the preferred number of collapsed iterations to be executed concurrently. When the **simdlen** clause appears on a **declare_simd** directive, if a SIMD version of the associated procedure is created, *length* corresponds to the number of concurrent arguments of the procedure.

Cross References

- **declare_simd** Directive, see Section 9.8
- **simd** Construct, see Section 12.4

12.5 masked Construct

Name: masked **Category:** executable	**Association:** block **Properties:** thread-limiting, thread-selecting

Clauses

filter

Binding

The binding thread set for a **masked** region is the current team. A **masked** region binds to the innermost enclosing parallel region.

Semantics

The **masked** construct specifies a structured block that is executed by a subset of the threads of the current team. The **filter** clause selects a subset of the threads of the team that executes the binding parallel region to execute the structured block of the **masked** region. Other threads in the team do not execute the associated structured block. No implied barrier occurs either on entry to or exit from the **masked** construct. The result of evaluating the *thread_num* argument of the **filter** clause may vary across threads.

If more than one thread in the team executes the structured block of a **masked** region, the structured block must include any synchronization required to ensure that data races do not occur.

Execution Model Events

The *masked-begin* event occurs in any thread of a team that executes the **masked** region on entry to the region. The *masked-end* event occurs in any thread of a team that executes the **masked** region on exit from the region.

Tool Callbacks

A thread dispatches a registered **masked** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *masked-begin* event in that thread. Similarly, a thread dispatches a registered **masked** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of a *masked-end* event in that thread. These callbacks occur in the context of the task executed by the encountering thread.

Cross References

- **filter** Clause, see Section 12.5.1
- **masked** Callback, see Section 34.3.3
- OMPT **scope_endpoint** Type, see Section 33.27

12.5.1 filter Clause

Name: **filter**	**Properties:** unique

Arguments

Name	Type	Properties
thread_num	expression of integer type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

masked

Semantics

If *thread_num* specifies the thread number of the encountering thread in the current team then the **filter** clause selects the encountering thread. If the **filter** clause is not specified, the effect is as if the clause is specified with *thread_num* equal to zero, so that the **filter** clause selects the primary thread. The use of a variable in a *thread_num* argument expression causes an implicit reference to the variable in all enclosing constructs.

Cross References

- **masked** Construct, see Section 12.5

13 Work-Distribution Constructs

A work-distribution construct distributes the execution of the corresponding region among the threads in its binding thread set. Threads execute portions of the region in the context of the implicit tasks that each thread is executing.

A work-distribution construct is a worksharing construct if the binding thread set is a team. A worksharing region has no barrier on entry. However, an implied barrier exists at the end of the worksharing region, unless a **nowait** clause is specified with *do_not_synchronize* specified as *true*, in which case an implementation may omit the barrier at the end of the worksharing region. In this case, threads that finish early may proceed straight to the instructions that follow the worksharing region without waiting for the other members of the team to finish the worksharing region, and without performing a flush operation.

If a work-distribution construct is a partitioned construct then all user code encountered in the region, but not in a nested region that is not a closely nested region, is executed by one thread from the binding thread set.

For loop-nest-associated constructs, the loop schedule is determined by a schedule specification for the construct, which is defined by schedule-specification clauses and (where applicable) the *run-sched-var* ICV. OpenMP programs can only depend on which thread executes a particular collapsed iteration if the construct specifies a reproducible schedule. Schedule reproducibility also determines whether constructs with the same schedule specification will have consistent schedules (see Section 6.4.4).

Restrictions

The following restrictions apply to work-distribution constructs:

- Each work-distribution region must be encountered by all threads in the binding thread set or by none at all unless cancellation has been requested for the innermost enclosing parallel region.
- The sequence of encountered work-distribution regions that have the same binding thread set must be the same for every thread in the binding thread set.
- The sequence of encountered worksharing regions and **barrier** regions that bind to the same team must be the same for every thread in the team.

Fortran

- A variable must not be private within a **teams** or **parallel** region if it has either **LOCAL_INIT** or **SHARED** locality in a **DO CONCURRENT** loop that is associated with a work-distribution construct, where the **teams** or **parallel** region is a binding region of the corresponding work-distribution region.

Fortran

13.1 single Construct

Name: single **Category:** executable	**Association:** block **Properties:** work-distribution, team-executed, partitioned, worksharing, thread-limiting, thread-selecting

Clauses

allocate, **copyprivate**, **firstprivate**, **nowait**, **private**

Clause set

Properties: exclusive	**Members:** **copyprivate**, **nowait**

Binding

The binding thread set for a **single** region is the current team. A **single** region binds to the innermost enclosing parallel region. Only the threads of the team that executes the binding parallel region participate in the execution of the structured block and the implied barrier of the **single** region if the barrier is not eliminated by a **nowait** clause.

Semantics

The **single** construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the primary thread), in the context of its implicit task. The method of choosing a thread to execute the structured block each time the team encounters the construct is implementation defined. An implicit barrier occurs at the end of a **single** region if the **nowait** clause does not specify otherwise.

Execution Model Events

The *single-begin* event occurs after an implicit task encounters a **single** construct but before the task starts to execute the structured block of the **single** region. The *single-end* event occurs after an implicit task finishes execution of a **single** region but before it resumes execution of the enclosing region.

Tool Callbacks

A thread dispatches a registered **work** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *single-begin* event in that thread. Similarly, a thread dispatches a registered **work** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of a *single-end* event in that thread. For each of these callbacks, the *work_type* argument is **ompt_work_single_executor** if the thread executes the structured block associated with the **single** region; otherwise, the *work_type* argument is **ompt_work_single_other**.

Cross References

- **allocate** Clause, see Section 8.6
- **copyprivate** Clause, see Section 7.8.2
- **firstprivate** Clause, see Section 7.5.4
- **nowait** Clause, see Section 17.6
- **private** Clause, see Section 7.5.3
- OMPT **scope_endpoint** Type, see Section 33.27
- **work** Callback, see Section 34.4.1
- OMPT **work** Type, see Section 33.41

13.2 scope Construct

Name: scope **Category:** executable	**Association:** block **Properties:** work-distribution, team-executed, worksharing, thread-limiting

Clauses

allocate, **firstprivate**, **nowait**, **private**, **reduction**

Binding

The binding thread set for a **scope** region is the current team. A **scope** region binds to the innermost enclosing parallel region. Only the threads of the team that executes the binding parallel region participate in the execution of the structured block and the implied barrier of the **scope** region if the barrier is not eliminated by a **nowait** clause.

Semantics

The **scope** construct specifies that all threads in a team execute the associated structured block and any additionally specified OpenMP operations. An implicit barrier occurs at the end of a **scope** region if the **nowait** clause does not specify otherwise.

Execution Model Events

The *scope-begin* event occurs after an implicit task encounters a **scope** construct but before the task starts to execute the structured block of the **scope** region. The *scope-end* event occurs after an implicit task finishes execution of a **scope** region but before it resumes execution of the enclosing region.

Tool Callbacks

A thread dispatches a registered **work** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_work_scope** as its *work_type* argument for each occurrence of a *scope-begin* event in that thread. Similarly, a thread dispatches a registered **work** callback with

ompt_scope_end as its *endpoint* argument and **ompt_work_scope** as its *work_type* argument for each occurrence of a *scope-end* event in that thread. The callbacks occur in the context of the implicit task.

Cross References

- **allocate** Clause, see Section 8.6
- **firstprivate** Clause, see Section 7.5.4
- **nowait** Clause, see Section 17.6
- **private** Clause, see Section 7.5.3
- **reduction** Clause, see Section 7.6.10
- OMPT **scope_endpoint** Type, see Section 33.27
- **work** Callback, see Section 34.4.1
- OMPT **work** Type, see Section 33.41

13.3 sections Construct

Name: sections **Category:** executable	**Association:** block **Properties:** work-distribution, team-executed, partitioned, worksharing, thread-limiting, cancellable

Separating directives

section

Clauses

allocate, **firstprivate**, **lastprivate**, **nowait**, **private**, **reduction**

Binding

The binding thread set for a **sections** region is the current team. A **sections** region binds to the innermost enclosing parallel region. Only the threads of the team that executes the binding parallel region participate in the execution of the structured block sequences and the implied barrier of the **sections** region if the barrier is not eliminated by a **nowait** clause.

Semantics

The **sections** construct is a non-iterative worksharing construct that contains a structured block that consists of a set of structured block sequences that are to be distributed among and executed by the threads in a team. Each structured block sequence is executed by one of the threads in the team in the context of its implicit task. An implicit barrier occurs at the end of a **sections** region if the **nowait** clause does not specify otherwise.

Each structured block sequence in the **sections** construct is preceded by a **section** subsidiary directive except possibly the first sequence, for which a preceding **section** subsidiary directive is optional. The method of scheduling the structured block sequences among the threads in the team is implementation defined.

Execution Model Events

The *sections-begin* event occurs after an implicit task encounters a **sections** construct but before the task executes any structured block sequences of the **sections** region. The *sections-end* event occurs after an implicit task finishes execution of a **sections** region but before it resumes execution of the enclosing context.

Tool Callbacks

A thread dispatches a registered **work** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_work_sections** as its *work_type* argument for each occurrence of a *sections-begin* event in that thread. Similarly, a thread dispatches a registered **work** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_work_sections** as its *work_type* argument for each occurrence of a *sections-end* event in that thread. The callbacks occur in the context of the implicit task.

Cross References

- **allocate** Clause, see Section 8.6
- **firstprivate** Clause, see Section 7.5.4
- **lastprivate** Clause, see Section 7.5.5
- **nowait** Clause, see Section 17.6
- **private** Clause, see Section 7.5.3
- **reduction** Clause, see Section 7.6.10
- OMPT **scope_endpoint** Type, see Section 33.27
- **section** Directive, see Section 13.3.1
- **work** Callback, see Section 34.4.1
- OMPT **work** Type, see Section 33.41

13.3.1 section Directive

Name: section **Category:** subsidiary	**Association:** separating **Properties:** *default*

Separated directives

sections

Semantics

The **section** directive splits a structured block sequence that is associated with a **sections** construct into two structured block sequences.

Execution Model Events

The *section-begin* event occurs before an implicit task starts to execute a structured block sequence in the **sections** construct for each of those structured block sequences that the task executes.

Tool Callbacks

A thread dispatches a registered **dispatch** callback for each occurrence of a *section-begin* event in that thread. The callback occurs in the context of the implicit task.

Cross References

- **dispatch** Callback, see Section 34.4.2
- **sections** Construct, see Section 13.3

Fortran

13.4 workshare Construct

Name: workshare **Category:** executable	**Association:** block **Properties:** work-distribution, team-executed, partitioned, worksharing

Clauses

nowait

Binding

The binding thread set for a **workshare** region is the current team. A **workshare** region binds to the innermost enclosing parallel region. Only the threads of the team that executes the binding parallel region participate in the execution of the units of work and the implied barrier of the **workshare** region if the barrier is not eliminated by a **nowait** clause.

Semantics

The **workshare** construct divides the execution of the associated structured block into separate units of work and causes the threads of the team to share the work such that each unit of work is executed only once by one thread, in the context of its implicit task. An implicit barrier occurs at the end of a **workshare** region if a **nowait** clause does not specify otherwise.

An implementation of the **workshare** construct must insert any synchronization that is required to maintain Fortran semantics. For example, the effects of each statement within the structured block must appear to occur before the execution of the following statements, and the evaluation of the right hand side of an assignment must appear to complete prior to the effects of assigning to the left hand side.

The statements in the **workshare** construct are divided into units of work as follows:

- For array expressions within each statement, including transformational array intrinsic functions that compute scalar values from arrays:
  - Evaluation of each element of the array expression, including any references to elemental functions, is a unit of work.
  - Evaluation of transformational array intrinsic functions may be subdivided into any number of units of work.
- For array assignment statements, assignment of each element is a unit of work.
- For scalar assignment statements, each assignment operation is a unit of work.
- For **WHERE** statements or constructs, evaluation of the mask expression and the masked assignments are each a unit of work.
- For **FORALL** statements or constructs, evaluation of the mask expression, expressions occurring in the specification of the iteration space, and the masked assignments are each a unit of work.
- For **atomic** constructs, **critical** constructs, and **parallel** constructs, the construct is a unit of work. A new team executes the statements contained in a **parallel** construct.
- If none of the rules above apply to a portion of a statement in the structured block, then that portion is a unit of work.

The transformational array intrinsic functions are **MATMUL**, **DOT_PRODUCT**, **SUM**, **PRODUCT**, **MAXVAL**, **MINVAL**, **COUNT**, **ANY**, **ALL**, **SPREAD**, **PACK**, **UNPACK**, **RESHAPE**, **TRANSPOSE**, **EOSHIFT**, **CSHIFT**, **MINLOC**, and **MAXLOC**.

The units of work are assigned to the threads that execute a **workshare** region such that each unit of work is executed once.

If an array expression in the structured block references the value, association status, or allocation status of private variables, the value of the expression is undefined, unless the same value would be computed by every thread.

If an array assignment, a scalar assignment, a masked array assignment, or a **FORALL** assignment assigns to a private variable in the structured block, the result is unspecified.

The **workshare** directive causes the sharing of work to occur only in the **workshare** construct, and not in the remainder of the **workshare** region.

Execution Model Events

The *workshare-begin* event occurs after an implicit task encounters a **workshare** construct but before the task starts to execute the structured block of the **workshare** region. The *workshare-end* event occurs after an implicit task finishes execution of a **workshare** region but before it resumes execution of the enclosing context.

Tool Callbacks

A thread dispatches a registered **work** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_work_workshare** as its *work_type* argument for each occurrence of a *workshare-begin* event in that thread. Similarly, a thread dispatches a registered **work** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_work_workshare** as its *work_type* argument for each occurrence of a *workshare-end* event in that thread. The callbacks occur in the context of the implicit task.

Restrictions

Restrictions to the **workshare** construct are as follows:

- The only OpenMP constructs that may be closely nested constructs of a **workshare** construct are the **atomic**, **critical**, and **parallel** constructs.
- Base language statements that are encountered inside a **workshare** construct but that are not enclosed within a **parallel** or **atomic** construct that is nested inside the **workshare** construct must consist of only the following:
  - array assignments;
  - scalar assignments;
  - **FORALL** statements;
  - **FORALL** constructs;
  - **WHERE** statements;
  - **WHERE** constructs; and
  - **BLOCK** constructs that are strictly structured blocks associated with directives.
- All array assignments, scalar assignments, and masked array assignments that are encountered inside a **workshare** construct but are not nested inside a **parallel** construct that is nested inside the **workshare** construct must be intrinsic assignments.
- The construct must not contain any user-defined function calls unless either the function is pure and elemental or the function call is contained inside a **parallel** construct that is nested inside the **workshare** construct.

Cross References

- **atomic** Construct, see Section 17.8.5
- **critical** Construct, see Section 17.2
- **nowait** Clause, see Section 17.6
- **parallel** Construct, see Section 12.1
- OMPT **scope_endpoint** Type, see Section 33.27

- **work** Callback, see Section 34.4.1
- OMPT **work** Type, see Section 33.41

Fortran

Fortran

13.5 workdistribute Construct

Name: **workdistribute**	Association: block
Category: executable	Properties: work-distribution, partitioned

Binding

The binding region is the innermost enclosing **teams** region. The binding thread set is the set of initial threads executing the enclosing **teams** region.

Semantics

The **workdistribute** construct divides the execution of the associated structured block into separate units of work and causes the threads of the binding thread set to share the work such that each unit of work is executed only once by one thread, in the context of its implicit task. No implicit barrier occurs at the end of a **workdistribute** region.

An implementation must enforce ordering of statements that is required to maintain Fortran semantics. For example, the effects of each statement within the structured block must appear to occur before the execution of the subsequent statements, and the evaluation of the right hand side of an assignment must appear to complete prior to the effects of assigning to the left hand side.

The statements in the **workdistribute** construct are divided into units of work as follows:

- For array expressions within each statement, including transformational array intrinsic functions that compute scalar values from arrays:
  - Evaluation of each element of the array expression, including any references to pure elemental procedures, is a unit of work.
  - Evaluation of transformational array intrinsic functions may be subdivided into any number of units of work.
- For array assignment statements, assignment of each element is a unit of work.
- For scalar assignment statements, each assignment operation is a unit of work.

The transformational array intrinsic functions are **MATMUL**, **DOT_PRODUCT**, **SUM**, **PRODUCT**, **MAXVAL**, **MINVAL**, **COUNT**, **ANY**, **ALL**, **SPREAD**, **PACK**, **UNPACK**, **RESHAPE**, **TRANSPOSE**, **EOSHIFT**, **CSHIFT**, **MINLOC**, and **MAXLOC**.

The units of work are assigned to the binding thread set that execute a **workdistribute** region such that each unit of work is executed once.

If an array expression in the structured block references the value, association status, or allocation status of private variables, the value of the expression is undefined, unless the same value would be computed by every thread.

Execution Model Events

The *workdistribute-begin* event occurs after an initial task encounters a **workdistribute** construct but before the task starts to execute the structured block of the **workdistribute** region. The *workdistribute-end* event occurs after an initial task finishes execution of a **workdistribute** region but before it resumes execution of the enclosing context.

Tool Callbacks

A thread dispatches a registered **work** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_work_workdistribute** as its *work_type* argument for each occurrence of a *workdistribute-begin* event in that thread. Similarly, a thread dispatches a registered **work** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_work_workdistribute** as its *work_type* argument for each occurrence of a *workdistribute-end* event in that thread. The callbacks occur in the context of the implicit task.

Restrictions

Restrictions to the **workdistribute** construct are as follows:

- The **workdistribute** construct must be a closely nested construct inside a **teams** construct.
- No explicit region may be nested inside a **workdistribute** region.
- Base language statements that are encountered inside a **workdistribute** must consist of only the following:
  - array assignments;
  - scalar assignments; and
  - calls to pure and elemental procedures.
- All array assignments and scalar assignments that are encountered inside a **workdistribute** construct must be intrinsic assignments.
- The construct must not contain any calls to procedures that are not pure and elemental.
- If a threadprivate variable or groupprivate variable is referenced inside a **workdistribute** region, the behavior is unspecified.

Cross References

- OMPT **scope_endpoint** Type, see Section 33.27

- **target** Construct, see Section 15.8
- **teams** Construct, see Section 12.2
- **work** Callback, see Section 34.4.1
- OMPT **work** Type, see Section 33.41

Fortran

13.6 Worksharing-Loop Constructs

Binding

The binding thread set for a worksharing-loop region is the current team. A worksharing-loop region binds to the innermost enclosing parallel region. Only those threads participate in execution of the collapsed iterations and the implied barrier of the worksharing-loop region when that barrier is not eliminated by a **nowait** clause.

Semantics

The worksharing-loop construct is a worksharing construct that specifies that the collapsed iterations will be executed in parallel by threads in the team in the context of their implicit tasks. The collapsed iterations are distributed across the assigned threads of the team that is executing the parallel region to which the worksharing-loop region binds. Each thread executes its assigned chunks in the context of its implicit task. The execution of the collapsed iterations of a given chunk is consistent with their sequential order.

At the beginning of each collapsed iteration, the loop iteration variable or the variable declared by *range-decl* of each collapsed loop has the value that it would have if the collapsed loops were executed sequentially.

The loop schedule is reproducible if one of the following conditions is true:

- The **order** clause is specified with the **reproducible** *order-modifier* modifier; or
- The **schedule** clause is specified with **static** as the *kind* argument but not with the **simd** *ordering-modifier* and the **order** clause is not specified with the **unconstrained** *order-modifier*.

Execution Model Events

The *ws-loop-begin* event occurs after an implicit task encounters a worksharing-loop construct but before the task starts execution of the structured block of the worksharing-loop region. The *ws-loop-end* event occurs after a worksharing-loop region finishes execution but before resuming execution of the encountering task.

The *ws-loop-iteration-begin* event occurs at the beginning of each collapsed iteration of a worksharing-loop region. The *ws-loop-chunk-begin* event occurs for each scheduled chunk of a worksharing-loop region before the implicit task executes any of the collapsed iterations.

Tool Callbacks

A thread dispatches a registered **work** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *ws-loop-begin* event in that thread. Similarly, a thread dispatches a registered **work** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of a *ws-loop-end* event in that thread. The callbacks occur in the context of the implicit task. The *work_type* argument indicates the schedule type as shown in Table 13.1.

A thread dispatches a registered **dispatch** callback for each occurrence of a *ws-loop-iteration-begin* or *ws-loop-chunk-begin* event in that thread. The callback occurs in the context of the implicit task.

TABLE 13.1: **work** OMPT types for Worksharing-Loop

Value of *work_type*	If determined schedule is
ompt_work_loop	unknown at runtime
ompt_work_loop_static	**static**
ompt_work_loop_dynamic	**dynamic**
ompt_work_loop_guided	**guided**
ompt_work_loop_other	implementation defined

Restrictions

Restrictions to the worksharing-loop construct are as follows:

- The collapsed iteration space must be the same for all threads in the team.
- The value of the *run-sched-var* ICV must be the same for all threads in the team.

Cross References

- **dispatch** Callback, see Section 34.4.2
- *run-sched-var* ICV, see Table 3.1
- **nowait** Clause, see Section 17.6
- **order** Clause, see Section 12.3
- **schedule** Clause, see Section 13.6.3
- OMPT **scope_endpoint** Type, see Section 33.27
- **work** Callback, see Section 34.4.1
- OMPT **work** Type, see Section 33.41

C / C++

13.6.1 for Construct

Name: **for**	Association: loop nest
Category: executable	**Properties:** work-distribution, team-executed, partitioned, SIMD-partitionable, worksharing, worksharing-loop, cancellable, context-matching

Separating directives

scan

Clauses

allocate, **collapse**, **firstprivate**, **induction**, **lastprivate**, **linear**, **nowait**, **order**, **ordered**, **private**, **reduction**, **schedule**

Semantics

The **for** construct is a worksharing-loop construct.

Cross References

- **allocate** Clause, see Section 8.6
- **collapse** Clause, see Section 6.4.5
- **firstprivate** Clause, see Section 7.5.4
- Worksharing-Loop Constructs, see Section 13.6
- **induction** Clause, see Section 7.6.13
- **lastprivate** Clause, see Section 7.5.5
- **linear** Clause, see Section 7.5.6
- **nowait** Clause, see Section 17.6
- **order** Clause, see Section 12.3
- **ordered** Clause, see Section 6.4.6
- **private** Clause, see Section 7.5.3
- **reduction** Clause, see Section 7.6.10
- **scan** Directive, see Section 7.7
- **schedule** Clause, see Section 13.6.3

C / C++

Fortran

13.6.2 do Construct

Name: **do** **Category:** executable	**Association:** loop nest **Properties:** work-distribution, team-executed, partitioned, SIMD-partitionable, worksharing, worksharing-loop, cancellable, context-matching

Separating directives

scan

Clauses

allocate, **collapse**, **firstprivate**, **induction**, **lastprivate**, **linear**, **nowait**, **order**, **ordered**, **private**, **reduction**, **schedule**

Semantics

The **do** construct is a worksharing-loop construct.

Cross References

- **allocate** Clause, see Section 8.6
- **collapse** Clause, see Section 6.4.5
- **firstprivate** Clause, see Section 7.5.4
- Worksharing-Loop Constructs, see Section 13.6
- **induction** Clause, see Section 7.6.13
- **lastprivate** Clause, see Section 7.5.5
- **linear** Clause, see Section 7.5.6
- **nowait** Clause, see Section 17.6
- **order** Clause, see Section 12.3
- **ordered** Clause, see Section 6.4.6
- **private** Clause, see Section 7.5.3
- **reduction** Clause, see Section 7.6.10
- **scan** Directive, see Section 7.7
- **schedule** Clause, see Section 13.6.3

Fortran

13.6.3 `schedule` Clause

Name: `schedule`	**Properties:** schedule-specification, unique

Arguments

Name	Type	Properties
kind	Keyword: **`auto`**, **`dynamic`**, **`guided`**, **`runtime`**, **`static`**	*default*
chunk_size	expression of integer type	ultimate, optional, positive, region-invariant

Modifiers

Name	Modifies	Type	Properties
ordering-modifier	*kind*	Keyword: **`monotonic`**, **`nonmonotonic`**	unique
chunk-modifier	*kind*	Keyword: **`simd`**	unique
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`do`, **`for`**

Semantics

The **`schedule`** clause specifies how collapsed iterations of a worksharing-loop construct are divided into chunks, and how these chunks are distributed among threads of the team.

The *chunk_size* expression is evaluated using the original list items of any variables that are made private variables in the worksharing-loop construct. Whether, in what order, or how many times, any side effects of the evaluation of this expression occur is unspecified. The use of a variable in a **`schedule`** clause expression of a worksharing-loop construct causes an implicit reference to the variable in all enclosing constructs.

If the *kind* argument is **`static`**, chunks of increasing collapsed iteration numbers are assigned to the threads of the team in a round-robin fashion in the order of the thread number. Each chunk includes *chunk_size* collapsed iterations, except possibly for the chunk that contains the sequentially last iteration, which may have fewer iterations. If *chunk_size* is not specified, the collapsed iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread.

If the *kind* argument is **`dynamic`**, each thread executes a chunk, then requests another chunk, until no chunks remain to be assigned. Each chunk contains *chunk_size* collapsed iterations, except for the chunk that contains the sequentially last iteration, which may have fewer iterations. If *chunk_size* is not specified, it defaults to 1.

If the *kind* argument is **`guided`**, each thread executes a chunk, then requests another chunk, until no chunks remain to be assigned. For a *chunk_size* of 1, the size of each chunk is proportional to the number of unassigned collapsed iterations divided by the number of threads in the team, decreasing to 1. For a *chunk_size* with value $k > 1$, the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer than k collapsed iterations (except for the chunk that contains the sequentially last iteration, which may have fewer than k iterations). If *chunk_size* is not specified, it defaults to 1.

If the *kind* argument is **auto**, the decision regarding scheduling is implementation defined. If the **schedule** clause is not specified on a worksharing-loop construct then the effect is as if the **schedule** clause was specified with **auto** as its *kind* argument.

If the *kind* argument is **runtime**, the decision regarding scheduling is deferred until runtime, and the behavior is as if the clause specifies *kind*, *chunk-size* and *ordering-modifier* as set in the *run-sched-var* ICV. If the **schedule** clause explicitly specifies any modifiers then they override any corresponding modifiers that are specified in the *run-sched-var* ICV.

If the **simd** *chunk-modifier* is specified and the canonical loop nest is associated with a SIMD construct, $new\_chunk\_size = \lceil chunk\_size/simd\_width \rceil * simd\_width$ is the *chunk_size* for all chunks except the first and last chunks, where *simd_width* is an implementation defined value. The first chunk will have at least *new_chunk_size* collapsed iterations except if it is also the last chunk. The last chunk may have fewer collapsed iterations than *new_chunk_size*. If the **simd** *chunk-modifier* is specified and the canonical loop nest is not associated with a SIMD construct, the modifier is ignored.

Note – For a team of p threads and collapsed loops of n collapsed iterations, let $\lceil n/p \rceil$ be the integer q that satisfies $n = p * q - r$, with $0 <= r < p$. One compliant implementation of the **static** schedule type (with no specified *chunk_size*) would behave as though *chunk_size* had been specified with value q. Another compliant implementation would assign q collapsed iterations to the first $p - r$ threads, and $q - 1$ collapsed iterations to the remaining r threads. This illustrates why a conforming program must not rely on the details of a particular implementation.

A compliant implementation of the **guided** schedule type with a *chunk_size* value of k would assign $q = \lceil n/p \rceil$ collapsed iterations to the first available thread and set n to the larger of $n - q$ and $p * k$. It would then repeat this process until q is greater than or equal to the number of remaining collapsed iterations, at which time the remaining iterations form the final chunk. Another compliant implementation could use the same method, except with $q = \lceil n/(2p) \rceil$, and set n to the larger of $n - q$ and $2 * p * k$.

If the **monotonic** *ordering-modifier* is specified then each thread executes the chunks that it is assigned in increasing collapsed iteration order. When the **nonmonotonic** *ordering-modifier* is specified then chunks may be assigned to threads in any order and the behavior of an application that depends on any execution order of the chunks is unspecified. If an *ordering-modifier* is not specified, the effect is as if the **monotonic** *ordering-modifier* is specified if the *kind* argument is **static** or an **ordered** clause is specified on the construct; otherwise, the effect is as if the **nonmonotonic** *ordering-modifier* is specified.

Restrictions

Restrictions to the **schedule** clause are as follows:

- The **schedule** clause cannot be specified if any of the collapsed loops is a non-rectangular loop.
- The value of the *chunk_size* expression must be the same for all threads in the team.
- If **runtime** or **auto** is specified for *kind*, *chunk_size* must not be specified.
- The **nonmonotonic** *ordering-modifier* cannot be specified if an **ordered** clause is specified on the same construct.

Cross References

- **do** Construct, see Section 13.6.2
- **for** Construct, see Section 13.6.1
- *run-sched-var* ICV, see Table 3.1
- **ordered** Clause, see Section 6.4.6

13.7 `distribute` Construct

Name: distribute **Category:** executable	**Association:** loop nest **Properties:** SIMD-partitionable, teams-nestable, work-distribution, partitioned

Clauses

allocate, **collapse**, **dist_schedule**, **firstprivate**, **induction**, **lastprivate**, **order**, **private**

Binding

The binding thread set for a **distribute** region is the set of initial threads executing an enclosing **teams** region. A **distribute** region binds to this **teams** region.

Semantics

The **distribute** construct specifies that the collapsed iterations will be executed by the initial teams in the context of their implicit tasks. The collapsed iterations are distributed across the initial threads of all initial teams that execute the **teams** region to which the **distribute** region binds. No implicit barrier occurs at the end of a **distribute** region. To avoid data races the original list items that are modified due to **lastprivate** clauses should not be accessed between the end of the **distribute** construct and the end of the **teams** region to which the **distribute** binds.

If the **dist_schedule** clause is not specified, the loop schedule is implementation defined.

The schedule is reproducible if one of the following conditions is true:

- The **order** clause is specified with the **reproducible** *order-modifier* modifier; or
- The **dist_schedule** clause is specified with **static** as the *kind* argument and the **order** clause is not specified with the **unconstrained** *order-modifier*.

Execution Model Events

The *distribute-begin* event occurs after an initial task encounters a **distribute** construct but before the task starts to execute the structured block of the **distribute** region. The *distribute-end* event occurs after an initial task finishes execution of a **distribute** region but before it resumes execution of the enclosing context.

The *distribute-chunk-begin* event occurs for each scheduled chunk of a **distribute** region before execution of any collapsed iteration.

Tool Callbacks

A thread dispatches a registered **work** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_work_distribute** as its *work_type* argument for each occurrence of a *distribute-begin* event in that thread. Similarly, a thread dispatches a registered **work** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_work_distribute** as its *work_type* argument for each occurrence of a *distribute-end* event in that thread. The callbacks occur in the context of the implicit task.

A thread dispatches a registered **dispatch** callback for each occurrence of a *distribute-chunk-begin* event in that thread. The callback occurs in the context of the initial task.

Restrictions

Restrictions to the **distribute** construct are as follows:

- The collapsed iteration space must the same for all teams in the league.
- The region that corresponds to the **distribute** construct must be a strictly nested region of a **teams** region.
- A list item may appear in a **firstprivate** or **lastprivate** clause, but not in both.
- The **conditional** *lastprivate-modifier* must not be specified.
- All list items that appear in an **induction** clause must be private variables in the enclosing context.

Cross References

- **allocate** Clause, see Section 8.6
- **collapse** Clause, see Section 6.4.5
- **dispatch** Callback, see Section 34.4.2
- **dist_schedule** Clause, see Section 13.7.1
- **firstprivate** Clause, see Section 7.5.4

- Consistent Loop Schedules, see Section 6.4.4
- **induction** Clause, see Section 7.6.13
- **lastprivate** Clause, see Section 7.5.5
- **order** Clause, see Section 12.3
- **private** Clause, see Section 7.5.3
- OMPT **scope_endpoint** Type, see Section 33.27
- **teams** Construct, see Section 12.2
- **work** Callback, see Section 34.4.1
- OMPT **work** Type, see Section 33.41

13.7.1 dist_schedule Clause

Name: **dist_schedule**	**Properties:** schedule-specification, unique

Arguments

Name	Type	Properties
kind	Keyword: **static**	*default*
chunk_size	expression of integer type	ultimate, optional, positive, region-invariant

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

distribute

Semantics

The **dist_schedule** clause specifies how collapsed iterations of a **distribute** construct are divided into chunks, and how these chunks are distributed among the teams of the league. If *chunk_size* is not specified, the collapsed iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each initial team of the league. If the *chunk_size* argument is specified, collapsed iterations are divided into chunks of *chunk_size* iterations. The *chunk_size* expression is evaluated using the original list items of any variables that become private variables in the **distribute** construct. Whether, in what order, or how many times, any side effects of the evaluation of this expression occur is unspecified. The use of a variable in a **dist_schedule** clause expression of a **distribute** construct causes an implicit reference to the variable in all enclosing constructs. These chunks are assigned to the initial teams of the league in a round-robin fashion in the order of their team number.

Restrictions

Restrictions to the **dist_schedule** clause are as follows:

- The value of the *chunk_size* expression must be the same for all teams in the league.
- The **dist_schedule** clause cannot be specified if any of the collapsed loops is a non-rectangular loop.

Cross References

- **distribute** Construct, see Section 13.7

13.8 loop Construct

Name: loop Category: executable	Association: loop nest **Properties:** order-concurrent-nestable, partitioned, simdizable, team-executed, teams-nestable, work-distribution, worksharing

Clauses

bind, **collapse**, **lastprivate**, **order**, **private**, **reduction**

Binding

The **bind** clause determines the binding region, which determines the binding thread set.

Semantics

A **loop** construct specifies that the collapsed iterations execute in the context of the binding thread set, in an order specified by the **order** clause. If the **order** clause is not specified, the behavior is as if the **order** clause is present and specifies the **concurrent** *ordering*. The collapsed iterations are executed as if by the binding thread set, once per instance of the **loop** region that is encountered by the binding thread set.

The loop schedule for a **loop** construct is reproducible unless the **order** clause is present with the **unconstrained** *order-modifier*.

If the **loop** region binds to a **teams** region, the threads in the binding thread set may continue execution after the **loop** region without waiting for all collapsed iterations to complete. The collapsed iterations are guaranteed to complete before the end of the **teams** region. If the **loop** region does not bind to a **teams** region, all collapsed iterations must complete before the encountering threads continue execution after the **loop** region.

While a **loop** construct is always a work-distribution construct, it is a worksharing construct if and only if its binding region is the innermost enclosing parallel region. Further, the **loop** construct has the SIMDizable property if and only if its binding region is not defined.

Fortran

The collapsed loop may be a **DO CONCURRENT** loop.

Fortran

Restrictions

Restrictions to the **loop** construct are as follows:

- A list item must not appear in a **lastprivate** clause unless it is the loop-iteration variable of an affected loop.
- If a *reduction-modifier* is specified in a **reduction** clause that appears on the directive then the *reduction-modifier* must be **default**.
- If a **loop** construct is not nested inside another construct then the **bind** clause must be present.
- If a **loop** region binds to a **teams** region or parallel region, it must be encountered by all threads in the binding thread set or by none of them.

Fortran

- If the collapsed loop is a **DO CONCURRENT** loop, neither the data-sharing attribute clauses nor the **collapse** clause may be specified.
- If a variable is accessed in more than one iteration of a **DO CONCURRENT** loop that is associated with a **loop** construct and at least one of the accesses modifies the variable, the variable must have locality specified in the **DO CONCURRENT** loop.

Fortran

Cross References

- **bind** Clause, see Section 13.8.1
- **collapse** Clause, see Section 6.4.5
- Consistent Loop Schedules, see Section 6.4.4
- **lastprivate** Clause, see Section 7.5.5
- **order** Clause, see Section 12.3
- **private** Clause, see Section 7.5.3
- **reduction** Clause, see Section 7.6.10
- **teams** Construct, see Section 12.2

13.8.1 bind Clause

Name: bind	**Properties:** unique

Arguments

Name	Type	Properties
binding	Keyword: **parallel**, **teams**, **thread**	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

loop

Semantics

The **bind** clause specifies the binding region of the construct on which it appears. Specifically, if *binding* is **teams** and an innermost enclosing **teams** region exists then the binding region is that **teams** region; if *binding* is **parallel** then the binding region is the innermost enclosing parallel region, which may be an implicit parallel region; and if *binding* is **thread** then the binding region is not defined. If the **bind** clause is not specified on a construct for which it may be specified and the construct is a closely nested construct of a **teams** or **parallel** construct, the effect is as if *binding* is **teams** or **parallel**. If none of those conditions hold, the binding region is not defined.

The specified binding region determines the binding thread set. Specifically, if the binding region is a **teams** region, then the binding thread set is the set of initial threads that are executing that region while if the binding region is a parallel region, then the binding thread set is the team of threads that are executing that region. If the binding region is not defined, then the binding thread set is the encountering thread.

Restrictions

Restrictions to the **bind** clause are as follows:

- If **teams** is specified as *binding* then the corresponding **loop** region must be a strictly nested region of a **teams** region.
- If **teams** is specified as *binding* and the corresponding **loop** region executes on a non-host device then the behavior of a **reduction** clause that appears on the corresponding **loop** construct is unspecified if the construct is not nested inside a **teams** construct.
- If **parallel** is specified as *binding*, the behavior is unspecified if the corresponding **loop** region is a closely nested region of a **simd** region.

Cross References

- **loop** Construct, see Section 13.8

14 Tasking Constructs

This chapter defines directives and concepts related to explicit tasks.

14.1 `task` Construct

Name: `task` **Category:** executable	**Association:** block **Properties:** parallelism-generating, thread-limiting, task-generating

Clauses

`affinity`, **`allocate`**, **`default`**, **`depend`**, **`detach`**, **`final`**, **`firstprivate`**, **`if`**, **`in_reduction`**, **`mergeable`**, **`priority`**, **`private`**, **`replayable`**, **`shared`**, **`threadset`**, **`transparent`**, **`untied`**

Clause set

Properties: exclusive	**Members:** **`detach`**, **`mergeable`**

Binding

The binding thread set of the **`task`** region is the set of threads specified in the **`threadset`** clause. A **`task`** region binds to the innermost enclosing parallel region.

Semantics

When a thread encounters a **`task`** construct, an explicit task is generated from the code for the associated structured block. The data environment of the task is created according to the data-sharing attribute clauses on the **`task`** construct, per-data environment ICVs, and any defaults that apply. The data environment of the task is destroyed when the execution code of the associated structured block is completed.

The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread of the current binding thread set may be assigned the task. Task completion of the task can be guaranteed using task synchronization constructs and clauses. If a **`task`** construct is encountered during execution of an outer task, the generated task region that corresponds to this construct is not a part of the outer task region unless the generated task is an included task.

A detachable task is completed when the execution of its associated structured block is completed and the *allow-completion* event is fulfilled. If no **`detach`** clause is present on a **`task`** construct, the generated task is completed when the execution of its associated structured block is completed.

A thread that encounters a task scheduling point within the **`task`** region may temporarily suspend the **`task`** region.

The **`task`** construct includes a task scheduling point in the task region of its generating task, immediately following the generation of the explicit task. Each explicit task region includes a task scheduling point at the end of its associated structured block.

When storage is shared by an explicit task region, the programmer must ensure, by adding proper synchronization, that the storage does not reach the end of its lifetime before the explicit task region completes its execution.

When an **`if`** clause is present on a **`task`** construct and the **`if`** clause expression evaluates to *false*, an undeferred task is generated, and the encountering thread must suspend the current task region, for which execution cannot be resumed until execution of the structured block that is associated with the generated task is completed. The use of a variable in an **`if`** clause expression of a **`task`** construct causes an implicit reference to the variable in all enclosing constructs. The **`if`** clause expression is evaluated in the context outside of the **`task`** construct.

Execution Model Events

The *task-create* event occurs when a thread encounters a task-generating construct. The event occurs after the task is initialized but before its execution begins and before the encountering thread resumes execution of any task.

Tool Callbacks

A thread dispatches a registered **`task_create`** callback for each occurrence of a *task-create* event in the context of the encountering task. The *flags* argument of this callback indicates the task types shown in Table 14.1.

TABLE 14.1: **`task_create`** Callback Flags Evaluation

Operation	Evaluates to *true*
(*flags* & **`ompt_task_explicit`**)	Always in the dispatched callback
(*flags* & **`ompt_task_importing`**)	If the task is an importing task
(*flags* & **`ompt_task_exporting`**)	If the task is an exporting task
(*flags* & **`ompt_task_undeferred`**)	If the task is an undeferred task
(*flags* & **`ompt_task_final`**)	If the task is a final task
(*flags* & **`ompt_task_untied`**)	If the task is an untied task
(*flags* & **`ompt_task_mergeable`**)	If the task is a mergeable task

table continued on next page

table continued from previous page

Operation	Evaluates to ***true***
(*flags* & **ompt_task_merged**)	If the task is a merged task

Cross References

- **affinity** Clause, see Section 14.10
- **allocate** Clause, see Section 8.6
- **default** Clause, see Section 7.5.1
- **depend** Clause, see Section 17.9.5
- **detach** Clause, see Section 14.11
- **final** Clause, see Section 14.7
- **firstprivate** Clause, see Section 7.5.4
- Task Scheduling, see Section 14.14
- **if** Clause, see Section 5.5
- **in_reduction** Clause, see Section 7.6.12
- **mergeable** Clause, see Section 14.5
- **omp_fulfill_event** Routine, see Section 23.2.1
- **priority** Clause, see Section 14.9
- **private** Clause, see Section 7.5.3
- **replayable** Clause, see Section 14.6
- **shared** Clause, see Section 7.5.2
- **task_create** Callback, see Section 34.5.1
- OMPT **task_flag** Type, see Section 33.37
- **threadset** Clause, see Section 14.8
- **transparent** Clause, see Section 17.9.6
- **untied** Clause, see Section 14.4

14.2 taskloop Construct

Name: **taskloop**	**Association:** loop nest
Category: executable	**Properties:** parallelism-generating, SIMD-partitionable, task-generating

Subsidiary directives

task_iteration

Clauses

allocate, **collapse**, **default**, **final**, **firstprivate**, **grainsize**, **if**, **in_reduction**, **induction**, **lastprivate**, **mergeable**, **nogroup**, **num_tasks**, **priority**, **private**, **reduction**, **replayable**, **shared**, **threadset**, **transparent**, **untied**

Clause set

synchronization-clause

Properties: exclusive	**Members:** **nogroup**, **reduction**

Clause set

granularity-clause

Properties: exclusive	**Members:** **grainsize**, **num_tasks**

Binding

The binding thread set of the **taskloop** region is the set of threads specified in the **threadset** clause. A **taskloop** region binds to the innermost enclosing parallel region.

Semantics

When a thread encounters a **taskloop** construct, the construct partitions the collapsed iterations into chunks, each of which is assigned to an explicit task for parallel execution. The data environment of each generated task is created according to the data-sharing attribute clauses on the **taskloop** construct, per-data environment ICVs, and any defaults that apply. Tasks created by a **taskloop** directive can be affected by **task_iteration** directives that are subsidiary directives of that **taskloop** directive. If a **task_iteration** directive on which a **depend** clause appears is a subsidiary directive of the **taskloop** construct then the behavior is as if the order of the creation of the generated tasks is in increasing collapsed iteration order with respect to their assigned chunks. Otherwise, the order of the creation of the generated tasks is unspecified and programs that rely on the execution order of the logical iterations are non-conforming.

If the **nogroup** clause is not present, the **taskloop** construct executes as if it was enclosed in a **taskgroup** construct with no statements or directives outside of the **taskloop** construct. Thus, the **taskloop** construct creates an implicit **taskgroup** region. If the **nogroup** clause is present, no implicit **taskgroup** region is created.

If a **reduction** clause is present, the behavior is as if a **task_reduction** clause with the same reduction identifier and list items was applied to the implicit **taskgroup** construct that encloses the **taskloop** construct. The **taskloop** construct executes as if each generated task was defined by a **task** construct on which an **in_reduction** clause with the same reduction identifier and list items is present. Thus, the generated tasks are participants of the reduction defined by the **task_reduction** clause that was applied to the implicit **taskgroup** construct.

If an **in_reduction** clause is present, the behavior is as if each generated task was defined by a **task** construct on which an **in_reduction** clause with the same reduction identifier and list items is present. Thus, the generated tasks are participants of a reduction previously defined by a reduction-scoping clause.

If a **threadset** clause is present, the behavior is as if each generated task was defined by a **task** construct on which a **threadset** clause with the same set of threads is present. Thus, the binding thread set of the generated tasks is the same as that of the **taskloop** region.

If a **transparent** clause is present, the behavior is as if each generated task was defined by a **task** construct on which a **transparent** clause with the same *impex-type* argument is present.

If no clause from the *granularity-clause* clause set is present, the number of loop tasks generated and the number of logical iterations assigned to these tasks is implementation defined.

When an **if** clause is present and the **if** clause expression evaluates to *false*, undeferred tasks are generated. The use of a variable in an **if** clause expression causes an implicit reference to the variable in all enclosing constructs.

C++

For firstprivate variables of class type, the number of invocations of copy constructors that perform the initialization is implementation defined.

C++

When storage is shared by a **taskloop** region, the programmer must ensure, by adding proper synchronization, that the storage does not reach the end of its lifetime before the **taskloop** region and its descendent tasks complete their execution.

Execution Model Events

The *taskloop-begin* event occurs upon entering the **taskloop** region. A *taskloop-begin* will precede any *task-create* events for the generated tasks. The *taskloop-end* event occurs upon completion of the **taskloop** region.

Events for an implicit **taskgroup** region that surrounds the **taskloop** region are the same as for the **taskgroup** construct.

The *taskloop-iteration-begin* event occurs at the beginning of each *logical-iteration* of a **taskloop** region before an explicit task executes the logical iteration. The *taskloop-chunk-begin* event occurs before an explicit task executes any of its associated logical iterations in a **taskloop** region.

Tool Callbacks

A thread dispatches a registered **work** callback for each occurrence of a *taskloop-begin* and *taskloop-end* event in that thread. The callback occurs in the context of the encountering task. The callback receives **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate, and **ompt_work_taskloop** as its *work_type* argument.

A thread dispatches a registered **dispatch** callback for each occurrence of a *taskloop-iteration-begin* or *taskloop-chunk-begin* event in that thread. The callback binds to the explicit task executing the logical iterations.

Restrictions

Restrictions to the **taskloop** construct are as follows:

- The *reduction-modifier* must be **default**.
- The **conditional** *lastprivate-modifier* must not be specified.
- If the **taskloop** construct is associated with a **task_iteration** directive, none of the **taskloop**-affected loops may be the generated loop of a loop-transforming construct.

Cross References

- **allocate** Clause, see Section 8.6
- **collapse** Clause, see Section 6.4.5
- **default** Clause, see Section 7.5.1
- **dispatch** Callback, see Section 34.4.2
- **final** Clause, see Section 14.7
- **firstprivate** Clause, see Section 7.5.4
- Canonical Loop Nest Form, see Section 6.4.1
- **grainsize** Clause, see Section 14.2.1
- **if** Clause, see Section 5.5
- **in_reduction** Clause, see Section 7.6.12
- **induction** Clause, see Section 7.6.13
- **lastprivate** Clause, see Section 7.5.5
- **mergeable** Clause, see Section 14.5
- **nogroup** Clause, see Section 17.7
- **num_tasks** Clause, see Section 14.2.2
- **priority** Clause, see Section 14.9

- **private** Clause, see Section 7.5.3
- **reduction** Clause, see Section 7.6.10
- **replayable** Clause, see Section 14.6
- OMPT **scope_endpoint** Type, see Section 33.27
- **shared** Clause, see Section 7.5.2
- **task** Construct, see Section 14.1
- **task_iteration** Directive, see Section 14.2.3
- **taskgroup** Construct, see Section 17.4
- **threadset** Clause, see Section 14.8
- **transparent** Clause, see Section 17.9.6
- **untied** Clause, see Section 14.4
- **work** Callback, see Section 34.4.1
- OMPT **work** Type, see Section 33.41

14.2.1 grainsize Clause

Name: **grainsize**	**Properties:** taskgraph-altering, unique

Arguments

Name	Type	Properties
grain-size	expression of integer type	positive

Modifiers

Name	Modifies	Type	Properties
prescriptiveness	*grain-size*	Keyword: **strict**	unique
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

taskloop

Semantics

The **grainsize** clause specifies the number of logical iterations, L_t, that are assigned to each generated task t. If *prescriptiveness* is not specified as **strict**, other than possibly for the generated task that contains the sequentially last iteration, L_t is greater than or equal to the minimum of the value of the *grain-size* expression and the number of logical iterations, but less than two times the value of the *grain-size* expression. If *prescriptiveness* is specified as **strict**, other than possibly for the generated task that contains the sequentially last iteration, L_t is equal to the value of the *grain-size* expression. In both cases, the generated task that contains the sequentially last iteration may have fewer logical iterations than the value of the *grain-size* expression.

Restrictions

Restrictions to the **`grainsize`** clause are as follows:

- None of the collapsed loops may be non-rectangular loops.

Cross References

- **`taskloop`** Construct, see Section 14.2

14.2.2 `num_tasks` Clause

Name: **`num_tasks`**	**Properties:** taskgraph-altering, unique

Arguments

Name	Type	Properties
num-tasks	expression of integer type	positive

Modifiers

Name	Modifies	Type	Properties
prescriptiveness	*num-tasks*	Keyword: **`strict`**	unique
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`taskloop`

Semantics

The **`num_tasks`** clause specifies that the **`taskloop`** construct create as many tasks as the minimum of the *num-tasks* expression and the number of logical iterations. Each task must have at least one logical iteration. If *prescriptiveness* is specified as **`strict`** for a **`taskloop`** region with N logical iterations, the logical iterations are partitioned in a balanced manner and each partition is assigned, in order, to a generated task. The partition size is $\lceil N/\textit{num-tasks} \rceil$ until the number of remaining logical iterations divides the number of remaining tasks evenly, at which point the partition size becomes $\lfloor N/\textit{num-tasks} \rfloor$.

Restrictions

Restrictions to the **`num_tasks`** clause are as follows:

- None of the collapsed loops may be non-rectangular loops.

Cross References

- **`taskloop`** Construct, see Section 14.2

14.2.3 `task_iteration` Directive

Name: `task_iteration`	Association: unassociated
Category: subsidiary	Properties: *default*

Enclosing directives

`taskloop`

Clauses

`affinity`, **`depend`**, **`if`**

Semantics

The **`task_iteration`** directive is a subsidiary directive that controls the per-iteration task-execution attributes of the generated tasks of its associated **`taskloop`** construct, which is the innermost enclosing **`taskloop`** construct, as described below.

For each task-inherited clause specified on the **`task_iteration`** directive, the behavior is as if each task generated by the enclosing **`taskloop`** construct is specified with a corresponding clause that has the same *clause-specification*, but adjusted as follows. These clauses are instantiated for each instance of the loop-iteration variables for which the *if-expression* of the **`if`** clause evaluates to *true*. If an **`if`** clause is not specified on the **`task_iteration`** directive, the behavior is as if the *if-expression* evaluates to *true*.

Restrictions

The restrictions to the **`task_iteration`** directive are as follows:

- Each **`task_iteration`** directive must appear in the loop body of one of the **`taskloop`**-affected loops and must precede all statements and directives (except other **`task_iteration`** directives) in that loop body.
- If a **`task_iteration`** directive appears in the loop body of one of the **`taskloop`**-affected loops, no intervening code may occur between any two collapsed loops of the **`taskloop`**-affected loops.

Cross References

- **`affinity`** Clause, see Section 14.10
- **`depend`** Clause, see Section 17.9.5
- **`if`** Clause, see Section 5.5
- **`iterator`** Modifier, see Section 5.2.6
- **`task`** Construct, see Section 14.1
- **`taskloop`** Construct, see Section 14.2

14.3 taskgraph Construct

Name: **taskgraph**	Association: block
Category: executable	**Properties:** *default*

Clauses

graph_id, **graph_reset**, **if**, **nogroup**

Binding

The binding thread set of a **taskgraph** region is all threads on the current device. The binding task set of a **taskgraph** region is all tasks of the current team that are generated in the region.

Semantics

When a thread encounters a **taskgraph** construct, a **taskgraph** region is generated for which execution entails one of the following:

- Execution of the structured block associated with the construct, while optionally creating a taskgraph record of all encountered replayable constructs and the sequence in which they are encountered; or
- A replay execution of the last matching taskgraph record of the construct.

If a **taskloop** construct is encountered in the **taskgraph** region, the behavior is as if each task that it generates is instead generated by a **task** construct. If a task-generating construct is encountered in the **taskgraph** construct as part of its corresponding region, then it is a replayable construct of the region unless otherwise specified by the **replayable** clause. If a **depend** clause with a **depobj** *task-dependence-type* is present on a replayable construct then for each listed depend object the behavior is as if a **depend** clause with the dependence type and locator list item represented by the depend object is instead present on the construct. Whether a task-generating construct that is encountered as part of the **taskgraph** region, but not in the **taskgraph** construct, is a replayable construct of the region is unspecified, unless the **replayable** clause is present on that construct. For the purposes of the **taskgraph** region, a **taskwait** construct on which the **depend** clause appears is a task-generating construct.

A taskgraph record contains a record of the following:

- The *graph-id-value* specified in the **graph_id** clause upon encountering the construct;
- The sequence of encountered replayable constructs in the **taskgraph** region, along with their subsidiary directives; and
- For each replayable construct, a saved data environment.

A clause or modifier argument for a replayable construct is recorded after evaluating all expressions that compose the argument and substituting the resulting values for those expressions. Additionally, if a clause argument or a modifier argument specification requires a locator list item or a variable list item, then:

- For a locator list item of a taskgraph-altering clause, only the storage locations are recorded;

- Otherwise, the identifier that designates the base variable or base pointer of the list item is recorded along with any values that are needed to reconstruct the list item.

The saved data environment of each replayable construct in the taskgraph record includes copies of all variables that do not have static storage duration and that are firstprivate in the replayable construct, with values that are captured from the enclosing data environment when the construct is encountered. Additionally, it includes copies of all variables that have static storage duration and that appear in a **firstprivate** clause that has the *saved* modifier on the construct. Finally, it includes references to any other variables that have static storage duration, exist in the enclosing data environment of the replayable construct, and do not exist in the enclosing data environment of the **taskgraph** construct.

The taskgraph record becomes a finalized taskgraph record on exit from the **taskgraph** region in which it is created. An implementation may create a finalized taskgraph record prior to the first execution of the **taskgraph** region, if it can guarantee that the contents of the record would match the record that would have been created during an execution of the region. In this case, a replay execution of that taskgraph record may occur upon first encountering the **taskgraph** construct.

If the **graph_id** clause is not present, an existing finalized taskgraph record that was generated for the construct when encountered on the same device is the matching taskgraph record. Otherwise, an existing finalized taskgraph record that was generated for the construct when encountered on the same device is the matching taskgraph record if the *graph-id-value* specified in the **graph_id** clause matches the value in the **graph_id** clause that was saved in the record.

Each finalized taskgraph record has an associated *replay count* that is initialized to zero. If the **graph_reset** clause is not present or its argument evalutes to *false*, the encountering task of the **taskgraph** region is not a final task, and a matching taskgraph record exists, the matching taskgraph record is replayed and its replay count is incremented by one. A replay execution of a taskgraph record has the effect of encountering the recorded replayable constructs, with their recorded clause and modifier arguments unless otherwise specified, in their recorded sequence and implies all semantics defined for those constructs except as otherwise specified in this section. A replay execution does not entail execution of any code that is part of both the **taskgraph** region and the encountering task region. Any changes from when the matching taskgraph record was created to the arguments or modifiers of a taskgraph-altering clause that appears on a replayable construct does not alter the behavior of a replay execution of that taskgraph record. The replay count is decremented by one once all tasks that are generated by the replayable constructs have completed.

If completion of a **taskgraph** region results in a new finalized taskgraph record when a matching taskgraph record already exists, the behavior is as if the new record replaces the old record, with the old record being discarded once its replay count reaches zero.

When executing a replayable construct during a replay execution, unless otherwise specified by a *saved* modifier on a data-environment attribute clause, its enclosing data environment (inclusive of ICVs with data environment ICV scope) is the enclosing data environment of the **taskgraph** construct. If a variable does not exist in the enclosing data environment of the **taskgraph** construct then the saved data environment in the taskgraph record is used as the enclosing data environment for that variable. If the replayable construct permits an ICV-defaulted clause and the clause is not present, in a replay execution of the construct the ICV in the enclosing data environment of the **taskgraph** construct determines the value of the clause argument.

If the **if** clause is present and its argument evaluates to *false*, execution of the **taskgraph** region will not create a taskgraph record or entail replaying a matching taskgraph record of the construct.

If the **nogroup** clause is not present, the **taskgraph** region executes as if enclosed by a **taskgroup** region.

Whether foreign tasks are recorded or not in a taskgraph record and the manner in which they are executed during a replay execution if they are recorded is implementation defined.

Execution Model Events

Events for the implicit **taskgroup** region that surrounds the **taskgraph** region when no **nogroup** clause is specified are the same as for the **taskgroup** construct.

The events that occur during a replay execution of a **taskgraph** region is unspecified.

Tool Callbacks

Callbacks associated with events for the **taskgroup** region are the same as for the **taskgroup** construct as defined in Section 17.4.

Restrictions

Restrictions to the **taskgraph** construct are as follows:

- Task-generating constructs are the only constructs that may be encountered as part of the **taskgraph** region.
- A **taskgraph** construct must not be encountered in a final task region.
- A replayable construct that generates an importing or exporting transparent task, a detachable task, or an undeferred task must not be encountered in a **taskgraph** region.
- Any variable referenced in a replayable construct that does not have static storage duration and that does not exist in the enclosing data environment of the **taskgraph** construct must be a private-only or firstprivate variable in the replayable construct.
- A list item of a clause on a replayable construct that accepts a locator list and is not a taskgraph-altering clause must have a base variable or base pointer.
- Any variable that appears in an expression of a variable list item or locator list item for a clause on a replayable construct and does not designate the base variable or base pointer of that list item must be listed in a data-environment attribute clause with the *saved* modifier on that construct.
- If a construct that permits the **nogroup** clause is encountered in a **taskgraph** region then the **nogroup** clause must be specified with the *do_not_synchronize* argument evaluating to *true*.

Cross References

- **graph_id** Clause, see Section 14.3.1
- **graph_reset** Clause, see Section 14.3.2
- **if** Clause, see Section 5.5
- **nogroup** Clause, see Section 17.7
- **task** Construct, see Section 14.1
- **taskgroup** Construct, see Section 17.4

14.3.1 graph_id Clause

Name: **graph_id**	**Properties:** unique

Arguments

Name	Type	Properties
graph-id-value	expression of OpenMP integer type	*default*

Directives

taskgraph

Semantics

The **graph_id** clause specifies the *graph-id-value* that identifies a taskgraph record. At most, one matching taskgraph record exists for a given *graph-id-value*.

Cross References

- **taskgraph** Construct, see Section 14.3

14.3.2 graph_reset Clause

Name: **graph_reset**	**Properties:** unique

Arguments

Name	Type	Properties
graph-reset-expression	expression of OpenMP logical type	*default*

Directives

taskgraph

Semantics

If *graph-reset-expression* evaluates to *true*, any existing matching taskgraph record is discarded if a replay of the record is not in progress (i.e., if its replay count equals zero). If the replay count is non-zero, the matching taskgraph record is not replayed and instead the structured block associated with the **`taskgraph`** construct is executed; in this case, the matching taskgraph record is discarded once its replay count reaches zero. If *graph-reset-expression* is not specified, the effect is as if *graph-reset-expression* evaluates to *true*.

Cross References

- **`taskgraph`** Construct, see Section 14.3

14.4 `untied` Clause

Name: `untied`	**Properties:** unique

Arguments

Name	Type	Properties
can_change_threads	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`task`, **`taskloop`**

Semantics

If *can-change-threads* evaluates to *true*, the **`untied`** clause specifies that tasks generated by the construct on which it appears are untied tasks, which means that any thread in the binding thread set can resume the **`task`** region after a suspension. If *can-change-threads* evaluates to *false* or if the **`untied`** clause is not specified on a construct on which it may appear, generated tasks are tied; if a tied task is suspended, its **`task`** region can only be resumed by the thread that started its execution. If a generated task is a final task or an included task, the **`untied`** clause is ignored and the task is tied. If *can-change-threads* is not specified, the effect is as if *can-change-threads* evaluates to *true*.

Cross References

- **`task`** Construct, see Section 14.1
- **`taskloop`** Construct, see Section 14.2

14.5 `mergeable` Clause

Name: `mergeable`	**Properties:** unique

Arguments

Name	Type	Properties
can_merge	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`target_data`, **`task`**, **`taskloop`**

Semantics

If *can_merge* evaluates to *true*, the **`mergeable`** clause specifies that tasks generated by the construct on which it appears are mergeable tasks. If *can_merge* evaluates to *false*, the **`mergeable`** clause specifies that tasks generated by the construct on which it appears are not mergeable tasks. If *can_merge* is not specified, the effect is as if *can_merge* evaluates to *true*. If the generated task is a mergeable task that is also an undeferred task, the implementation may generate a merged task instead.

Cross References

- **`target_data`** Construct, see Section 15.7
- **`task`** Construct, see Section 14.1
- **`taskloop`** Construct, see Section 14.2

14.6 `replayable` Clause

Name: `replayable`	**Properties:** *default*

Arguments

Name	Type	Properties
replayable-expression	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

target, **target_enter_data**, **target_exit_data**, **target_update**, **task**, **taskloop**, **taskwait**

Semantics

If *replayable-expression* evaluates to *true*, the **replayable** clause specifies that the construct on which it appears is a replayable construct. If *replayable-expression* evaluates to *false*, the **replayable** clause specifies that the construct on which it appears is not a replayable construct. If *replayable-expression* is not specified, the effect is as if *replayable-expression* evaluates to *true*.

Cross References

- **target** Construct, see Section 15.8
- **target_enter_data** Construct, see Section 15.5
- **target_exit_data** Construct, see Section 15.6
- **target_update** Construct, see Section 15.9
- **task** Construct, see Section 14.1
- **taskloop** Construct, see Section 14.2
- **taskwait** Construct, see Section 17.5

14.7 final Clause

Name: final	**Properties:** unique

Arguments

Name	Type	Properties
finalize	expression of OpenMP logical type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

task, **taskloop**

Semantics

The **final** clause specifies that tasks generated by the construct on which it appears are final tasks if the *finalize* expression evaluates to *true*. All task-generating constructs on which the **final** clause may be specified that are encountered during execution of a final task generate included final tasks. The use of a variable in a *finalize* expression causes an implicit reference to the variable in all enclosing constructs. The *finalize* expression is evaluated in the context outside of the construct on which the clause appears, If *finalize* is not specified, the effect is as if *finalize* evaluates to *true*.

Cross References

- **task** Construct, see Section 14.1
- **taskloop** Construct, see Section 14.2

14.8 threadset Clause

Name: threadset	**Properties:** unique

Arguments

Name	Type	Properties
set	Keyword: **omp_pool**, **omp_team**	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

task, **taskloop**

Semantics

The **threadset** clause specifies the set of threads that may execute tasks that are generated by the construct on which it appears. If the *set* argument is **omp_team**, the generated tasks may only be scheduled onto threads of the current team. If the *set* argument is **omp_pool**, the generated tasks may be scheduled onto unassigned threads of the current OpenMP thread pool in addition to threads of the current team. If the **threadset** clause is not specified on a construct on which it may appear, then the effect is as if the **threadset** clause was specified with **omp_team** as its *set* argument. If the encountering task is a final task, the **threadset** clause is ignored.

Cross References

- **task** Construct, see Section 14.1
- **taskloop** Construct, see Section 14.2

14.9 `priority` Clause

Name: `priority`	Properties: unique

Arguments

Name	Type	Properties
priority-value	expression of integer type	constant, non-negative

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`target`, **`target_data`**, **`target_enter_data`**, **`target_exit_data`**, **`target_update`**, **`task`**, **`taskgraph`**, **`taskloop`**

Semantics

The **`priority`** clause specifies, in the *priority-value* argument, a task priority for the construct on which it appears . Among all tasks ready to be executed, higher priority tasks (those with a higher numerical *priority-value*) are recommended to execute before lower priority ones. The default *priority-value* when no **`priority`** clause is specified is zero (the lowest task priority). If a specified *priority-value* is higher than the *max-task-priority-var* ICV then the implementation will use the value of that ICV. An OpenMP program that relies on the task execution order being determined by the task priorities may have unspecified behavior.

Cross References

- *max-task-priority-var* ICV, see Table 3.1
- **`target`** Construct, see Section 15.8
- **`target_data`** Construct, see Section 15.7
- **`target_enter_data`** Construct, see Section 15.5
- **`target_exit_data`** Construct, see Section 15.6
- **`target_update`** Construct, see Section 15.9
- **`task`** Construct, see Section 14.1
- **`taskgraph`** Construct, see Section 14.3
- **`taskloop`** Construct, see Section 14.2

14.10 affinity Clause

Name: affinity	**Properties:** task-inherited

Arguments

Name	Type	Properties
locator-list	list of locator list item type	*default*

Modifiers

Name	Modifies	Type	Properties
iterator	*locator-list*	Complex, name: **iterator** Arguments: ***iterator-specifier*** list of iterator specifier list item type (*default*)	unique
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

target_data, **task**, **task_iteration**

Semantics

The **affinity** clause specifies a hint to indicate data affinity of tasks generated by the construct on which it appears. The hint recommends to execute generated tasks close to the location of the original list items. A program that relies on the task execution location being determined by this list may have unspecified behavior.

The list items that appear in the **affinity** clause may also appear in data-environment clauses. The list items may reference any *iterators-identifier* that is defined in the same clause and may include array sections.

C / C++

The list items that appear in the **affinity** clause may use shape-operators.

C / C++

Cross References

- **iterator** Modifier, see Section 5.2.6
- **target_data** Construct, see Section 15.7
- **task** Construct, see Section 14.1
- **task_iteration** Directive, see Section 14.2.3

14.11 `detach` Clause

Name: `detach`	**Properties:** data-sharing attribute, innermost-leaf, privatization, unique

Arguments

Name	Type	Properties
event-handle	variable of event_handle type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`target_data`, **`task`**

Semantics

The **`detach`** clause specifies that the task generated by the construct on which it appears is a detachable task. The clause provides a superset of the functionality provided by the **`private`** clause. A new *allow-completion* event is created and connected to the completion of the associated **`task`** region. The original *event-handle* is updated to represent that *allow-completion* event before the task data environment is created. The use of a variable in a **`detach`** clause expression of a **`task`** construct causes an implicit reference to the variable in all enclosing constructs.

Restrictions

Restrictions to the **`detach`** clause are as follows:

- If a **`detach`** clause appears on a directive, then the encountering task must not be a final task.
- A variable that appears in a **`detach`** clause cannot appear as a list item on any data-environment attribute clause on the same construct.
- A variable that is part of an aggregate variable cannot appear in a **`detach`** clause.

Fortran

- *event-handle* must not have the **`POINTER`** attribute.
- If *event-handle* has the **`ALLOCATABLE`** attribute, the allocation status must be allocated when the **`task`** construct is encountered, and the allocation status must not be changed, either explicitly or implicitly, in the **`task`** region.

Fortran

Cross References

- OpenMP **event_handle** Type, see Section 20.6.1
- **target_data** Construct, see Section 15.7
- **task** Construct, see Section 14.1

14.12 taskyield Construct

Name: **taskyield**	Association: unassociated
Category: executable	Properties: *default*

Binding

A **taskyield** region binds to the current task region. The binding thread set of the **taskyield** region is the current team.

Semantics

The **taskyield** region includes an explicit task scheduling point in the current task region.

Cross References

- Task Scheduling, see Section 14.14

14.13 Initial Task

Execution Model Events

While no events are associated with the implicit parallel region in each initial thread, several events are associated with initial tasks. The *initial-thread-begin* event occurs in an initial thread after the OpenMP runtime invokes the OMPT-tool initializer but before the initial thread begins to execute the first explicit region in the initial task. The *initial-task-begin* event occurs after an *initial-thread-begin* event but before the first explicit region in the initial task begins to execute. The *initial-task-end* event occurs before an *initial-thread-end* event but after the last region in the initial task finishes execution. The *initial-thread-end* event occurs as the final event in an initial thread at the end of an initial task immediately prior to invocation of the OMPT-tool finalizer.

Tool Callbacks

A thread dispatches a registered **thread_begin** callback for the *initial-thread-begin* event in an initial thread. The callback occurs in the context of the initial thread. The callback receives **ompt_thread_initial** as its *thread_type* argument.

A thread dispatches a registered **implicit_task** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of an *initial-task-begin* event in that thread. Similarly, a thread dispatches a registered **implicit_task** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of an *initial-task-end* event in that thread. The callbacks occur in the context of the initial task. In the dispatched callback, (*flags* **&** **ompt_task_initial)** and (*flags* **&** **ompt_task_implicit)** evaluate to *true*.

A thread dispatches a registered **thread_end** callback for the *initial-thread-end* event in that thread. The callback occurs in the context of the thread. The implicit parallel region does not dispatch a **parallel_end** callback; however, the implicit parallel region can be finalized within this **thread_end** callback.

Cross References

- **implicit_task** Callback, see Section 34.5.3
- **parallel_end** Callback, see Section 34.3.2
- OMPT **scope_endpoint** Type, see Section 33.27
- OMPT **task_flag** Type, see Section 33.37
- OMPT **thread** Type, see Section 33.39
- **thread_begin** Callback, see Section 34.1.3
- **thread_end** Callback, see Section 34.1.4

14.14 Task Scheduling

Whenever a thread reaches a task scheduling point, it may begin or resume execution of a task from its schedulable task set. An idle thread is treated as if it is always at a task scheduling point. For other threads, task scheduling points are implied at the following locations:

- During the generation of an explicit task;
- The point immediately following the generation of an explicit task;
- After the point of completion of the structured block associated with a task;
- In a **taskyield** region;
- In a **taskwait** region;
- At the end of a **taskgroup** region;
- At the beginning and end of a **taskgraph** region;
- In an implicit barrier region;
- In an explicit **barrier** region;
- During the generation of a **target** region;
- The point immediately following the generation of a **target** region;
- In a **target_update** region;

- In a **`target_enter_data`** region;
- In a **`target_exit_data`** region;
- In each instance of any memory-copying routine;and
- In each instance of any memory-setting routine.

When a thread encounters a task scheduling point it may do one of the following, subject to the task scheduling constraints specified below:

- Begin execution of a tied task in its schedulable task set;
- Resume the suspended task region of any task to which it is tied;
- Begin execution of an untied task in its schedulable task set; or
- Resume the suspended task region of any untied task in its schedulable task set.

If more than one of the above choices is available, which one is chosen is unspecified.

Task Scheduling Constraints are as follows:

1. If any suspended tasks are tied to the thread and are not suspended in a barrier region, a new explicit tied task may be scheduled only if it is a descendent task of all of those suspended tasks. Otherwise, any new explicit tied task may be scheduled.
2. A dependent task shall not start its execution until its task dependences are fulfilled.
3. A task shall not be scheduled while another task has been scheduled but has not yet completed, if they are mutually exclusive tasks.
4. A task shall not start or resume execution on an unassigned thread if it would result in the total number of free-agent threads in the OpenMP thread pool exceeding *free-agent-thread-limit-var*.

Task scheduling points dynamically divide task regions into subtasks. Each subtask is executed uninterrupted from start to end. Different subtasks of the same task region are executed in the order in which they are encountered. In the absence of task synchronization constructs, the order in which a thread executes subtasks of different tasks in its schedulable task set is unspecified.

A program must behave correctly and consistently with all conceivable scheduling sequences that are compatible with the rules above. A program that relies on any other assumption about task scheduling is a non-conforming program.

Note – For example, if threadprivate memory is accessed (explicitly in the source code or implicitly in calls to library procedures) in one subtask of a task region, its value cannot be assumed to be preserved into the next subtask of the same task region if another schedulable task exists that modifies it.

As another example, if different subtasks of a task region invoke a lock-acquiring routine and its corresponding lock-releasing routine, no invocation of a lock-acquiring routine for the same lock should be made in any subtask of another task that the executing thread may schedule. Otherwise, deadlock is possible. A similar situation can occur when a **critical** region spans multiple subtasks of a task and another schedulable task contains a **critical** region with the same name.

Execution Model Events

The *task-schedule* event occurs in a thread when the thread switches tasks at a task scheduling point; no event occurs when switching to or from a merged task.

Tool Callbacks

A thread dispatches a registered **task_schedule** callback for each occurrence of a *task-schedule* event in the context of the task that begins or resumes. The *prior_task_status* argument is used to indicate the cause for suspending the prior task. This cause may be the completion of the prior task region, the encountering of a **taskyield** construct, or the encountering of an active cancellation point.

Cross References

- **task_schedule** Callback, see Section 34.5.2

15 Device Directives and Clauses

This chapter defines constructs and concepts related to device execution.

15.1 `device_type` Clause

Name: `device_type`	**Properties:** unique

Arguments

Name	Type	Properties
device-type-description	Keyword: **`any`**, **`host`**, **`nohost`**	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`begin declare_target`, **`declare_target`**, **`groupprivate`**, **`target`**

Semantics

If the **`device_type`** clause appears on a declarative directive, the *device-type-description* argument specifies the type of devices for which a version of the procedure or variable should be made available. If the **`device_type`** clause appears on a **`target`** construct, the argument specifies the type of devices for which the implementation should support execution of the corresponding **`target`** region.

The **`host`** *device-type-description* specifies the host device. The **`nohost`** *device-type-description* specifies any supported non-host device. The **`any`** *device-type-description* specifies any supported device. If the **`device_type`** clause is not specified, the behavior is as if the **`device_type`** clause appears with **`any`** specified.

If the **`device_type`** clause specifies the host device on a **`target`** construct for which the target device is a non-host device, the corresponding region executes on the host device. Otherwise, if the devices specified by the **`device_type`** clause does not include the target device then runtime error termination is performed.

Cross References

- **begin declare_target** Directive, see Section 9.9.2
- **declare_target** Directive, see Section 9.9.1
- **groupprivate** Directive, see Section 7.13
- **target** Construct, see Section 15.8

15.2 device Clause

Name: device	**Properties:** ICV-defaulted, unique

Arguments

Name	Type	Properties
device-description	expression of integer type	*default*

Modifiers

Name	Modifies	Type	Properties
device-modifier	*device-description*	Keyword: **ancestor**, **device_num**	*default*
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

dispatch, **interop**, **target**, **target_data**, **target_enter_data**, **target_exit_data**, **target_update**

Semantics

The **device** clause identifies the target device that is associated with a device construct.

If **device_num** is specified as the *device-modifier*, the *device-description* specifies the device number of the target device. If *device-modifier* does not appear in the clause, the behavior of the clause is as if *device-modifier* is **device_num**. If the *device-description* evaluates to **omp_invalid_device**, runtime error termination is performed.

If **ancestor** is specified as the *device-modifier*, the *device-description* specifies the number of target nesting levels of the target device. Specifically, if the *device-description* evaluates to 1, the target device is the parent device of the enclosing **target** region. If the construct on which the **device** clause appears is not encountered in a **target** region, the current device is treated as the parent device.

Unless otherwise specified, for directives that accept the **device** clause, if no **device** clause is present, the behavior is as if the **device** clause appears with **device_num** as *device-modifier* and with a *device-description* that evaluates to the value of the *default-device-var* ICV.

Restrictions

- The **ancestor** *device-modifier* must not appear on the **device** clause on any directive other than the **target** construct.
- If the **ancestor** *device-modifier* is specified, the *device-description* must evaluate to 1 and a **requires** directive with the **reverse_offload** clause must be specified;
- If the **device_num** *device-modifier* is specified and *target-offload-var* is not **mandatory**, *device-description* must evaluate to a conforming device number.

Cross References

- **dispatch** Construct, see Section 9.7
- *target-offload-var* ICV, see Table 3.1
- **interop** Construct, see Section 16.1
- **target** Construct, see Section 15.8
- **target_data** Construct, see Section 15.7
- **target_enter_data** Construct, see Section 15.5
- **target_exit_data** Construct, see Section 15.6
- **target_update** Construct, see Section 15.9

15.3 thread_limit Clause

Name: thread_limit	Properties: ICV-modifying, target-consistent, unique

Arguments

Name	Type	Properties
threadlim	expression of integer type	positive

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

target, **teams**

Semantics

As described in Section 3.4, some constructs limit the number of threads that may participate in the parallel execution of tasks in a contention group initiated by each team by setting the value of the *thread-limit-var* ICV for the initial task to an implementation defined value greater than zero. If the **thread_limit** clause is specified, the number of threads will be less than or equal to *threadlim*. Otherwise, if the *teams-thread-limit-var* ICV is greater than zero, the effect on a **teams** construct is as if the **thread_limit** clause was specified with a *threadlim* that evaluates to an implementation defined value less than or equal to the *teams-thread-limit-var* ICV.

Cross References

- **target** Construct, see Section 15.8
- **teams** Construct, see Section 12.2

15.4 Device Initialization

Execution Model Events

The *device-initialize* event occurs in a thread that begins initialization of OpenMP on the device, after OpenMP initialization of the device, which may include device-side tool initialization, completes. The *device-load* event for a code block for a target device occurs in some thread before any thread executes code from that code block on that target device. The *device-unload* event for a target device occurs in some thread whenever a code block is unloaded from the device. The *device-finalize* event for a target device that has been initialized occurs in some thread before an OpenMP implementation shuts down.

Tool Callbacks

A thread dispatches a registered **device_initialize** callback for each occurrence of a *device-initialize* event in that thread. A thread dispatches a registered **device_load** callback for each occurrence of a *device-load* event in that thread. A thread dispatches a registered **device_unload** callback for each occurrence of a *device-unload* event in that thread. A thread dispatches a registered **device_finalize** callback for each occurrence of a *device-finalize* event in that thread.

Restrictions

Restrictions to OpenMP device initialization are as follows:

- No thread may offload execution of a construct to a device until a dispatched **device_initialize** callback completes.
- No thread may offload execution of a construct to a device after a dispatched **device_finalize** callback occurs.

Cross References

- **device_finalize** Callback, see Section 35.2

- **device_initialize** Callback, see Section 35.1
- **device_load** Callback, see Section 35.3
- **device_unload** Callback, see Section 35.4

15.5 target_enter_data Construct

Name: **target_enter_data** **Category:** executable	**Association:** unassociated **Properties:** parallelism-generating, task-generating, device, device-affecting, data-mapping, map-entering

Clauses

depend, **device**, **if**, **map**, **nowait**, **priority**, **replayable**

Additional information

The **target_enter_data** directive may alternatively be specified with **target enter data** as the *directive-name*.

Binding

The binding task set for a **target_enter_data** region is the generating task, which is the target task generated by the **target_enter_data** construct. The **target_enter_data** region binds to the corresponding target task region.

Semantics

When a **target_enter_data** construct is encountered, the list items in the **map** clause are mapped to the device data environment according to the **map** clause semantics. The **target_enter_data** construct generates a target task. The generated task region encloses the **target_enter_data** region. If a **depend** clause is present, it is associated with the target task. If the **nowait** clause is present, execution of the target task may be deferred. If the **nowait** clause is not present, the target task is an included task.

All clauses are evaluated when the **target_enter_data** construct is encountered. The data environment of the target task is created according to the data-mapping attribute clauses on the **target_enter_data** construct, ICVs with data environment ICV scope, and any default data-sharing attribute rules that apply to the **target_enter_data** construct. If a variable or part of a variable is mapped by the **target_enter_data** construct, the variable has a default data-sharing attribute of shared in the data environment of the target task.

Assignment operations associated with mapping a variable (see Section 7.9.6) occur when the target task executes.

When an **if** clause is present and *if-expression* evaluates to *false*, the target device is the host device.

Execution Model Events

Events associated with a target task are the same as for the **task** construct defined in Section 14.1.

The *target-enter-data-begin* event occurs after creation of the target task and completion of all predecessor tasks that are not target tasks for the same device. The *target-enter-data-begin* event is a *target-task-begin* event. The *target-enter-data-end* event occurs after all other events associated with the **target_enter_data** construct.

Tool Callbacks

Callbacks associated with events for target tasks are the same as for the **task** construct defined in Section 14.1; **(***flags* **& ompt_task_target)** always evaluates to *true* in the dispatched callback.

A thread dispatches a registered **target_emi** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_target_enter_data** or **ompt_target_enter_data_nowait** if the **nowait** clause is present as its *kind* argument for each occurrence of a *target-enter-data-begin* event in that thread in the context of the target task on the host device. Similarly, a thread dispatches a registered **target_emi** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_target_enter_data** or **ompt_target_enter_data_nowait** if the **nowait** clause is present as its *kind* argument for each occurrence of a *target-enter-data-end* event in that thread in the context of the target task on the host device.

Restrictions

Restrictions to the **target_enter_data** construct are as follows:

- At least one **map** clause must appear on the directive.
- All **map** clauses must be map-entering clauses.

Cross References

- **depend** Clause, see Section 17.9.5
- **device** Clause, see Section 15.2
- **if** Clause, see Section 5.5
- **map** Clause, see Section 7.9.6
- **nowait** Clause, see Section 17.6
- **priority** Clause, see Section 14.9
- **replayable** Clause, see Section 14.6
- OMPT **scope_endpoint** Type, see Section 33.27
- OMPT **target** Type, see Section 33.34
- **target_emi** Callback, see Section 35.8

- **task** Construct, see Section 14.1
- OMPT **task_flag** Type, see Section 33.37

15.6 target_exit_data Construct

Name: **target_exit_data** **Category:** executable	**Association:** unassociated **Properties:** parallelism-generating, task-generating, device, device-affecting, data-mapping, map-exiting

Clauses

depend, **device**, **if**, **map**, **nowait**, **priority**, **replayable**

Additional information

The **target_exit_data** directive may alternatively be specified with **target exit data** as the *directive-name*.

Binding

The binding task set for a **target_exit_data** region is the generating task, which is the target task generated by the **target_exit_data** construct. The **target_exit_data** region binds to the corresponding target task region.

Semantics

When a **target_exit_data** construct is encountered, the list items in the **map** clauses are unmapped from the device data environment according to the **map** clause semantics. The **target_exit_data** construct generates a target task. The generated task region encloses the **target_exit_data** region. If a **depend** clause is present, it is associated with the target task. If the **nowait** clause is present, execution of the target task may be deferred. If the **nowait** clause is not present, the target task is an included task.

All clauses are evaluated when the **target_exit_data** construct is encountered. The data environment of the target task is created according to the data-mapping attribute clauses on the **target_exit_data** construct, ICVs with data environment ICV scope, and any default data-sharing attribute rules that apply to the **target_exit_data** construct. If a variable or part of a variable is mapped by the **target_exit_data** construct, the variable has a default data-sharing attribute of shared in the data environment of the target task.

Assignment operations associated with mapping a variable (see Section 7.9.6) occur when the target task executes.

When an **if** clause is present and *if-expression* evaluates to *false*, the target device is the host device.

Execution Model Events

Events associated with a target task are the same as for the **task** construct defined in Section 14.1.

The *target-exit-data-begin* event occurs after creation of the target task and completion of all predecessor tasks that are not target tasks for the same device. The *target-exit-data-begin* event is a *target-task-begin* event. The *target-exit-data-end* event occurs after all other events associated with the **target_exit_data** construct.

Tool Callbacks

Callbacks associated with events for target tasks are the same as for the **task** construct defined in Section 14.1; **(***flags* **& ompt_task_target)** always evaluates to *true* in the dispatched callback.

A thread dispatches a registered **target_emi** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_target_exit_data** or **ompt_target_exit_data_nowait** if the **nowait** clause is present as its *kind* argument for each occurrence of a *target-exit-data-begin* event in that thread in the context of the target task on the host device. Similarly, a thread dispatches a registered **target_emi** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_target_exit_data** or **ompt_target_exit_data_nowait** if the **nowait** clause is present as its *kind* argument for each occurrence of a *target-exit-data-end* event in that thread in the context of the target task on the host device.

Restrictions

Restrictions to the **target_exit_data** construct are as follows:

- At least one **map** clause must appear on the directive.
- All **map** clauses must be map-exiting clauses.

Cross References

- **depend** Clause, see Section 17.9.5
- **device** Clause, see Section 15.2
- **if** Clause, see Section 5.5
- **map** Clause, see Section 7.9.6
- **nowait** Clause, see Section 17.6
- **priority** Clause, see Section 14.9
- **replayable** Clause, see Section 14.6
- OMPT **scope_endpoint** Type, see Section 33.27
- OMPT **target** Type, see Section 33.34
- **target_emi** Callback, see Section 35.8
- **task** Construct, see Section 14.1
- OMPT **task_flag** Type, see Section 33.37

15.7 target_data Construct

Name: **target_data** **Category:** executable	**Association:** block **Properties:** device, device-affecting, data-mapping, map-entering, map-exiting, parallelism-generating, sharing-task, task-generating

Clauses

affinity, **allocate**, **default**, **depend**, **detach**, **device**, **firstprivate**, **if**, **in_reduction**, **map**, **mergeable**, **nogroup**, **nowait**, **priority**, **private**, **shared**, **transparent**, **use_device_addr**, **use_device_ptr**

Clause set

data-environment-clause

Properties: required	**Members:** **map**, **use_device_addr**, **use_device_ptr**

Additional information

The **target_data** directive may alternatively be specified with **target data** as the *directive-name*.

Binding

The binding task set for a **target_data** region is the generating task. The **target_data** region binds to the region of the generating task.

Semantics

The **target_data** construct is a composite directive that provides a superset of the functionality provided by the **target_enter_data** and **target_exit_data** directives. The functionality added by the **target_data** directive is the inclusion of a task region for which data-sharing attributes may be specified. The effect of a **target_data** directive is equivalent to that of specifying three constituent directives, as described in the following, except expressions in all clauses are evaluated when the **target_data** construct is encountered.

The first constituent directive is a **target_enter_data** directive that is specified in the same code location as the **target_data** directive. The second constituent directive is a **task** directive that is specified immediately after the **target_enter_data** directive and that is associated with the structured block associated with the **target_data** directive. This **task** directive generates a sharing task. The third constituent directive is a **target_exit_data** directive that is specified immediately following the structured block that is associated with the **target_data** directive.

Since each constituent directive is a task-generating construct, the **target_data** directive generates three tasks. The task that is generated by the constituent **target_exit_data** directive is a dependent task of the task that is generated by the constituent **task** directive, which is a dependent task of the task that is generated by the constituent **target_enter_data** directive.

When an **if** clause is present on a **target_data** construct, the effect is as if the clause is present only on the constituent data-mapping constructs.

When a **nowait** clause is present on a **target_data** construct, the effect is as if the clause is present on the constituent data-mapping constructs. In addition, the task associated with the structured block may be deferred unless otherwise specified. If the **nowait** clause is not present, all tasks associated with the constituent directives are included tasks and, in addition, the task associated with the structured block is a merged task.

If the **transparent** clause is not specified then the effect is as if a **transparent** clause is specified such that *impex-type* evaluates to **omp_impex**. If the **mergeable** clause is not specified then the effect is as if a **mergeable** clause is specified such that *can_merge* evaluates to *true*.

When a **map** clause is present on a **target_data** construct, the effect is as if the clause is present on the constituent data-mapping constructs with substituted *map-type* modifiers that are determined according to the rules of map-type decay.

A list item that appears in a **map** clause may also appear in a **use_device_ptr** clause or a **use_device_addr** clause. If one or more **map** clauses are present, the list item conversions that are performed for any **use_device_ptr** and **use_device_addr** clauses occur after all variables are mapped on entry to the region according to those **map** clauses.

If the **nogroup** clause is not present, the **target_data** construct executes as if the structured block of the constituent **task** were enclosed in a **taskgroup** region. If the **nogroup** clause is present, no implicit **taskgroup** region is created.

Execution Model Events

The events associated with entering a **target_data** region are the same events as are associated with a **target_enter_data** construct, as described in Section 15.5, followed by the same events that are associated with a **task** construct, as described in Section 14.1.

The events associated with exiting a **target_data** region are the same events as are associated with a **target_exit_data** construct, as described in Section 15.6.

Tool Callbacks

The tool callbacks dispatched when entering a **target_data** region are the same as the tool callbacks dispatched when encountering a **target_enter_data** construct, as described in Section 15.5, followed by the same tool callbacks that are dispatched when encountering a **task** construct, as described in Section 14.1.

The tool callbacks dispatched when exiting a **target_data** region are the same as the tool callbacks dispatched when encountering a **target_exit_data** construct, as described in Section 15.6.

Cross References

- **affinity** Clause, see Section 14.10
- **allocate** Clause, see Section 8.6
- **default** Clause, see Section 7.5.1

- **depend** Clause, see Section 17.9.5
- **detach** Clause, see Section 14.11
- **device** Clause, see Section 15.2
- **firstprivate** Clause, see Section 7.5.4
- **if** Clause, see Section 5.5
- **in_reduction** Clause, see Section 7.6.12
- **map** Clause, see Section 7.9.6
- **mergeable** Clause, see Section 14.5
- **nogroup** Clause, see Section 17.7
- **nowait** Clause, see Section 17.6
- **priority** Clause, see Section 14.9
- **private** Clause, see Section 7.5.3
- **shared** Clause, see Section 7.5.2
- **target_enter_data** Construct, see Section 15.5
- **target_exit_data** Construct, see Section 15.6
- **task** Construct, see Section 14.1
- **transparent** Clause, see Section 17.9.6
- **use_device_addr** Clause, see Section 7.5.10
- **use_device_ptr** Clause, see Section 7.5.8

15.8 `target` Construct

Name: **target** **Category:** executable	**Association:** block **Properties:** parallelism-generating, team-generating, thread-limiting, exception-aborting, task-generating, device, device-affecting, data-mapping, map-entering, map-exiting, context-matching

Clauses

allocate, **default**, **defaultmap**, **depend**, **device**, **device_type**, **firstprivate**, **has_device_addr**, **if**, **in_reduction**, **is_device_ptr**, **map**, **nowait**, **priority**, **private**, **replayable**, **thread_limit**, **uses_allocators**

Binding

The binding task set for a **target** region is the generating task, which is the target task generated by the **target** construct. The **target** region binds to the corresponding target task region.

Semantics

The **target** construct generates a target task that encloses a **target** region to be executed on a device. If a **depend** clause is present, it is associated with the target task. The **device** and **device_type** clauses determine the device on which to execute the target task region. If the **nowait** clause is present, execution of the target tasks may be deferred. If the **nowait** clause is not present, the target task is an included tasks. The effect of any **map** clauses occur on entry to and exit from the generated **target** region, as specified in Section 7.9.6.

All clauses are evaluated when the **target** construct is encountered. The data environment of the target task is created according to the data-sharing attribute clauses and data-mapping attribute clauses on the **target** construct, ICVs with data environment ICV scope, and any default data-sharing attribute rules that apply to the **target** construct. If a variable or part of a variable is mapped by the **target** construct and does not appear as a list item in an **in_reduction** clause on the construct, the variable has a default data-sharing attribute of shared in the data environment of the target task. Assignment operations associated with mapping a variable (see Section 7.9.6) occur when the target task executes.

If the **device** clause is specified with the **ancestor** *device-modifier*, the encountering thread waits for completion of the **target** region on the parent device before resuming. For any list item that appears in a **map** clause on the same construct, if the corresponding list item exists in the device data environment of the parent device, it is treated as if it has a reference count of positive infinity.

When an **if** clause is present and *if-expression* evaluates to *false*, the effect is as if a **device** clause that specifies **omp_initial_device** as the device number is present, regardless of any other **device** clause on the directive.

If a procedure is explicitly or implicitly referenced in a **target** construct that does not specify a **device** clause in which the **ancestor** *device-modifier* appears then that procedure is treated as if its name had appeared in an **enter** clause on a declare target directive.

If a variable with static storage duration is declared in a **target** construct that does not specify a **device** clause in which the **ancestor** *device-modifier* appears then the named variable is treated as if it had appeared in an **enter** clause on a declare target directive if it is not a groupprivate variable and otherwise as if it had appeared in a **local** clause on a declare target directive.

If a list item in a **map** clause has a base pointer that is predetermined firstprivate or a base referencing variable for which the referring pointer is predetermined firstprivate (see Section 7.1.1), and on entry to the **target** region the list item is mapped, the firstprivate pointer is updated via corresponding pointer initialization.

Fortran

When an internal procedure is called in a **target** region, any references to variables that are host associated in the procedure have unspecified behavior.

Fortran

Execution Model Events

Events associated with a target task are the same as for the **task** construct defined in Section 14.1. Events associated with the initial task that executes the **target** region are defined in Section 14.13. The *target-submit-begin* event occurs prior to initiating creation of an initial task on a target device for a **target** region. The *target-submit-end* event occurs after initiating creation of an initial task on a target device for a **target** region. The *target-begin* event occurs after creation of the target task and completion of all predecessor tasks that are not target tasks for the same device. The *target-begin* event is a *target-task-begin* event. The *target-end* event occurs after the *target-submit-begin*, *target-submit-end* and *target-begin* events associated with the **target** construct and any events associated with **map** clauses on the construct. If the **nowait** clause is not present, the *target-end* event also occurs after all events associated with the target task and initial task but before the thread resumes execution of the encountering task.

Tool Callbacks

Callbacks associated with events for target tasks are the same as for the **task** construct defined in Section 14.1; **(***flags* **& ompt_task_target)** always evaluates to *true* in the dispatched callback.

A thread dispatches a registered **target_emi** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_target** or **ompt_target_nowait** if the **nowait** clause is present as its *kind* argument for each occurrence of a *target-begin* event in that thread in the context of the target task on the host device. Similarly, a thread dispatches a registered **target_emi** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_target** or **ompt_target_nowait** if the **nowait** clause is present as its *kind* argument for each occurrence of a *target-end* event in that thread in the context of the target task on the host device.

A thread dispatches a registered **target_submit_emi** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *target-submit-begin* event in that thread. Similarly, a thread dispatches a registered **target_submit_emi** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of a *target-submit-end* event in that thread. These callbacks occur in the context of the target task.

Restrictions

Restrictions to the **target** construct are as follows:

- Device-affecting constructs, other than **target** constructs for which the **ancestor** *device-modifier* is specified, must not be encountered during execution of a **target** region.
- The result of an **omp_set_default_device**, **omp_get_default_device**, or **omp_get_num_devices** routine called within a **target** region is unspecified.
- The effect of an access to a threadprivate variable in a **target** region is unspecified.
- If a list item in a **map** clause is a structure element, any other element of that structure that is referenced in the **target** construct must also appear as a list item in a **map** clause.
- A list item in a **map** clause that is specified on a **target** construct must have a base variable or base pointer.

- A list item in a data-sharing attribute clause that is specified on a **target** construct must not have the same base variable as a list item in a **map** clause on the construct.
- A variable referenced in a **target** region but not the **target** construct that is not declared in the **target** region must appear in a declare target directive.
- If a **device** clause is specified with the **ancestor** *device-modifier*, only the **device**, **firstprivate**, **private**, **defaultmap**, **nowait**, and **map** clauses may appear on the construct and no constructs or calls to routines are allowed inside the corresponding **target** region.
- If a **device** clause is specified with the **ancestor** *device-modifier*, whether a storage block on the encountering device that has no corresponding storage on the specified device may be mapped is implementation defined.
- Memory allocators that do not appear in a **uses_allocators** clause cannot appear as an allocator in an **allocate** clause or be used in the **target** region unless a **requires** directive with the **dynamic_allocators** clause is present in the same compilation unit.
- Any IEEE floating-point exception status flag, halting mode, or rounding mode set prior to a **target** region is unspecified in the region.
- Any IEEE floating-point exception status flag, halting mode, or rounding mode set in a **target** region is unspecified upon exiting the region.
- An OpenMP program must not rely on the value of a function address in a **target** region except for assignments, pointer association queries, and indirect calls.

C / C++

- Upon exit from a **target** region, the value of an attached pointer must not be different from the value when entering the region.

C / C++

C++

- The run-time type information (RTTI) of an object can only be accessed from the device on which it was constructed.
- Invoking a virtual member function of an object on a device other than the device on which the object was constructed results in unspecified behavior, unless the object is accessible and was constructed on the host device.
- If an object of polymorphic class type is destructed, virtual member functions of any previously existing corresponding objects in other device data environments must not be invoked.

C++

Fortran

- An attached pointer that is associated with a given pointer target must not be associated with a different pointer target upon exit from a **target** region.

- A reference to a coarray that is encountered on a non-host device must not be coindexed or appear as an actual argument to a procedure where the corresponding dummy argument is a coarray.
- If the allocation status of a mapped variable or a list item that appears in a **has_device_addr** clause that has the **ALLOCATABLE** attribute is unallocated on entry to a **target** region, the allocation status of the corresponding variable in the device data environment must be unallocated upon exiting the region.
- If the allocation status of a mapped variable or a list item that appears in a **has_device_addr** clause that has the **ALLOCATABLE** attribute is allocated on entry to a **target** region, the allocation status and shape of the corresponding variable in the device data environment may not be changed, either explicitly or implicitly, in the region after entry to it.
- If the association status of a list item with the **POINTER** attribute that appears in a **map** or **has_device_addr** clause on the construct is disassociated upon entry to the **target** region, the list item must be disassociated upon exit from the region.
- If the association status of a list item with the **POINTER** attribute that appears in a **map** or **has_device_addr** clause on the construct is associated upon entry to the **target** region, the list item must be associated with the same pointer target upon exit from the region.
- An OpenMP program must not rely on the association status of a procedure pointer in a **target** region except for calls to the **ASSOCIATED** inquiry function without the optional *proc-target* argument, pointer assignments and indirect calls.

Fortran

Cross References

- **allocate** Clause, see Section 8.6
- **default** Clause, see Section 7.5.1
- **defaultmap** Clause, see Section 7.9.9
- **depend** Clause, see Section 17.9.5
- **device** Clause, see Section 15.2
- **device_type** Clause, see Section 15.1
- **firstprivate** Clause, see Section 7.5.4
- **has_device_addr** Clause, see Section 7.5.9
- **if** Clause, see Section 5.5
- **in_reduction** Clause, see Section 7.6.12
- **is_device_ptr** Clause, see Section 7.5.7

- **map** Clause, see Section 7.9.6
- **nowait** Clause, see Section 17.6
- **priority** Clause, see Section 14.9
- **private** Clause, see Section 7.5.3
- **replayable** Clause, see Section 14.6
- OMPT **scope_endpoint** Type, see Section 33.27
- OMPT **target** Type, see Section 33.34
- **target_data** Construct, see Section 15.7
- **target_emi** Callback, see Section 35.8
- **target_submit_emi** Callback, see Section 35.10
- **task** Construct, see Section 14.1
- OMPT **task_flag** Type, see Section 33.37
- **thread_limit** Clause, see Section 15.3
- **uses_allocators** Clause, see Section 8.8

15.9 target_update Construct

Name: **target_update** **Category:** executable	**Association:** unassociated **Properties:** parallelism-generating, task-generating, device, device-affecting

Clauses

depend, **device**, **from**, **if**, **nowait**, **priority**, **replayable**, **to**

Clause set

Properties: required	**Members:** **from**, **to**

Additional information

The **target_update** directive may alternatively be specified with **target update** as the *directive-name*.

Binding

The binding task set for a **target_update** region is the generating task, which is the target task generated by the **target_update** construct. The **target_update** region binds to the corresponding target task region.

Semantics

The **target_update** directive makes the corresponding list items in the device data environment consistent with their original list items, according to the specified data-motion clauses. The **target_update** construct generates a target task. The generated task region encloses the **target_update** region. If a **depend** clause is present, it is associated with the target task. If the **nowait** clause is present, execution of the target task may be deferred. If the **nowait** clause is not present, the target task is an included task.

All clauses are evaluated when the **target_update** construct is encountered. The data environment of the target task is created according to data-motion clauses on the **target_update** construct, ICVs with data environment ICV scope, and any default data-sharing attribute rules that apply to the **target_update** construct. If a variable or part of a variable is a list item in a data-motion clause on the **target_update** construct, the variable has a default data-sharing attribute of shared in the data environment of the target task.

Assignment operations associated with any data-motion clauses occur when the target task executes. When an **if** clause is present and *if-expression* evaluates to *false*, no assignments occur.

Execution Model Events

Events associated with a target task are the same as for the **task** construct defined in Section 14.1.

The *target-update-begin* event occurs after creation of the target task and completion of all predecessor tasks that are not target tasks for the same device. The *target-update-end* event occurs after all other events associated with the **target_update** construct.

The *target-data-op-begin* event occurs in the **target_update** region before a thread initiates a data operation on the target device. The *target-data-op-end* event occurs in the **target_update** region after a thread initiates a data operation on the target device.

Tool Callbacks

Callbacks associated with events for target tasks are the same as for the **task** construct defined in Section 14.1; **(***flags* **& ompt_task_target)** always evaluates to *true* in the dispatched callback.

A thread dispatches a registered **target_emi** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_target_update** or **ompt_target_update_nowait** if the **nowait** clause is present as its *kind* argument for each occurrence of a *target-update-begin* event in that thread in the context of the target task on the host device. Similarly, a thread dispatches a registered **target_emi** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_target_update** or **ompt_target_update_nowait** if the **nowait** clause is present as its *kind* argument for each occurrence of a *target-update-end* event in that thread in the context of the target task on the host device.

A thread dispatches a registered **target_data_op_emi** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *target-data-op-begin* event in that thread. Similarly, a thread dispatches a registered **target_data_op_emi** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of a *target-data-op-end* event in that thread. These callbacks occur in the context of the target task.

Cross References

- **depend** Clause, see Section 17.9.5
- **device** Clause, see Section 15.2
- **from** Clause, see Section 7.10.2
- **if** Clause, see Section 5.5
- **nowait** Clause, see Section 17.6
- **priority** Clause, see Section 14.9
- **replayable** Clause, see Section 14.6
- OMPT **scope_endpoint** Type, see Section 33.27
- OMPT **target** Type, see Section 33.34
- **target_data_op_emi** Callback, see Section 35.7
- **target_emi** Callback, see Section 35.8
- **task** Construct, see Section 14.1
- OMPT **task_flag** Type, see Section 33.37
- **to** Clause, see Section 7.10.1

16 Interoperability

An OpenMP implementation may interoperate with one or more foreign runtime environments through the use of the **`interop`** construct that is described in this chapter, the **`interop`** operation for a declared function variant and the interoperability routines.

Cross References

- Interoperability Routines, see Chapter 26

16.1 `interop` Construct

Name: **`interop`**	**Association:** unassociated
Category: executable	**Properties:** device

Clauses

`depend`, **`destroy`**, **`device`**, **`init`**, **`nowait`**, **`use`**

Clause set

action-clause

Properties: required	**Members:** **`destroy`**, **`init`**, **`use`**

Binding

The binding task set for an **`interop`** region is the generating task. The **`interop`** region binds to the region of the generating task.

Semantics

The **`interop`** construct retrieves interoperability properties from the OpenMP implementation to enable interoperability with foreign execution contexts. When an **`interop`** construct is encountered, the encountering task executes the region.

The *interop-type* set for an **`init`** clause is the set of specified *interop-type* modifiers. For any other *action-clause* and the interoperability object that its argument specifies, the *interop-type* set is the set of modifiers that were specified by the **`init`** clause that initialized that interoperability object.

If the *interop-type* set includes **`targetsync`**, an empty mergeable task is generated. If the **`nowait`** clause is not present on the construct then the task is also an included task. If the *interop-type* set does not include **`targetsync`**, the **`nowait`** clause has no effect. Any **`depend`** clauses that are present on the construct apply to the generated task.

The **interop** construct ensures an ordered execution of the generated task relative to foreign tasks executed in the foreign execution context through the foreign synchronization object that is accessible through the **targetsync** property. When the creation of the foreign task precedes the encountering of an **interop** construct in happens-before order, the foreign task must complete execution before the generated task begins execution. Similarly, when the creation of a foreign task follows the encountering of an **interop** construct in between the encountering thread and either foreign tasks or OpenMP tasks by the **interop** construct.

Restrictions

Restrictions to the **interop** construct are as follows:

- A **depend** clause must only appear on the directive if the *interop-type* includes **targetsync**.
- An interoperability object must not be specified in more than one *action-clause* that appears on the **interop** construct.

Cross References

- **depend** Clause, see Section 17.9.5
- **destroy** Clause, see Section 5.7
- **device** Clause, see Section 15.2
- **init** Clause, see Section 5.6
- **nowait** Clause, see Section 17.6
- **use** Clause, see Section 16.1.2

16.1.1 OpenMP Foreign Runtime Identifiers

Allowed values for foreign runtime identifiers include the names (as string literals) and integer values that the OpenMP Additional Definitions document specifies and the corresponding **omp_ifr_*name*** values of the **interop_fr** OpenMP type. Implementation defined values for foreign runtime identifiers may also be supported.

16.1.2 use Clause

Name: use	**Properties:** *default*

Arguments

Name	Type	Properties
interop-var	variable of interop OpenMP type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`interop`

Semantics

The `use` clause specifies the *interop-var* that is used for the effects of the directive on which the clause appears. However, *interop-var* is not initialized, destroyed or otherwise modified. The *interop-type* set is inferred based on the *interop-type* modifiers used to initialize *interop-var*.

Restrictions

- The state of *interop-var* must be *initialized*.

Cross References

- `interop` Construct, see Section 16.1

16.1.3 `prefer-type` Modifier

Modifiers

Name	Modifies	Type	Properties
prefer-type	*init-var*	Complex, name: `prefer_type` Arguments: ***prefer-type-specification*** list of preference specification list item type (*default*)	complex, unique

Clauses

`init`

Semantics

The *prefer-type* modifier specifies a set of preferences to be used to initialize an interoperability object. Each preference specification list item specified in the *prefer-type-specification* argument is a preference specification that has the following syntax:

```
preference-specification:
    {preference-selector[, preference-selector[, ...]]}
    foreign-runtime-identifier

preference-selector:
    fr(foreign-runtime-identifier)
```

```
    attr(preference-property-extension[, preference-property-extension[, ...]])

preference-property-extension:
    ext-string-literal
```

Where *foreign-runtime-identifier* is a foreign runtime identifier and an implementation defined *ext-string-literal* is a string literal that must start with the **ompx_** prefix and must not include any commas (i.e., instances of the character '**,**').

The **fr** *preference-selector* specifies a foreign runtime environment identified by its foreign runtime identifier. The **attr** *preference-selector* specifies a preference for the attributes specified as its arguments.

If a *preference-specification* is a *foreign-runtime-identifier*, it is equivalent to specifying a *preference-specification* that uses the **fr** *preference-selector* and the foreign runtime identifier as its argument.

The interoperability object specified by the *init-var* argument of the **init** clause is initialized based on the first supported preference specification, if any, in left-to-right order. If the implementation does not support any of the specified preference specifications, *init-var* is initialized based on an implementation defined preference specification.

Restrictions

Restrictions to the *prefer-type* modifier are as follows:

- At most one **fr** *preference-selector* may be specified for each *preference-specification*.

Cross References

- **init** Clause, see Section 5.6

17 Synchronization Constructs and Clauses

A synchronization construct imposes an order on the completion of code executed by different threads through synchronizing flushes that are executed as part of the region that corresponds to the construct. Section 1.3.4 and Section 1.3.6 describe synchronization through the use of synchronizing flushes and atomic operations. Section 17.8.7 defines the behavior of synchronizing flushes that are implied at various other locations in an OpenMP program.

17.1 `hint` Clause

Name: `hint`	**Properties:** unique

Arguments

Name	Type	Properties
hint-expr	expression of sync_hint type	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`atomic`, **`critical`**

Semantics

The **`hint`** clause gives the implementation additional information about the expected runtime properties of the region that corresponds to the construct on which it appears and that can optionally be used to optimize the implementation. The presence of a **`hint`** clause does not affect the semantics of the construct. If no **`hint`** clause is specified for a construct that accepts it, the effect is as if **`omp_sync_hint_none`** had been specified as *hint-expr*.

Restrictions

- *hint-expr* must evaluate to a valid synchronization hint.

Cross References

- **atomic** Construct, see Section 17.8.5
- **critical** Construct, see Section 17.2
- OpenMP **sync_hint** Type, see Section 20.9.5

17.2 critical Construct

Name: **critical** Category: executable	Association: block **Properties:** mutual-exclusion, thread-limiting, thread-exclusive

Arguments

critical (*name*)

Name	Type	Properties
name	base language identifier	optional

Clauses

hint

Binding

The binding thread set for a **critical** region is all threads executing tasks in the contention group.

Semantics

The *name* argument is used to identify the **critical** construct. For any **critical** construct for which *name* is not specified, the effect is as if an identical (unspecified) name was specified. The regions that correspond to any **critical** construct of a given name are executed as if only by a single thread at a time among all threads associated with the contention group that execute the regions, without regard to the teams to which the threads belong.

C / C++

Identifiers used to identify a **critical** construct have external linkage and are in a name space that is separate from the name spaces used by labels, tags, members, and ordinary identifiers.

C / C++

Fortran

The names of **critical** constructs are global entities of the OpenMP program. If a name conflicts with any other entity, the behavior of the program is unspecified.

Fortran

Execution Model Events

The *critical-acquiring* event occurs in a thread that encounters the **critical** construct on entry to the **critical** region before initiating synchronization for the region. The *critical-acquired* event occurs in a thread that encounters the **critical** construct after it enters the region, but before it executes the structured block of the **critical** region. The *critical-released* event occurs in a thread that encounters the **critical** construct after it completes any synchronization on exit from the **critical** region.

Tool Callbacks

A thread dispatches a registered **mutex_acquire** callback for each occurrence of a *critical-acquiring* event in that thread. A thread dispatches a registered **mutex_acquired** callback for each occurrence of a *critical-acquired* event in that thread. A thread dispatches a registered **mutex_released** callback for each occurrence of a *critical-released* event in that thread. These callbacks occur in the task that encounters the **critical** construct. The callbacks should receive **ompt_mutex_critical** as their *kind* argument if practical, but a less specific kind is acceptable.

Restrictions

Restrictions to the **critical** construct are as follows:

- Unless **omp_sync_hint_none** is specified in a **hint** clause, the **critical** construct must specify a name.
- The *hint-expr* that is specified in the **hint** clause on each **critical** construct with the same *name* must evaluate to the same value.
- A **critical** region must not be nested (closely or otherwise) inside a **critical** region with the same *name*. This restriction is not sufficient to prevent deadlock.

Fortran

- If a *name* is specified on a **critical** directive and a paired end directive is specified, the same *name* must also be specified on the end directive.
- If no *name* appears on the **critical** directive and a paired end directive is specified, no *name* can appear on the end directive.

Fortran

Cross References

- **hint** Clause, see Section 17.1
- OMPT **mutex** Type, see Section 33.20
- **mutex_acquire** Callback, see Section 34.7.8
- **mutex_acquired** Callback, see Section 34.7.12
- **mutex_released** Callback, see Section 34.7.13
- OpenMP **sync_hint** Type, see Section 20.9.5

17.3 Barriers

17.3.1 barrier Construct

Name: **barrier**	**Association:** unassociated
Category: executable	**Properties:** team-executed

Binding

The binding thread set for a **barrier** region is the current team. A **barrier** region binds to the innermost enclosing parallel region.

Semantics

The **barrier** construct specifies an explicit barrier at the point at which the construct appears. Unless the binding region is canceled, all threads of the team that executes that binding region must enter the **barrier** region and complete execution of all explicit tasks bound to that binding region before any of the threads continue execution beyond the barrier.

The **barrier** region includes an implicit task scheduling point in the current task region.

Execution Model Events

The *explicit-barrier-begin* event occurs in each thread that encounters the **barrier** construct on entry to the **barrier** region. The *explicit-barrier-wait-begin* event occurs when a task begins a waiting interval in a **barrier** region. The *explicit-barrier-wait-end* event occurs when a task ends a waiting interval and resumes execution in a **barrier** region. The *explicit-barrier-end* event occurs in each thread that encounters the **barrier** construct after the barrier synchronization on exit from the **barrier** region. A *cancellation* event occurs if cancellation is activated at an implicit cancellation point in a **barrier** region.

Tool Callbacks

A thread dispatches a registered **sync_region** callback with **ompt_sync_region_barrier_explicit** as its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of an *explicit-barrier-begin* event. Similarly, a thread dispatches a registered **sync_region** callback with **ompt_sync_region_barrier_explicit** as its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of an *explicit-barrier-end* event. These callbacks occur in the context of the task that encountered the **barrier** construct.

A thread dispatches a registered **sync_region_wait** callback with **ompt_sync_region_barrier_explicit** as its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of an *explicit-barrier-wait-begin* event. Similarly, a thread dispatches a registered **sync_region_wait** callback with **ompt_sync_region_barrier_explicit** as its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of an *explicit-barrier-wait-end* event. These callbacks occur in the context of the task that encountered the **barrier** construct.

A thread dispatches a registered **cancel** callback with **ompt_cancel_detected** as its *flags* argument for each occurrence of a *cancellation* event in that thread. The callback occurs in the context of the encountering task.

Restrictions

Restrictions to the **barrier** construct are as follows:

- Each **barrier** region must be encountered by all threads in a team or by none at all, unless cancellation has been requested for the innermost enclosing parallel region.
- The sequence of worksharing regions and **barrier** regions encountered must be the same for every thread in a team.

Cross References

- **cancel** Callback, see Section 34.6
- OMPT **cancel_flag** Type, see Section 33.7
- OMPT **scope_endpoint** Type, see Section 33.27
- **sync_region** Callback, see Section 34.7.4
- OMPT **sync_region** Type, see Section 33.33
- **sync_region_wait** Callback, see Section 34.7.5

17.3.2 Implicit Barriers

This section describes the OMPT events and tool callbacks associated with implicit barriers, which occur at the end of various regions as defined in the description of the constructs to which they correspond. Implicit barriers are task scheduling points. For a description of task scheduling points, associated events, and tool callbacks, see Section 14.14.

Execution Model Events

The *implicit-barrier-begin* event occurs in each task that encounters an implicit barrier at the beginning of the implicit barrier region. The *implicit-barrier-wait-begin* event occurs when a task begins a waiting interval in an implicit barrier region. The *implicit-barrier-wait-end* event occurs when a task ends a waiting interval and resumes execution of an implicit barrier region. The *implicit-barrier-end* event occurs in a task that encounters an implicit barrier after the barrier synchronization on exit from an implicit barrier region. A *cancellation* event occurs if cancellation is activated at an implicit cancellation point in an implicit barrier region.

Tool Callbacks

A thread dispatches a registered **sync_region** callback for each *implicit-barrier-begin* and *implicit-barrier-end* event. Similarly, a thread dispatches a registered **sync_region_wait** callback for each *implicit-barrier-wait-begin* and *implicit-barrier-wait-end* event. All callbacks for implicit barrier events execute in the context of the encountering task.

For the implicit barrier at the end of a worksharing construct, the *kind* argument is **ompt_sync_region_barrier_implicit_workshare**. For the implicit barrier at the end of a **parallel** region, the *kind* argument is **ompt_sync_region_barrier_implicit_parallel**. For a barrier at the end of a **teams** region, the *kind* argument is **ompt_sync_region_barrier_teams**. For an extra barrier added by an OpenMP implementation, the *kind* argument is **ompt_sync_region_barrier_implementation**.

A thread dispatches a registered **cancel** callback with **ompt_cancel_detected** as its *flags* argument for each occurrence of a *cancellation* event in that thread. The callback occurs in the context of the encountering task.

Restrictions

Restrictions to implicit barriers are as follows:

- If a thread is in the **ompt_state_wait_barrier_implicit_parallel** state, a call to **get_parallel_info** may return a pointer to a copy of the data object associated with the parallel region rather than a pointer to the associated data object itself. Writing to the data object returned by **get_parallel_info** when a thread is in the **ompt_state_wait_barrier_implicit_parallel** state results in unspecified behavior.

Cross References

- **cancel** Callback, see Section 34.6
- OMPT **cancel_flag** Type, see Section 33.7
- **get_parallel_info** Entry Point, see Section 36.14
- OMPT **scope_endpoint** Type, see Section 33.27
- OMPT **state** Type, see Section 33.31
- **sync_region** Callback, see Section 34.7.4
- OMPT **sync_region** Type, see Section 33.33
- **sync_region_wait** Callback, see Section 34.7.5

17.3.3 Implementation-Specific Barriers

An OpenMP implementation can execute implementation-specific barriers that the OpenMP specification does not imply; therefore, no execution model events are bound to them. The implementation can handle these barriers like implicit barriers and dispatch all events as for implicit barriers. Any callbacks for these events use **ompt_sync_region_barrier_implementation** as the *kind* argument when they are dispatched.

17.4 taskgroup Construct

Name: **taskgroup**	Association: block
Category: executable	Properties: cancellable

Clauses

allocate, **task_reduction**

Binding

The binding task set of a **taskgroup** region is all tasks of the current team that are generated in the region. A **taskgroup** region binds to the innermost enclosing parallel region.

Semantics

The **taskgroup** construct specifies a wait on completion of the taskgroup set associated with the **taskgroup** region. When a thread encounters a **taskgroup** construct, it starts executing the region. An implicit task scheduling point occurs at the end of the **taskgroup** region. The current task is suspended at the task scheduling point until all tasks in the taskgroup set complete execution.

Execution Model Events

The *taskgroup-begin* event occurs in each thread that encounters the **taskgroup** construct on entry to the **taskgroup** region. The *taskgroup-wait-begin* event occurs when a task begins a waiting interval in a **taskgroup** region. The *taskgroup-wait-end* event occurs when a task ends a waiting interval and resumes execution in a **taskgroup** region. The *taskgroup-end* event occurs in each thread that encounters the **taskgroup** construct after the taskgroup synchronization on exit from the **taskgroup** region.

Tool Callbacks

A thread dispatches a registered **sync_region** callback with **ompt_sync_region_taskgroup** as its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *taskgroup-begin* event in the task that encounters the **taskgroup** construct. Similarly, a thread dispatches a registered **sync_region** callback with **ompt_sync_region_taskgroup** as its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of a *taskgroup-end* event in the task that encounters the **taskgroup** construct. These callbacks occur in the task that encounters the **taskgroup** construct.

A thread dispatches a registered **sync_region_wait** callback with **ompt_sync_region_taskgroup** as its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *taskgroup-wait-begin* event. Similarly, a thread dispatches a registered **sync_region_wait** callback with **ompt_sync_region_taskgroup** as its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of a *taskgroup-wait-end* event. These callbacks occur in the context of the task that encounters the **taskgroup** construct.

Cross References

- **allocate** Clause, see Section 8.6
- Task Scheduling, see Section 14.14
- OMPT **scope_endpoint** Type, see Section 33.27
- **sync_region** Callback, see Section 34.7.4
- OMPT **sync_region** Type, see Section 33.33
- **sync_region_wait** Callback, see Section 34.7.5
- **task_reduction** Clause, see Section 7.6.11

17.5 taskwait Construct

Name: **taskwait** Category: executable	Association: unassociated **Properties:** *default*

Clauses

depend, **nowait**, **replayable**

Binding

The binding thread set of the **taskwait** region is the current team. The **taskwait** region binds to the current task region.

Semantics

The **taskwait** construct specifies a wait on the completion of child tasks of the current task.

If no **depend** clause is present on the **taskwait** construct, the current task region is suspended at an implicit task scheduling point associated with the construct. The current task region remains suspended until all child tasks that it generated before the **taskwait** region complete execution.

If one or more **depend** clauses are present on the **taskwait** construct and the **nowait** clause is not also present, the behavior is as if these clauses were applied to a **task** construct with an empty associated structured block that generates a mergeable task and included task. Thus, the current task region is suspended until the predecessor tasks of this task complete execution.

If one or more **depend** clauses are present on the **taskwait** construct and the **nowait** clause is also present, the behavior is as if these clauses were applied to a **task** construct with an empty associated structured block that generates a task for which execution may be deferred. Thus, all predecessor tasks of this task must complete execution before any subsequently generated task that depends on this task starts its execution.

Execution Model Events

The *taskwait-begin* event occurs in a thread when it encounters a **taskwait** construct with no **depend** clause on entry to the **taskwait** region. The *taskwait-wait-begin* event occurs when a task begins a waiting interval in a region that corresponds to a **taskwait** construct with no **depend** clause. The *taskwait-wait-end* event occurs when a task ends a waiting interval and resumes execution from a region that corresponds to a **taskwait** construct with no **depend** clause. The *taskwait-end* event occurs in a thread when it encounters a **taskwait** construct with no **depend** clause after the taskwait synchronization on exit from the **taskwait** region.

The *taskwait-init* event occurs in a thread when it encounters a **taskwait** construct with one or more **depend** clauses on entry to the **taskwait** region. The *taskwait-complete* event occurs on completion of the dependent task that results from a **taskwait** construct with one or more **depend** clauses, in the context of the thread that executes the dependent task and before any subsequently generated task that depends on the dependent task starts its execution.

Tool Callbacks

A thread dispatches a registered **sync_region** callback with **ompt_sync_region_taskwait** as its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *taskwait-begin* event in the task that encounters the **taskwait** construct. Similarly, a thread dispatches a registered **sync_region** callback with **ompt_sync_region_taskwait** as its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of a *taskwait-end* event in the task that encounters the **taskwait** construct. These callbacks occur in the task that encounters the **taskwait** construct.

A thread dispatches a registered **sync_region_wait** callback with **ompt_sync_region_taskwait** as its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *taskwait-wait-begin* event. Similarly, a thread dispatches a registered **sync_region_wait** callback with **ompt_sync_region_taskwait** as its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of a *taskwait-wait-end* event. These callbacks occur in the context of the task that encounters the **taskwait** construct.

A thread dispatches a registered **task_create** callback for each occurrence of a *taskwait-init* event in the context of the encountering task. In the dispatched callback, **(***flags* **& ompt_task_taskwait)** always evaluates to *true*. If the **nowait** clause is not present, **(***flags* **& ompt_task_undeferred)** also evaluates to *true*.

A thread dispatches a registered **task_schedule** callback for each occurrence of a *taskwait-complete* event. This callback has **ompt_taskwait_complete** as its *prior_task_status* argument.

Restrictions

Restrictions to the **taskwait** construct are as follows:

- The **mutexinoutset** *task-dependence-type* may not appear in a **depend** clause on a **taskwait** construct.

- If the *task-dependence-type* of a **depend** clause is **depobj** then the depend objects may not represent dependences of the **mutexinoutset** dependence type.
- The **nowait** clause may only appear on a **taskwait** directive if the **depend** clause is present.
- The **replayable** clause may only appear on a **taskwait** directive if the **depend** clause is present.

Cross References

- **depend** Clause, see Section 17.9.5
- **nowait** Clause, see Section 17.6
- **replayable** Clause, see Section 14.6
- OMPT **scope_endpoint** Type, see Section 33.27
- **sync_region** Callback, see Section 34.7.4
- OMPT **sync_region** Type, see Section 33.33
- **sync_region_wait** Callback, see Section 34.7.5
- **task** Construct, see Section 14.1
- OMPT **task_flag** Type, see Section 33.37
- **task_schedule** Callback, see Section 34.5.2
- OMPT **task_status** Type, see Section 33.38

17.6 nowait Clause

Name: nowait	**Properties:** outermost-leaf, unique, end-clause

Arguments

Name	Type	Properties
do_not_synchronize	expression of OpenMP logical type	optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

dispatch, **do**, **for**, **interop**, **scope**, **sections**, **single**, **target**, **target_data**, **target_enter_data**, **target_exit_data**, **target_update**, **taskwait**, **workshare**

Semantics

If *do_not_synchronize* evaluates to *true*, the **nowait** clause overrides any synchronization that would otherwise occur at the end of a construct. It can also specify that a semantic requirement set includes the *nowait* property. If *do_not_synchronize* is not specified, the effect is as if *do_not_synchronize* evaluates to *true*. If *do_not_synchronize* evaluates to *false*, the effect is as if the **nowait** clause is not specified on the directive.

If the construct includes an implicit barrier and *do_not_synchronize* evaluates to *true*, the **nowait** clause specifies that the barrier will not occur. If the construct includes an implicit barrier and the **nowait** is not specified, the barrier will occur.

For constructs that generate a task, if *do_not_synchronize* evaluates to *true*, the **nowait** clause specifies that the generated task may be deferred. If the **nowait** clause is not specified on the directive then the generated task is an included task (so it executes synchronously in the context of the encountering task).

For directives that generate a semantic requirement set, the **nowait** clause adds the *nowait* property to the set if *do-not-synchronize* evaluates to *true*.

Restrictions

Restrictions to the **nowait** clause are as follows:

- The *do_not_synchronize* argument must evaluate to the same value for all threads in the binding thread set, if defined for the construct on which the **nowait** clause appears.
- The *do_not_synchronize* argument must evaluate to the same value for all tasks in the binding task set, if defined for the construct on which the **nowait** clause appears.

Cross References

- **dispatch** Construct, see Section 9.7
- **do** Construct, see Section 13.6.2
- **for** Construct, see Section 13.6.1
- **interop** Construct, see Section 16.1
- **scope** Construct, see Section 13.2
- **sections** Construct, see Section 13.3
- **single** Construct, see Section 13.1
- **target** Construct, see Section 15.8
- **target_data** Construct, see Section 15.7

- **target_enter_data** Construct, see Section 15.5
- **target_exit_data** Construct, see Section 15.6
- **target_update** Construct, see Section 15.9
- **taskwait** Construct, see Section 17.5
- **workshare** Construct, see Section 13.4

17.7 nogroup Clause

Name: nogroup	Properties: outermost-leaf, unique

Arguments

Name	Type	Properties
do_not_synchronize	expression of OpenMP logical type	optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

target_data, **taskgraph**, **taskloop**

Semantics

If *do_not_synchronize* evaluates to *true*, the **nogroup** clause overrides any implicit **taskgroup** that would otherwise enclose the construct. If *do_not_synchronize* evaluates to *false*, the effect is as if the **nogroup** clause is not specified on the directive. If *do_not_synchronize* is not specified, the effect is as if *do_not_synchronize* evaluates to *true*.

Cross References

- **target_data** Construct, see Section 15.7
- **taskgraph** Construct, see Section 14.3
- **taskloop** Construct, see Section 14.2

17.8 OpenMP Memory Ordering

This sections describes constructs and clauses that support ordering of memory operations.

17.8.1 *memory-order* Clauses

Clause groups

Properties: exclusive, unique	**Members:** **Clauses** **acq_rel**, **acquire**, **relaxed**, **release**, **seq_cst**

Directives

atomic, **flush**

Semantics

The *memory-order* clause group defines a set of clauses that indicate the memory ordering requirements for the visibility of the effects of the constructs on which they may be specified.

Cross References

- **atomic** Construct, see Section 17.8.5
- **flush** Construct, see Section 17.8.6
- OpenMP Memory Consistency, see Section 1.3.6

17.8.1.1 acq_rel Clause

Name: acq_rel	**Properties:** unique

Arguments

Name	Type	Properties
use-semantics	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

atomic, **flush**

Semantics

If *use_semantics* evaluates to *true*, the **acq_rel** clause specifies for the construct to use acquire/release memory ordering semantics. If *use_semantics* evaluates to *false*, the effect is as if the **acq_rel** clause is not specified. If *use_semantics* is not specified, the effect is as if *use_semantics* evaluates to *true*.

Cross References

- **atomic** Construct, see Section 17.8.5
- **flush** Construct, see Section 17.8.6
- OpenMP Memory Consistency, see Section 1.3.6

17.8.1.2 acquire Clause

Name: **acquire**	**Properties:** unique

Arguments

Name	Type	Properties
use_semantics	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

atomic, **flush**

Semantics

If *use_semantics* evaluates to *true*, the **acquire** clause specifies for the construct to use acquire memory ordering semantics. If *use_semantics* evaluates to *false*, the effect is as if the **acquire** clause is not specified. If *use_semantics* is not specified, the effect is as if *use_semantics* evaluates to *true*.

Cross References

- **atomic** Construct, see Section 17.8.5
- **flush** Construct, see Section 17.8.6
- OpenMP Memory Consistency, see Section 1.3.6

17.8.1.3 `relaxed` Clause

Name: `relaxed`	Properties: unique

Arguments

Name	Type	Properties
use_semantics	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`atomic`, **`flush`**

Semantics

If *use_semantics* evaluates to *true*, the **`relaxed`** clause specifies for the construct to use relaxed memory ordering semantics. If *use_semantics* evaluates to *false*, the effect is as if the **`relaxed`** clause is not specified. If *use_semantics* is not specified, the effect is as if *use_semantics* evaluates to *true*.

Cross References

- **`atomic`** Construct, see Section 17.8.5
- **`flush`** Construct, see Section 17.8.6
- OpenMP Memory Consistency, see Section 1.3.6

17.8.1.4 `release` Clause

Name: `release`	Properties: unique

Arguments

Name	Type	Properties
use_semantics	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`atomic`, **`flush`**

Semantics

If *use_semantics* evaluates to *true*, the **release** clause specifies for the construct to use release memory ordering semantics. If *use_semantics* evaluates to *false*, the effect is as if the **release** clause is not specified. If *use_semantics* is not specified, the effect is as if *use_semantics* evaluates to *true*.

Cross References

- **atomic** Construct, see Section 17.8.5
- **flush** Construct, see Section 17.8.6
- OpenMP Memory Consistency, see Section 1.3.6

17.8.1.5 seq_cst Clause

Name: seq_cst	Properties: unique

Arguments

Name	Type	Properties
use_semantics	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

atomic, **flush**

Semantics

If *use_semantics* evaluates to *true*, the **seq_cst** clause specifies for the construct to use sequentially consistent memory ordering semantics. If *use_semantics* evaluates to *false*, the effect is as if the **seq_cst** clause is not specified. If *use_semantics* is not specified, the effect is as if *use_semantics* evaluates to *true*.

Cross References

- **atomic** Construct, see Section 17.8.5
- **flush** Construct, see Section 17.8.6
- OpenMP Memory Consistency, see Section 1.3.6

17.8.2 *atomic* Clauses

Clause groups

Properties: exclusive, unique	Members: **Clauses** `read`, `update`, `write`

Directives

`atomic`

Semantics

The *atomic* clause group defines a set of clauses that defines the semantics for which a directive enforces atomicity. If a construct accepts the *atomic* clause group and no member of the clause group is specified, the effect is as if the `update` clause is specified.

Cross References

- `atomic` Construct, see Section 17.8.5

17.8.2.1 `read` Clause

Name: `read`	Properties: innermost-leaf, unique

Arguments

Name	Type	Properties
use_semantics	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`atomic`

Semantics

If *use_semantics* evaluates to *true*, the `read` clause specifies that the `atomic` construct has atomic read semantics, which read the value of the shared variable atomically. If *use_semantics* evaluates to *false*, the effect is as if the `read` clause is not specified. If *use_semantics* is not specified, the effect is as if *use_semantics* evaluates to *true*.

Cross References

- `atomic` Construct, see Section 17.8.5

17.8.2.2 `update` Clause

Name: `update`	**Properties:** innermost-leaf, unique

Arguments

Name	Type	Properties
use_semantics	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`atomic`

Semantics

If *use_semantics* evaluates to *true*, the **`update`** clause specifies that the **`atomic`** construct has atomic update semantics, which read and write the value of the shared variable atomically. If *use_semantics* evaluates to *false*, the effect is as if the **`update`** clause is not specified. If *use_semantics* is not specified, the effect is as if *use_semantics* evaluates to *true*.

Cross References

- **`atomic`** Construct, see Section 17.8.5

17.8.2.3 `write` Clause

Name: `write`	**Properties:** innermost-leaf, unique

Arguments

Name	Type	Properties
use_semantics	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`atomic`

Semantics

If *use_semantics* evaluates to *true*, the **write** clause specifies that the **atomic** construct has atomic write semantics, which write the value of the shared variable atomically. If *use_semantics* evaluates to *false*, the effect is as if the **write** clause is not specified. If *use_semantics* is not specified, the effect is as if *use_semantics* evaluates to *true*.

Cross References

- **atomic** Construct, see Section 17.8.5

17.8.3 *extended-atomic* Clauses

Clause groups

Properties: unique	Members:
	Clauses
	capture, **compare**, **fail**, **weak**

Directives

atomic

Semantics

The *extended-atomic* clause group defines a set of clauses that extend the atomicity semantics specified by members of the *atomic* clause group.

Restrictions

Restrictions to the *extended-atomic* clause group are as follows:

- The **compare** clause may not be specified such that *use_semantics* evaluates to *false* if the **weak** clause is specified such that *use_semantics* evaluates to *true*.

Cross References

- **atomic** Construct, see Section 17.8.5
- *atomic* Clauses, see Section 17.8.2

17.8.3.1 capture Clause

Name: capture	Properties: innermost-leaf, unique

Arguments

Name	Type	Properties
use_semantics	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

atomic

Semantics

If *use_semantics* evaluates to *true*, the **capture** clause extends the semantics of the **atomic** construct to have atomic captured update semantics, which capture the value of the shared variable being updated atomically. If *use_semantics* evaluates to *false*, the value is not captured. If *use_semantics* is not specified, the effect is as if *use_semantics* evaluates to *true*.

Cross References

- **atomic** Construct, see Section 17.8.5

17.8.3.2 compare Clause

Name: compare	Properties: innermost-leaf, unique

Arguments

Name	Type	Properties
use_semantics	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

atomic

Semantics

If *use_semantics* evaluates to *true*, the **compare** clause extends the semantics of the **atomic** construct with atomic conditional update semantics so the atomic update is performed conditionally. If *use_semantics* evaluates to *false*, the atomic update is performed unconditionally. If *use_semantics* is not specified, the effect is as if *use_semantics* evaluates to *true*.

Cross References

- **atomic** Construct, see Section 17.8.5

17.8.3.3 `fail` Clause

Name: `fail`	**Properties:** innermost-leaf, unique

Arguments

Name	Type	Properties
memorder	Keyword: **`acquire`**, **`relaxed`**, **`seq_cst`**	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`atomic`

Semantics

The **`fail`** clause extends the semantics of the **`atomic`** construct to specify the memory ordering requirements for any comparison performed by any atomic conditional update that fails. Its argument overrides any other specified memory ordering. If an **`atomic`** construct has atomic conditional update semantics and the **`fail`** clause is not specified, the effect is as if the **`fail`** clause is specified with a default argument that depends on the effective memory ordering. If the effective memory ordering is **`acq_rel`**, the default argument is **`acquire`**. If the effective memory ordering is **`release`**, the default argument is **`relaxed`**. For any other effective memory ordering, the default argument is equal to that effective memory ordering. If the **`atomic`** construct does not have atomic conditional update semantics, the **`fail`** clause has no effect.

Restrictions

Restrictions to the **`fail`** clause are as follows:

- *memorder* may not be **`acq_rel`** or **`release`**.

Cross References

- **`atomic`** Construct, see Section 17.8.5
- *memory-order* Clauses, see Section 17.8.1

17.8.3.4 `weak` Clause

Name: `weak`	**Properties:** innermost-leaf, unique

Arguments

Name	Type	Properties
use_semantics	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

atomic

Semantics

If *use_semantics* evaluates to *true*, the **weak** clause has the same effect as the **compare** clause and, in addition, the **atomic** construct has weak comparison semantics, which mean that the comparison may spuriously fail, evaluating to not equal even when the values are equal. If *use_semantics* evaluates to *false*, the semantics of the **atomic** construct are not extended. If *use_semantics* is not specified, the effect is as if *use_semantics* evaluates to *true*.

Note – Allowing for spurious failure by specifying a **weak** clause can result in performance gains on some systems when using compare-and-swap in a loop. For cases where a single compare-and-swap would otherwise be sufficient, using a loop over a **weak** compare-and-swap is unlikely to improve performance.

Cross References

- **atomic** Construct, see Section 17.8.5

17.8.4 memscope Clause

Name: **memscope**	**Properties:** unique

Arguments

Name	Type	Properties
scope-specifier	Keyword: **all**, **cgroup**, **device**	*default*

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

atomic, **flush**

Semantics

The **memscope** clause determines the binding thread set of the region that corresponds to the construct on which it is specified.

If the *scope-specifier* is **device**, the binding thread set consists of all threads on the device. If the *scope-specifier* is **cgroup**, the binding thread set consists of all threads that are executing tasks in the contention group. If the *scope-specifier* is **all**, the binding thread set consists of all threads on all devices.

Unless otherwise stated, the thread-set of any flushes that are performed in an **atomic** or **flush** region is the same as the binding thread set of the region, as determined by the **memscope** clause.

Restrictions

The restrictions for the **memscope** clause are as follows:

- The binding thread set defined by the *scope-specifier* of the **memscope** clause on an **atomic** construct must be a subset of the atomic scope of the atomically accessed memory.
- The binding thread set defined by the *scope-specifier* of the **memscope** clause on an **atomic** construct must be a subset of all threads that are executing tasks in the contention group if the size of the atomically accessed storage location is not 8, 16, 32, or 64 bits.

Cross References

- **atomic** Construct, see Section 17.8.5
- **flush** Construct, see Section 17.8.6

17.8.5 atomic Construct

Name: **atomic** **Category:** executable	**Association:** block : atomic **Properties:** mutual-exclusion, order-concurrent-nestable, simdizable

Clause groups

atomic, *extended-atomic*, *memory-order*

Clauses

hint, **memscope**

Binding

The **memscope** clause determines the binding thread set for an **atomic** region. If the **memscope** clause is not present, the behavior is as if the **memscope** clause appeared on the construct with the **device** *scope-specifier*.

Semantics

This section refers to the symbols defined for atomic structured blocks. The **atomic** construct ensures that a specific storage location is accessed atomically so that possible simultaneous reads and writes by multiple threads do not result in indeterminate values. An **atomic** region enforces exclusive access with respect to other **atomic** regions that access the same storage location *x* among all threads in the binding thread set without regard to the teams to which the threads belong.

An **atomic** construct with the **read** clause results in an atomic read of the storage location designated by *x*. An **atomic** construct with the **write** clause results in an atomic write of the storage location designated by *x*. An **atomic** construct with the **update** clause results in an atomic update of the storage location designated by *x* using the designated operator or intrinsic. Only the read and write of the storage location designated by *x* are performed mutually atomically. The evaluation of *expr* or *expr-list* need not be atomic with respect to the read or write of the storage location designated by *x*. No task scheduling points are allowed between the read and the write of the storage location designated by *x*.

If the **capture** clause is present, the atomic update is an atomic captured update — an atomic update to the storage location designated by *x* using the designated operator or intrinsic while also capturing the original or final value of the storage location designated by *x* with respect to the atomic update. The original or final value of the storage location designated by *x* is written in the storage location designated by *v* based on the base language semantics of atomic structured blocks of the **atomic** construct. Only the read and write of the storage location designated by *x* are performed mutually atomically. Neither the evaluation of *expr* or *expr-list*, nor the write to the storage location designated by *v*, need be atomic with respect to the read or write of the storage location designated by *x*.

If the **compare** clause is present, the atomic update is an atomic conditional update. For forms that use an equality comparison, the operation is an atomic compare-and-swap. It atomically compares the value of *x* to *e* and writes the value of *d* into the storage location designated by *x* if they are equal. Based on the base language semantics of the associated atomic structured block, the original or final value of the storage location designated by *x* is written to the storage location designated by *v*, which is allowed to be the same storage location as designated by *e*, or the result of the comparison is written to the storage location designated by *r*. Only the read and write of the storage location designated by *x* are performed mutually atomically. Neither the evaluation of either *e* or *d* nor writes to the storage locations designated by *v* and *r* need be atomic with respect to the read or write of the storage location designated by *x*.

C / C++

If the **compare** clause is present, forms that use *ordop* are logically an atomic maximum or minimum, but they may be implemented with a compare-and-swap loop with short-circuiting. For forms where *statement* is *cond-expr-stmt*, if the result of the condition implies that the value of *x* does not change then the update may not occur.

C / C++

If a *memory-order* clause is present, or implicitly provided by a **requires** directive, it specifies the effective memory ordering. Otherwise the effect is as if the **relaxed** *memory-order* clause is specified.

The **atomic** construct may be used to enforce memory consistency between threads, based on the guarantees provided by Section 1.3.6. A strong flush on the storage location designated by *x* is performed on entry to and exit from the atomic operation, ensuring that the set of all atomic operations applied to the same storage location in a race-free program has a total completion order. If the **write** or **update** clause is specified, the atomic operation is not an atomic conditional update for which the comparison fails, and the effective memory ordering is **release**, **acq_rel**, or **seq_cst**, the strong flush on entry to the atomic operation is also a release flush. If the **read** or **update** clause is specified and the effective memory ordering is **acquire**, **acq_rel**, or **seq_cst** then the strong flush on exit from the atomic operation is also an acquire flush. Therefore, if the effective memory ordering is not **relaxed**, release flushes and/or acquire flushes are implied and permit synchronization between the threads without the use of explicit **flush** directives.

For all forms of the **atomic** construct, any combination of two or more of these **atomic** constructs enforces mutually exclusive access to the storage locations designated by *x* among threads in the binding thread set. To avoid data races, all accesses of the storage locations designated by *x* that could potentially occur in parallel must be protected with an **atomic** construct.

atomic regions do not guarantee exclusive access with respect to any accesses outside of **atomic** regions to the same storage location *x* even if those accesses occur during a **critical** or **ordered** region, while a lock is owned by the executing task, or during the execution of a **reduction** clause.

However, other OpenMP synchronization can ensure the desired exclusive access. For example, a barrier that follows a series of atomic updates to *x* guarantees that subsequent accesses do not form a data race with the atomic accesses.

A compliant implementation may enforce exclusive access between **atomic** regions that update different storage locations. The circumstances under which this occurs are implementation defined.

If the storage location designated by *x* is not size-aligned (that is, if the byte alignment of *x* is not a multiple of the size of *x*), then the behavior of the **atomic** region is implementation defined.

Execution Model Events

The *atomic-acquiring* event occurs in the thread that encounters the **atomic** construct on entry to the **atomic** region before initiating synchronization for the region. The *atomic-acquired* event occurs in the thread that encounters the **atomic** construct after it enters the region, but before it executes the atomic structured block of the **atomic** region. The *atomic-released* event occurs in the thread that encounters the **atomic** construct after it completes any synchronization on exit from the **atomic** region.

Tool Callbacks

A thread dispatches a registered **mutex_acquire** callback for each occurrence of an *atomic-acquiring* event in that thread. A thread dispatches a registered **mutex_acquired** callback for each occurrence of an *atomic-acquired* event in that thread. A thread dispatches a registered **mutex_released** callback with **ompt_mutex_atomic** as the *kind* argument if practical, although a less specific *kind* may be used, for each occurrence of an *atomic-released* event in that thread. These callbacks occurs in the task that encounters the **atomic** construct.

Restrictions

Restrictions to the **atomic** construct are as follows:

- Constructs may not be encountered during execution of an **atomic** region.
- If a **capture** or **compare** clause is specified, the *atomic* clause must be **update**.
- If a **capture** clause is specified but the **compare** clause is not specified, an update-capture structured block must be associated with the construct.
- If both **capture** and **compare** clauses are specified, a conditional-update-capture structured block must be associated with the construct.
- If a **compare** clause is specified but the **capture** clause is not specified, a conditional-update structured block must be associated with the construct.
- If a **write** clause is specified, a write structured block must be associated with the construct.
- If a **read** clause is specified, a read structured block must be associated with the construct.
- If the *atomic* clause is **read** then the *memory-order* clause must not be **release**.
- If the *atomic* clause is **write** then the *memory-order* clause must not be **acquire**.
- The **weak** clause may only appear if the resulting atomic operation is an atomic conditional update for which the comparison tests for equality.

C / C++

- All atomic accesses to the storage locations designated by *x* throughout the OpenMP program are required to have a compatible type.
- The **fail** clause may only appear if the resulting atomic operation is an atomic conditional update.

C / C++

Fortran

- All atomic accesses to the storage locations designated by *x* throughout the OpenMP program are required to have the same type and type parameters.
- The **fail** clause may only appear if the resulting atomic operation is an atomic conditional update or an atomic update where *intrinsic-procedure-name* is either **MAX** or **MIN**.

Fortran

Cross References

- **barrier** Construct, see Section 17.3.1
- **critical** Construct, see Section 17.2
- **flush** Construct, see Section 17.8.6
- Lock Routines, see Chapter 28
- OpenMP Atomic Structured Blocks, see Section 6.3.3
- **hint** Clause, see Section 17.1
- **memscope** Clause, see Section 17.8.4
- OMPT **mutex** Type, see Section 33.20
- **mutex_acquire** Callback, see Section 34.7.8
- **mutex_acquired** Callback, see Section 34.7.12
- **mutex_released** Callback, see Section 34.7.13
- **ordered** Construct, see Section 17.10
- **requires** Directive, see Section 10.5

17.8.6 flush Construct

Name: flush **Category:** executable	**Association:** unassociated **Properties:** *default*

Arguments

flush (*list*)

Name	Type	Properties
list	list of variable list item type	optional

Clause groups

memory-order

Clauses

memscope

Binding

The **memscope** clause determines the binding thread set for a **flush** region. If the **memscope** clause is not present the behavior is as if the **memscope** clause appeared on the construct with the **device** *scope-specifier*.

Semantics

The **flush** construct executes the flush OpenMP operation. This operation makes the temporary view of the memory of a thread consistent with the memory and enforces an order on the memory operations of the variables explicitly specified or implied. Execution of a **flush** region affects the memory and it affects the temporary view of the memory of the encountering thread. It does not affect the temporary view of other threads. Other threads in the thread-set must themselves execute a flush in order to be guaranteed to observe the effects of the flush of the encountering thread. See the memory model description in Section 1.3 and the **memscope** clause description in Section 17.8.4 for more details on thread-sets.

If neither a *memory-order* clause nor a *list* argument appears on a **flush** construct then the behavior is as if the *memory-order* clause is **seq_cst**.

A **flush** construct with the **seq_cst** clause, executed on a given thread, operates as if all storage locations that are accessible to the thread are flushed by a strong flush; that is, the flush has the strong flush property. A **flush** construct with a *list* applies a strong flush to the items in the *list*, and the flush does not complete until the operation is complete for all specified list items. An implementation may implement a **flush** construct with a *list* by ignoring the *list* and treating it the same as a **flush** construct with the **seq_cst** clause.

If no list items are specified, the flush operation has the release flush property and/or the acquire flush property:

- If the *memory-order* clause is **seq_cst** or **acq_rel**, the flush is both a release flush and an acquire flush.
- If the *memory-order* clause is **release**, the flush is a release flush.
- If the *memory-order* clause is **acquire**, the flush is an acquire flush.

C / C++

If a pointer is present in the *list*, the pointer itself is flushed, not the storage locations to which the pointer refers.

A **flush** construct without a *list* corresponds to a call to **atomic_thread_fence**, where the argument is given by the identifier that results from prefixing **memory_order_** to the *memory-order* clause name.

For a **flush** construct without a *list*, the generated **flush** region implicitly performs the corresponding call to **atomic_thread_fence**. The behavior of an explicit call to **atomic_thread_fence** that occurs in an OpenMP program and does not have the argument **memory_order_consume** is as if the call is replaced by its corresponding **flush** construct.

C / C++

Fortran

If the list item or a subobject of the list item has the **POINTER** attribute, the allocation or association status of the **POINTER** item is flushed, but the pointer target is not. If the list item is of type **C_PTR**, the variable is flushed, but the storage location that corresponds to that address is not flushed. If the list item or the subobject of the list item has the **ALLOCATABLE** attribute and has an allocation status of allocated, the allocated variable is flushed; otherwise the allocation status is flushed.

Fortran

Execution Model Events

The *flush* event occurs in a thread that encounters the **flush** construct.

Tool Callbacks

A thread dispatches a registered **flush** callback for each occurrence of a *flush* event in that thread.

Restrictions

Restrictions to the **flush** construct are as follows:

- If a *memory-order* clause is specified, the *list* argument must not be specified.
- The *memory-order* clause must not be **relaxed**.

Cross References

- **flush** Callback, see Section 34.7.15
- **memscope** Clause, see Section 17.8.4

17.8.7 Implicit Flushes

Flushes implied when executing an **atomic** region are described in Section 17.8.5.

A flush region that corresponds to a **flush** directive with the **release** clause present is implied at the following locations:

- During a barrier region;
- At entry to a **parallel** region;
- At entry to a **teams** region;
- At exit from a **critical** region;
- During an **omp_unset_lock** region;
- During an **omp_unset_nest_lock** region;
- During an **omp_fulfill_event** region;
- Immediately before every task scheduling point;

- At exit from the task region of each implicit task;
- At exit from an **ordered** region, if a **threads** clause or a **doacross** clause with a **source** *task-dependence-type* is present, or if no clauses are present; and
- During a **cancel** region, if the *cancel-var* ICV is *true*.

For a **target** construct, the thread-set of an implicit release flush that is performed in a target task during the generation of the **target** region and that is performed on exit from the initial task region that implicitly encloses the **target** region consists of the thread that executes the target task and the initial thread that executes the **target** region.

A flush region that corresponds to a **flush** directive with the **acquire** clause present is implied at the following locations:

- During a barrier region;
- At exit from a **teams** region;
- At entry to a **critical** region;
- If the region causes the lock to be set, during:
  - an **omp_set_lock** region;
  - an **omp_test_lock** region;
  - an **omp_set_nest_lock** region; and
  - an **omp_test_nest_lock** region;
- Immediately after every task scheduling point;
- At entry to the task region of each implicit task;
- At entry to an **ordered** region, if a **threads** clause or a **doacross** clause with a **sink** *task-dependence-type* is present, or if no clauses are present; and
- Immediately before a cancellation point, if the *cancel-var* ICV is *true* and cancellation has been activated.

For a **target** construct, the thread-set of an implicit acquire flush that is performed in a target task following the generation of the **target** region or that is performed on entry to the initial task region that implicitly encloses the **target** region consists of the thread that executes the target task and the initial thread that executes the **target** region.

Note – A flush region is not implied at the following locations:

- At entry to worksharing regions; and
- At entry to or exit from **masked** regions.

The synchronization behavior of implicit flushes is as follows:

- When a thread executes an **atomic** region for which the corresponding construct has the **release**, **acq_rel**, or **seq_cst** clause and specifies an atomic operation that starts a given release sequence, the release flush that is performed on entry to the atomic operation synchronizes with an acquire flush that is performed by a different thread and has an associated atomic operation that reads a value written by a modification in the release sequence.
- When a thread executes an **atomic** region for which the corresponding construct has the **acquire**, **acq_rel**, or **seq_cst** clause and specifies an atomic operation that reads a value written by a given modification, a release flush that is performed by a different thread and has an associated release sequence that contains that modification synchronizes with the acquire flush that is performed on exit from the atomic operation.
- When a thread executes a **critical** region that has a given *name*, the behavior is as if the release flush performed on exit from the region synchronizes with the acquire flush performed on entry to the next **critical** region with the same *name* that is performed by a different thread, if it exists.
- When a team executes a barrier region, the behavior is as if the release flush performed by each thread within the region, and the release flush performed by any other thread upon fulfilling the *allow-completion* event for a detachable task bound to the binding parallel region of the region, synchronizes with the acquire flush performed by all other threads within the region.
- When a thread executes a **taskwait** region that does not result in the creation of a dependent task and the task that encounters the corresponding **taskwait** construct has at least one child task, the behavior is as if each thread that executes a child task that is generated before the **taskwait** region performs a release flush upon completion of the associated structured block of the child task that synchronizes with an acquire flush performed in the **taskwait** region. If the child task is a detachable task, the thread that fulfills its *allow-completion* event performs a release flush upon fulfilling the event that synchronizes with the acquire flush performed in the **taskwait** region.
- When a thread executes a **taskgroup** region, the behavior is as if each thread that executes a remaining descendent task performs a release flush upon completion of the associated structured block of the descendent task that synchronizes with an acquire flush performed on exit from the **taskgroup** region. If the descendent task is a detachable task, the thread that fulfills its *allow-completion* event performs a release flush upon fulfilling the event that synchronizes with the acquire flush performed in the **taskgroup** region.
- When a thread executes an **ordered** region that does not arise from a stand-alone **ordered** directive, the behavior is as if the release flush performed on exit from the region synchronizes with the acquire flush performed on entry to an **ordered** region encountered in the next collapsed iteration to be executed by a different thread, if it exists.

- When a thread executes an **ordered** region that arises from a stand-alone **ordered** directive, the behavior is as if the release flush performed in the **ordered** region from a given source doacross iteration synchronizes with the acquire flush performed in all **ordered** regions executed by a different thread that are waiting for dependences on that doacross iteration to be satisfied.

- When a team begins execution of a **parallel** region, the behavior is as if the release flush performed by the primary thread on entry to the **parallel** region synchronizes with the acquire flush performed on entry to each implicit task that is assigned to a different thread.

- When an initial thread begins execution of a **target** region that is generated by a different thread from a target task, the behavior is as if the release flush performed by the generating thread in the target task synchronizes with the acquire flush performed by the initial thread on entry to its initial task region.

- When an initial thread completes execution of a **target** region that is generated by a different thread from a target task, the behavior is as if the release flush performed by the initial thread on exit from its initial task region synchronizes with the acquire flush performed by the generating thread in the target task.

- When a thread encounters a **teams** construct, the behavior is as if the release flush performed by the thread on entry to the **teams** region synchronizes with the acquire flush performed on entry to each initial task that is executed by a different initial thread that participates in the execution of the **teams** region.

- When a thread that encounters a **teams** construct reaches the end of the **teams** region, the behavior is as if the release flush performed by each different participating initial thread at exit from its initial task synchronizes with the acquire flush performed by the thread at exit from the **teams** region.

- When a task generates an explicit task that begins execution on a different thread, the behavior is as if the thread that is executing the generating task performs a release flush that synchronizes with the acquire flush performed by the thread that begins to execute the explicit task.

- When an undeferred task completes execution on a given thread that is different from the thread on which its generating task is suspended, the behavior is as if a release flush performed by the thread that completes execution of the associated structured block of the undeferred task synchronizes with an acquire flush performed by the thread that resumes execution of the generating task.

- When a dependent task with one or more antecedent tasks begins execution on a given thread, the behavior is as if each release flush performed by a different thread on completion of the associated structured block of a antecedent task synchronizes with the acquire flush performed by the thread that begins to execute the dependent task. If the antecedent task is a detachable task, the thread that fulfills its *allow-completion* event performs a release flush upon fulfilling the event that synchronizes with the acquire flush performed when the dependent task begins to execute.

- When a task begins execution on a given thread and it is mutually exclusive with respect to another dependence-compatible task that is executed by a different thread, the behavior is as if each release flush performed on completion of the dependence-compatible task synchronizes with the acquire flush performed by the thread that begins to execute the task.
- When a thread executes a **cancel** region, the *cancel-var* ICV is *true*, and cancellation is not already activated for the specified region, the behavior is as if the release flush performed during the **cancel** region synchronizes with the acquire flush performed by a different thread immediately before a cancellation point in which that thread observes cancellation was activated for the region.
- When a thread executes an **omp_unset_lock** region that causes the specified lock to be unset, the behavior is as if a release flush is performed during the **omp_unset_lock** region that synchronizes with an acquire flush that is performed during the next **omp_set_lock** or **omp_test_lock** region to be executed by a different thread that causes the specified lock to be set.
- When a thread executes an **omp_unset_nest_lock** region that causes the specified nestable lock to be unset, the behavior is as if a release flush is performed during the **omp_unset_nest_lock** region that synchronizes with an acquire flush that is performed during the next **omp_set_nest_lock** or **omp_test_nest_lock** region to be executed by a different thread that causes the specified nestable lock to be set.

17.9 OpenMP Dependences

This section describes constructs and clauses in OpenMP that support the specification and enforcement of dependences. OpenMP supports two kinds of dependences: task dependences, which enforce orderings between dependence-compatible tasks; and doacross dependences, which enforce orderings between doacross iterations of a loop.

17.9.1 *task-dependence-type* Modifier

Modifiers

Name	Modifies	Type	Properties
task-dependence-type	*all arguments*	Keyword: **depobj**, **in**, **inout**, **inoutset**, **mutexinoutset**, **out**	unique

Clauses

depend, **update**

Semantics

Clauses that are related to task dependences use the *task-dependence-type* modifier to identify the type of dependence relevant to that clause. The effect of the type of dependence is associated with locator list items as described with the **depend** clause, see Section 17.9.5.

Cross References

- **depend** Clause, see Section 17.9.5
- **update** Clause, see Section 17.9.4

17.9.2 Depend Objects

Depend objects are OpenMP objects that can be used to supply user-computed dependences to **depend** clauses. Depend objects must be accessed only through the **depobj** construct, the **depend** clause and the asynchronous device routines; OpenMP programs that otherwise access depend objects are non-conforming programs. A depend object can be in one of the following states: *uninitialized* or *initialized*. Initially, depend objects are in the *uninitialized* state.

17.9.3 depobj Construct

Name: depobj **Category:** executable	**Association:** unassociated **Properties:** *default*

Clauses

destroy, **init**, **update**

Clause set

Properties: required	**Members: destroy**, **init**, **update**

Additional information

The **depobj** construct may alternatively be specified with a directive argument *depend-object* that is a depend object. If this syntax is used, the **init** clause must not be specified and instead the **depend** clause may be specified to initialize *depend-object* to represent a given dependence type and locator list item. With this syntax the **update** clause is only permitted to specify the *task-dependence-type* as if it is the sole argument of the clause, with the effect being that the specified dependence type applies to *depend-object*. With this syntax, any *update-var* or *destroy-var* that is specified in an **update** or **destroy** clause must be the same as *depend-object*. Finally, with this syntax only one clause may be specified and it must be **depend**, **update**, or **destroy**.

Binding

The binding thread set for a **depobj** region is the encountering thread.

Semantics

The **depobj** construct initializes, updates or destroys depend objects. If an **init** clause is specified, the state of the specified depend object is set to *initialized* and the depend object is set to represent the specified dependence type and locator list item. If an **update** clause is specified, the specified depend object is updated to represent the new dependence type. If a **destroy** clause is specified, the specified depend object is set to *uninitialized*.

Cross References

- **destroy** Clause, see Section 5.7
- **init** Clause, see Section 5.6
- **update** Clause, see Section 17.9.4

17.9.4 update Clause

Name: update	**Properties:** innermost-leaf, unique

Arguments

Name	Type	Properties
update-var	variable of OpenMP depend type	*default*

Modifiers

Name	Modifies	Type	Properties
task-dependence-type	*all arguments*	Keyword: **depobj**, **in**, **inout**, **inoutset**, **mutexinoutset**, **out**	unique
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

depobj

Semantics

The **update** clause sets the dependence type of *update-var* to *task-dependence-type*.

Restrictions

Restrictions to the **update** clause are as follows:

- *task-dependence-type* must not be **depobj**.
- The state of *update-var* must be *initialized*.
- If the locator list item represented by *update-var* is the **omp_all_memory** reserved locator, *task-dependence-type* must be either **out** or **inout**.

Cross References

- **depobj** Construct, see Section 17.9.3
- *task-dependence-type* Modifier, see Section 17.9.1

17.9.5 depend Clause

Name: **depend**	**Properties:** taskgraph-altering, task-inherited

Arguments

Name	Type	Properties
locator-list	list of locator list item type	*default*

Modifiers

Name	Modifies	Type	Properties
task-dependence-type	*all arguments*	Keyword: **depobj**, **in**, **inout**, **inoutset**, **mutexinoutset**, **out**	unique
iterator	*locator-list*	Complex, name: **iterator** Arguments: ***iterator-specifier*** list of iterator specifier list item type (*default*)	unique
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

dispatch, **interop**, **target**, **target_data**, **target_enter_data**, **target_exit_data**, **target_update**, **task**, **task_iteration**, **taskwait**

Semantics

The **depend** clause enforces additional constraints on the scheduling of tasks. These constraints establish dependences only between two dependence-compatible tasks: the antecedent task and the dependent task. The scheduling constraints are transitive so that the antecedent task must complete execution before any of its successor tasks execute. Similarly, the dependent task cannot start execution before all of its predecessor tasks complete execution. Task dependences are derived from the *task-dependence-type* and the list items in the *locator-list* argument.

One task, A, is a preceding dependence-compatible task of another task, B, if one of the following is true:

- A is a previously generated sibling task of B;
- A is a preceding dependence-compatible task of an importing task for which B is a child task;

- *A* is a child task of an exporting task that is a predecessor task of *B*;
- *A* is a child task of an undeferred exporting task that is a previously generated sibling task of *B*.

The storage location of a list item matches the storage location of another list item if they have the same storage location, or if any of the list items is **omp_all_memory**.

For the **in** *task-dependence-type*, if the storage location of at least one of the list items matches the storage location of a list item appearing in a **depend** clause with an **out**, **inout**, **mutexinoutset**, or **inoutset** *task-dependence-type* on a construct from which a preceding dependence-compatible task was generated then the generated task will be a dependent task of that preceding dependence-compatible task.

For the **out** *task-dependence-type* and **inout** *task-dependence-type*, if the storage location of at least one of the list items matches the storage location of a list item appearing in a **depend** clause with an **in**, **out**, **inout**, **mutexinoutset**, or **inoutset** *task-dependence-type* on a construct from which a preceding dependence-compatible task was generated then the generated task will be a dependent task of that preceding dependence-compatible task.

For the **mutexinoutset** *task-dependence-type*, if the storage location of at least one of the list items matches the storage location of a list item appearing in a **depend** clause with an **in**, **out**, **inout**, or **inoutset** *task-dependence-type* on a construct from which a preceding dependence-compatible task was generated then the generated task will be a dependent task of that preceding dependence-compatible task.

If a list item appearing in a **depend** clause with a **mutexinoutset** *task-dependence-type* on a task-generating construct matches a list item appearing in a **depend** clause with a **mutexinoutset** *task-dependence-type* on a different task-generating construct, and both constructs generate dependence-compatible tasks, the dependence-compatible tasks will be mutually exclusive tasks.

For the **inoutset** *task-dependence-type*, if the storage location of at least one of the list items matches the storage location of a list item appearing in a **depend** clause with an **in**, **out**, **inout**, or **mutexinoutset** *task-dependence-type* on a construct from which a preceding dependence-compatible task was generated then the generated task will be a dependent task of that preceding dependence-compatible task.

When the *task-dependence-type* is **depobj**, the behavior is as if the dependence type and locator list item that each specified depend object list item represents was specified by **depend** clauses on the current construct.

The list items that appear in the **depend** clause may reference any *iterator-identifier* defined in its *iterator* modifier.

The list items that appear in the **depend** clause may include array sections or the **omp_all_memory** reserved locator.

C / C++

The list items that appear in a **depend** clause may use shape-operators.

C / C++

Note – The enforced task dependence establishes a synchronization of memory accesses performed by a dependent task with respect to accesses performed by the antecedent tasks. However, the programmer must properly synchronize with respect to other concurrent accesses that occur outside of those tasks.

Execution Model Events

The *task-dependences* event occurs in a thread that encounters a task-generating construct or a **taskwait** construct with a **depend** clause immediately after the *task-create* event for the generated task or the *taskwait-init* event. The *task-dependence* event indicates an unfulfilled dependence for the generated task. This event occurs in a thread that observes the unfulfilled dependence before it is satisfied.

Tool Callbacks

A thread dispatches the **dependences** callback for each occurrence of the *task-dependences* event to announce its dependences with respect to the list items in the **depend** clause. A thread dispatches the **task_dependence** callback for a *task-dependence* event to report a dependence between a antecedent task (*src_task_data*) and a dependent task (*sink_task_data*).

Restrictions

Restrictions to the **depend** clause are as follows:

- List items, other than reserved locators, used in **depend** clauses of the same task or dependence-compatible tasks must indicate identical storage locations or disjoint storage locations.
- List items used in **depend** clauses cannot be zero-length array sections.
- The **omp_all_memory** reserved locator can only be used in a **depend** clause with an **out** or **inout** *task-dependence-type*.
- Array sections cannot be specified in **depend** clauses with the **depobj** *task-dependence-type*.
- List items used in **depend** clauses with the **depobj** *task-dependence-type* must be expressions of the **depend** OpenMP type that correspond to depend objects in the *initialized* state.
- List items that are expressions of the **depend** OpenMP type can only be used in **depend** clauses with the **depobj** *task-dependence-type*.

Fortran

- A common block name cannot appear in a **depend** clause.
- If a locator list item has the **ALLOCATABLE** attribute and its allocation status is unallocated, the behavior is unspecified.
- If a locator list item has the **POINTER** attribute and its association status is disassociated or undefined, the behavior is unspecified.

Fortran

C / C++

- A bit-field cannot appear in a **depend** clause.

C / C++

Cross References

- **dependences** Callback, see Section 34.7.1
- **dispatch** Construct, see Section 9.7
- Array Sections, see Section 5.2.5
- Array Shaping, see Section 5.2.4
- **interop** Construct, see Section 16.1
- **iterator** Modifier, see Section 5.2.6
- *task-dependence-type* Modifier, see Section 17.9.1
- **target** Construct, see Section 15.8
- **target_data** Construct, see Section 15.7
- **target_enter_data** Construct, see Section 15.5
- **target_exit_data** Construct, see Section 15.6
- **target_update** Construct, see Section 15.9
- **task** Construct, see Section 14.1
- **task_dependence** Callback, see Section 34.7.2
- **task_iteration** Directive, see Section 14.2.3
- **taskwait** Construct, see Section 17.5

17.9.6 `transparent` Clause

Name: **transparent**	**Properties:** unique

Arguments

Name	Type	Properties
impex-type	expression of impex OpenMP type	optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

target_data, **task**, **taskloop**

Semantics

The **transparent** clause controls the task dependence importing and exporting characteristics of any generated tasks of the construct on which it appears. If *impex-type* evaluates to **omp_not_impex** then the generated tasks are neither importing tasks nor exporting tasks and so are not transparent tasks. Otherwise the clause extends the set of dependence-compatible tasks of any child task of any of the generated tasks as follows. If *impex-type* evaluates to **omp_import** then the generated tasks are importing tasks. If *impex-type* evaluates to **omp_export** then the generated tasks are exporting tasks. If *impex-type* evaluates to **omp_impex** then the generated tasks are both importing tasks and exporting tasks.

The use of a variable in an *impex-type* expression causes an implicit reference to the variable in all enclosing constructs. The *impex-type* expression is evaluated in the context outside of the construct on which the clause appears. If *impex-type* is not specified, the effect is as if *impex-type* evaluates to **omp_impex**.

Cross References

- **depend** Clause, see Section 17.9.5
- **target_data** Construct, see Section 15.7
- **task** Construct, see Section 14.1
- **taskloop** Construct, see Section 14.2

17.9.7 doacross Clause

Name: doacross	Properties: required

Arguments

Name	Type	Properties
iteration-specifier	OpenMP iteration specifier	*default*

Modifiers

Name	Modifies	Type	Properties
dependence-type	*iteration-specifier*	Keyword: **sink**, **source**	required
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

ordered

Semantics

The **doacross** clause identifies doacross dependences that imply additional constraints on the scheduling of doacross logical iterations of a doacross loop nest. These constraints establish dependences only between doacross iterations. The *iteration-specifier* specifies a doacross iteration and is either a loop-iteration vector or uses the **omp_cur_iteration** keyword (see Section 6.4.3).

The **source** *dependence-type* specifies that the current doacross iteration is a source iteration and, thus, satisfies doacross dependences that arise from the current doacross iteration. If the **source** *dependence-type* is specified then the *iteration-specifier* argument is optional; if *iteration-specifier* is omitted, it is assumed to be **omp_cur_iteration**.

The **sink** *dependence-type* specifies the current doacross iteration is a sink iteration and, thus, has a doacross dependence, where *iteration-specifier* indicates the doacross iteration that satisfies the dependence. If *iteration-specifier* indicates a doacross iteration that does not occur in the doacross iteration space, the **doacross** clause is ignored. If all **doacross** clauses on an **ordered** construct are ignored then the construct is ignored.

Note – If the **sink** *dependence-type* is specified for an *iteration-specifier* that does not indicate an earlier iteration of the doacross iteration space, deadlock may occur.

Restrictions

Restrictions to the **doacross** clause are as follows:

- If *iteration-specifier* is a loop-iteration vector that has *n* elements, the innermost loop-nest-associated construct that encloses the construct on which the clause appears must specify an **ordered** clause for which the parameter value equals *n*.
- If *iteration-specifier* is specified with the **omp_cur_iteration** keyword and with **sink** as the *dependence-type* then it must be **omp_cur_iteration** - 1.
- If *iteration-specifier* is specified with **source** as the *dependence-type* then it must be **omp_cur_iteration**.
- If *iteration-specifier* is a loop-iteration vector and the **sink** *dependence-type* is specified then for each element, if the loop-iteration variable var_i has an integral or pointer type, the i^{th}

expression of *vector* must be computable without overflow in that type for any value of var_i that can encounter the construct on which the **doacross** clause appears.

C++

- If *iteration-specifier* is a loop-iteration vector and the **sink** *dependence-type* is specified then for each element, if the loop-iteration variable var_i is of a random access iterator type other than pointer type, the i^{th} expression of *vector* must be computable without overflow in the type that would be used by **std::distance** applied to variables of the type of var_i for any value of var_i that can encounter the construct on which the **doacross** clause appears.

C++

Cross References

- OpenMP Loop-Iteration Spaces and Vectors, see Section 6.4.3
- **ordered** Clause, see Section 6.4.6
- Stand-alone **ordered** Construct, see Section 17.10.1

17.10 ordered Construct

This section describes two forms for the **ordered** construct, the stand-alone **ordered** construct and the block-associated **ordered** construct. Both forms include the execution model events, tool callbacks, and restrictions listed in this section.

Execution Model Events

The *ordered-acquiring* event occurs in the task that encounters the **ordered** construct on entry to the **ordered** region before it initiates synchronization for the region. The *ordered-released* event occurs in the task that encounters the **ordered** construct after it completes any synchronization on exit from the region.

Tool Callbacks

A thread dispatches a registered **mutex_acquire** callback for each occurrence of an *ordered-acquiring* event in that thread. A thread dispatches a registered **mutex_released** callback with **ompt_mutex_ordered** as the *kind* argument if practical, although a less specific *kind* may be used, for each occurrence of an *ordered-released* event in that thread. These callback occur in the task that encounters the construct.

Restrictions

- The construct that corresponds to the binding region of an **ordered** region must specify an **ordered** clause.
- The construct that corresponds to the binding region of an **ordered** region must not specify a **reduction** clause with the **inscan** modifier.

- The region of a block-associated **ordered** construct must not have a binding region that corresponds to a construct in which a stand-alone **ordered** construct is closely nested.
- An **ordered** region that corresponds to an **ordered** construct with the **threads** or **doacross** clause may not be closely nested inside a **critical**, **ordered**, **loop**, **task**, or **taskloop** region (see Section 17.10).
- The doacross-affected loops of a doacross loop nest must be perfectly nested loops.
- The construct that corresponds to the binding region of an **ordered** region must not specify a **linear** clause.

C++

- The doacross-affected loops of a doacross loop nest must not be range-based **for** loops.

C++

Cross References

- OMPT **mutex** Type, see Section 33.20
- **mutex_acquire** Callback, see Section 34.7.8
- **mutex_released** Callback, see Section 34.7.13

17.10.1 Stand-alone ordered Construct

Name: **ordered** **Category:** executable	**Association:** unassociated **Properties:** mutual-exclusion

Clauses

doacross

Binding

The binding thread set for a stand-alone **ordered** region is the current team. A stand-alone **ordered** region binds to the innermost enclosing worksharing-loop region.

Semantics

The innermost enclosing worksharing-loop construct of a stand-alone **ordered** construct is associated with a doacross loop nest of the *n* doacross-affected loops. The stand-alone **ordered** construct specifies that execution must not violate doacross dependences as specified in the **doacross** clauses that appear on the construct. When a thread that is executing a doacross iteration encounters an **ordered** construct with one or more **doacross** clauses for which the **sink** *dependence-type* is specified, the thread waits until its dependences on all valid doacross iterations specified by the **doacross** clauses are satisfied before it continues execution. A specific dependence is satisfied when a thread that is executing the corresponding doacross iteration encounters an **ordered** construct with a **doacross** clause for which the **source** *dependence-type* is specified.

Execution Model Events

The *doacross-sink* event occurs in the task that encounters an **ordered** construct for each **doacross** clause for which the **sink** *dependence-type* is specified after the dependence is fulfilled. The *doacross-source* event occurs in the task that encounters an **ordered** construct with a **doacross** clause for which the **source** *dependence-type* is specified before signaling that the dependence has been fulfilled.

Tool Callbacks

A thread dispatches a registered **dependences** callback with all vector entries listed as **ompt_dependence_type_sink** in the *deps* argument for each occurrence of a *doacross-sink* event in that thread. A thread dispatches a registered **dependences** callback with all vector entries listed as **ompt_dependence_type_source** in the *deps* argument for each occurrence of a *doacross-source* event in that thread.

Restrictions

Additional restrictions to the stand-alone **ordered** construct are as follows:

- At most one **doacross** clause may appear on the construct with **source** as the *dependence-type*.
- All **doacross** clauses that appear on the construct must specify the same *dependence-type*.
- The construct must not be an orphaned construct.
- The construct must be closely nested inside a worksharing-loop construct.

Cross References

- OMPT **dependence_type** Type, see Section 33.10
- **dependences** Callback, see Section 34.7.1
- **doacross** Clause, see Section 17.9.7
- Worksharing-Loop Constructs, see Section 13.6

17.10.2 Block-associated ordered Construct

Name: ordered **Category:** executable	**Association:** block **Properties:** mutual-exclusion, simdizable, thread-limiting, thread-exclusive

Clause groups

parallelization-level

Binding

The binding thread set for a block-associated **ordered** region is the current team. A block-associated **ordered** region binds to the innermost enclosing region that corresponds to a construct for which a worksharing-loop construct or **simd** construct is a constituent construct.

Semantics

If no clauses are specified, the effect is as if the **threads** *parallelization-level* clause was specified. If the **threads** clause is specified, the threads in the team that is executing the worksharing-loop region execute **ordered** regions sequentially in the order of the collapsed iterations. If the **simd** *parallelization-level* clause is specified, the **ordered** regions encountered by any thread will execute one at a time in the order of the collapsed iterations. With either *parallelization-level*, execution of code outside the region for different collapsed iterations can run in parallel; execution of that code within the same collapsed iteration must observe any constraints imposed by the base language semantics.

When the thread that is executing the first collapsed iteration of the loop encounters a block-associated **ordered** construct, it can enter the **ordered** region without waiting. When a thread that is executing any subsequent collapsed iteration encounters a block-associated **ordered** construct, it waits at the beginning of the **ordered** region until execution of all **ordered** regions that belong to all previous collapsed iterations has completed. **ordered** regions that bind to different regions execute independently of each other.

Execution Model Events

The *ordered-acquired* event occurs in the task that encounters the **ordered** construct after it enters the region, but before it executes the associated structured block.

Tool Callbacks

A thread dispatches a registered **mutex_acquired** callback for each occurrence of an *ordered-acquired* event in that thread. This callback occurs in the task that encounters the construct.

Restrictions

Additional restrictions to the block-associated **ordered** construct are as follows:

- The construct is SIMDizable only if the **simd** *parallelization-level* clause is specified.
- If the **simd** *parallelization-level* clause is specified, the binding region must correspond to a construct for which the **simd** construct is a leaf construct.
- If the **threads** *parallelization-level* clause is specified, the binding region must correspond to a construct for which a worksharing-loop construct is a leaf construct.
- If the **threads** *parallelization-level* clause is specified and the binding region corresponds to a compound construct then the **simd** construct must not be a leaf construct unless the **simd** *parallelization-level* clause is also specified.
- During execution of the collapsed iteration associated with a loop-nest-associated directive, a thread must not execute more than one block-associated **ordered** region that binds to the corresponding region of the loop-nest-associated directive.
- An **ordered** clause with an argument value equal to the number of collapsed loops must appear on the construct that corresponds to the binding region, if the binding region is not a **simd** region.

Cross References

- *parallelization-level* Clauses, see Section 17.10.3
- Worksharing-Loop Constructs, see Section 13.6
- **mutex_acquired** Callback, see Section 34.7.12
- **ordered** Clause, see Section 6.4.6
- **simd** Construct, see Section 12.4

17.10.3 *parallelization-level* Clauses

Clause groups

Properties: unique	**Members:** **Clauses** **simd**, **threads**

Directives

ordered

Semantics

The *parallelization-level* clause group defines a set of clauses that indicate the level of parallelization with which to associate a construct.

Cross References

- Block-associated **ordered** Construct, see Section 17.10.2

17.10.3.1 threads Clause

Name: **threads**	**Properties:** innermost-leaf, unique

Arguments

Name	Type	Properties
apply-to-threads	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

ordered

Semantics

If *apply_to_threads* evaluates to *true*, the effect is as if the **threads** *parallelization-level* clause is specified. If *apply_to_threads* evaluates to *false*, the effect is as if the **threads** clause is not specified. If *apply_to_threads* is not specified, the effect is as if *apply_to_threads* evaluates to *true*.

Cross References

- Block-associated **ordered** Construct, see Section 17.10.2

17.10.3.2 simd Clause

Name: **simd**	**Properties:** innermost-leaf, unique

Arguments

Name	Type	Properties
apply-to-simd	expression of OpenMP logical type	constant, optional

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

ordered

Semantics

If *apply_to_simd* evaluates to *true*, the effect is as if the **simd** *parallelization-level* clause is specified. If *apply_to_simd* evaluates to *false*, the effect is as if the **simd** clause is not specified. If *apply_to_simd* is not specified, the effect is as if *apply_to_simd* evaluates to *true*.

Cross References

- Block-associated **ordered** Construct, see Section 17.10.2

18 Cancellation Constructs

This chapter defines constructs related to cancellation of OpenMP regions.

18.1 *cancel-directive-name* Clauses

Clause groups

Properties: exclusive, required, unique	Members:
	Clauses **`do`**, **`for`**, **`parallel`**, **`sections`**, **`taskgroup`**

Modifiers

Name	Modifies	Type	Properties
directive-name-modifier	*all arguments*	Keyword: *directive-name* (a directive name)	unique

Directives

`cancel`, **`cancellation_point`**

Semantics

For each directive that has the cancellable property (i.e., the directive may subject to cancellation and is a cancellable construct), a corresponding clause for which *clause-name* is the *directive-name* of that directive is a member of the *cancel-directive-name* clause group. Each member of the *cancel-directive-name* clause group takes an optional argument, *apply-to-directive*, that must be a constant expression of logical OpenMP type. For each member of the clause group, if *apply_to_directive* evaluates to *true* then the semantics of the construct on which the clause appears are applied for the directive with the *directive-name* specified by the clause. If *apply_to_directive* evaluates to *false*, the effect is equivalent to specifying an **`if`** clause for which *if-expression* evaluates to *false*. If *apply_to_directive* is not specified, the effect is as if *apply_to_directive* evaluates to *true*.

Restrictions

Restrictions to any clauses in the *cancel-directive-name* clause group are as follows:

- If *apply_to_directive* evaluates to *false* and an **`if`** clause is specified for the same constituent construct, *if-expression* must evaluate to *false*.

Cross References

- **cancel** Construct, see Section 18.2
- **cancellation_point** Construct, see Section 18.3
- **do** Construct, see Section 13.6.2
- **for** Construct, see Section 13.6.1
- **parallel** Construct, see Section 12.1
- **sections** Construct, see Section 13.3
- **taskgroup** Construct, see Section 17.4

18.2 cancel Construct

Name: **cancel**	Association: unassociated
Category: executable	Properties: *default*

Clause groups

cancel-directive-name

Clauses

if

Binding

The binding thread set of the **cancel** region is the current team. The binding region of the **cancel** region is the innermost enclosing region of the type that corresponds to *cancel-directive-name*.

Semantics

The **cancel** construct activates cancellation of the innermost enclosing region of the type specified by *cancel-directive-name*, which must be the *directive-name* of a cancellable construct. Cancellation of the binding region is activated only if the *cancel-var* ICV is *true*, in which case the **cancel** construct causes the encountering task to continue execution at the end of the binding region if *cancel-directive-name* is not **taskgroup**. If the *cancel-var* ICV is *true* and *cancel-directive-name* is **taskgroup**, the encountering task continues execution at the end of the current task region. If the *cancel-var* ICV is *false*, the **cancel** construct is ignored.

Threads check for active cancellation only at cancellation points that are implied at the following locations:

- **cancel** regions;
- **cancellation_point** regions;
- **barrier** regions;

- at the end of a worksharing-loop construct with a **nowait** clause and for which the same list item appears in both **firstprivate** and **lastprivate** clauses; and

- implicit barrier regions.

When a thread reaches one of the above cancellation points and if the *cancel-var* ICV is *true*, then:

- If the thread is at a **cancel** or **cancellation_point** region and *cancel-directive-name* is not **taskgroup**, the thread continues execution at the end of the canceled region if cancellation has been activated for the innermost enclosing region of the type specified.

- If the thread is at a **cancel** or **cancellation_point** region and *cancel-directive-name* is **taskgroup**, the encountering task checks for active cancellation of all of the taskgroup sets to which the encountering task belongs, and continues execution at the end of the current task region if cancellation has been activated for any of the taskgroup sets.

- If the encountering task is at a barrier region or at the end of a worksharing-loop construct with a **nowait** clause and for which the same list item appears in both **firstprivate** and **lastprivate** clauses, the encountering task checks for active cancellation of the innermost enclosing **parallel** region. If cancellation has been activated, then the encountering task continues execution at the end of the canceled region.

When cancellation of tasks is activated through a **cancel** construct with **taskgroup** for *cancel-directive-name*, the tasks that belong to the taskgroup set of the innermost enclosing **taskgroup** region will be canceled; that taskgroup set is then the canceled taskgroup set corresponding to that **cancel** region. The task that encountered that construct continues execution at the end of its task region, which implies completion of that task. Any task that belongs to the canceled taskgroup set and has already begun execution must run to completion or until a cancellation point is reached. Upon reaching a cancellation point and if cancellation is active, the task continues execution at the end of its task region, which implies the completion of the task. Any task that belongs to the canceled taskgroup set and that has not begun execution or that has not yet been fulfilled through an event variable may be discarded, which implies its completion.

When cancellation of tasks is activated through a **cancel** construct with *cancel-directive-name* other than **taskgroup**, each thread of the binding thread set resumes execution at the end of the canceled region if a cancellation point is encountered. If the canceled region is a **parallel** region, any tasks that have been created by a **task** or a **taskloop** construct and their descendent tasks are canceled according to the above **taskgroup** cancellation semantics. If the canceled region is not a **parallel** region, no task cancellation occurs.

C++

The usual C++ rules for object destruction are followed when cancellation is performed.

C++

Fortran

All private objects or subobjects with the **ALLOCATABLE** attribute that are allocated inside the canceled construct are deallocated.

Fortran

If the canceled construct specifies an original list-item updating clause, the final values of the list items that appear in those clauses are undefined.

When an **if** clause is present on a **cancel** construct and *if-expression* evaluates to *false*, the **cancel** construct does not activate cancellation. The cancellation point associated with the **cancel** construct is always encountered regardless of the value of *if-expression*.

Note – The programmer is responsible for releasing locks and other synchronization data structures that might cause a deadlock when a **cancel** construct is encountered and blocked threads cannot be canceled. The programmer is also responsible for ensuring proper synchronizations to avoid deadlocks that might arise from cancellation of regions that contain synchronization constructs.

Execution Model Events

If a task encounters a **cancel** construct that will activate cancellation then a *cancel* event occurs. A *discarded-task* event occurs for any discarded tasks.

Tool Callbacks

A thread dispatches a registered **cancel** callback for each occurrence of a *cancel* event in the context of the encountering task. **(***flags* **& ompt_cancel_activated)** always evaluates to *true* in the dispatched callback; **(***flags* **& ompt_cancel_parallel)** evaluates to *true* in the dispatched callback if *cancel-directive-name* is **parallel**; **(***flags* **& ompt_cancel_sections)** evaluates to *true* in the dispatched callback if *cancel-directive-name* is **sections**; **(***flags* **& ompt_cancel_loop)** evaluates to *true* in the dispatched callback if *cancel-directive-name* is **for** or **do**; and **(***flags* **& ompt_cancel_taskgroup)** evaluates to *true* in the dispatched callback if *cancel-directive-name* is **taskgroup**.

A thread dispatches a registered **cancel** callback with its *task_data* argument pointing to the **data** object associated with the discarded task and with **ompt_cancel_discarded_task** as its *flags* argument for each occurrence of a *discarded-task* event. The callback occurs in the context of the task that discards the task.

Restrictions

Restrictions to the **cancel** construct are as follows:

- The behavior for concurrent cancellation of a region and a region nested within it is unspecified.
- If *cancel-directive-name* is **taskgroup**, the **cancel** construct must be a closely nested construct of a **task** or a **taskloop** construct and the **cancel** region must be a closely nested region of a **taskgroup** region.
- If *cancel-directive-name* is not **taskgroup**, the **cancel** construct must be a closely nested construct of a construct that matches *cancel-directive-name*.

- A worksharing construct that is canceled must not have a **nowait** clause or a **reduction** clause with a user-defined reduction that uses **omp_orig** in the *initializer-expr* of the corresponding **declare_reduction** directive.
- A worksharing-loop construct that is canceled must not have an **ordered** clause or a **reduction** clause with the **inscan** *reduction-modifier*.
- When cancellation is active for a **parallel** region, a thread in the team that binds to that region must not be executing or encounter a worksharing construct with an **ordered** clause, a **reduction** clause with the **inscan** *reduction-modifier* or a **reduction** clause with a user-defined reduction that uses **omp_orig** in the *initializer-expr* of the corresponding **declare_reduction** directive.
- During execution of a construct that may be subject to cancellation, a thread must not encounter an orphaned cancellation point. That is, a cancellation point must only be encountered within that construct and must not be encountered elsewhere in its region.

Cross References

- **barrier** Construct, see Section 17.3.1
- **cancel** Callback, see Section 34.6
- OMPT **cancel_flag** Type, see Section 33.7
- **cancellation_point** Construct, see Section 18.3
- OMPT **data** Type, see Section 33.8
- **declare_reduction** Directive, see Section 7.6.14
- **firstprivate** Clause, see Section 7.5.4
- *cancel-var* ICV, see Table 3.1
- **if** Clause, see Section 5.5
- **nowait** Clause, see Section 17.6
- **omp_get_cancellation** Routine, see Section 30.1
- **ordered** Clause, see Section 6.4.6
- **private** Clause, see Section 7.5.3
- **reduction** Clause, see Section 7.6.10
- **task** Construct, see Section 14.1

18.3 `cancellation_point` Construct

Name: `cancellation_point`	**Association:** unassociated
Category: executable	**Properties:** *default*

Clause groups

cancel-directive-name

Additional information

The **`cancellation_point`** directive may alternatively be specified with **`cancellation point`** as the *directive-name*.

Binding

The binding thread set of the **`cancellation_point`** construct is the current team. The binding region of the **`cancellation_point`** region is the innermost enclosing region of the type that corresponds to *cancel-directive-name*.

Semantics

The **`cancellation_point`** construct introduces a user-defined cancellation point at which an implicit task or explicit task must check if cancellation of the innermost enclosing region of the type specified by *cancel-directive-name*, which must be the *directive-name* of a cancellable construct, has been activated. This construct does not implement any synchronization between threads or tasks. The semantics, including the execution model events and tool callbacks, for when an implicit task or explicit task reaches a user-defined cancellation point are identical to those of any other cancellation point and are defined in Section 18.2.

Restrictions

Restrictions to the **`cancellation point`** construct are as follows:

- A **`cancellation_point`** construct for which *cancel-directive-name* is **`taskgroup`** must be a closely nested construct of a **`task`** or **`taskloop`** construct, and the **`cancellation_point`** region must be a closely nested region of a **`taskgroup`** region.
- A **`cancellation_point`** construct for which *cancel-directive-name* is not **`taskgroup`** must be a closely nested construct inside a construct that matches *cancel-directive-name*.

Cross References

- *cancel-var* ICV, see Table 3.1
- **`omp_get_cancellation`** Routine, see Section 30.1

19 Composition of Constructs

This chapter defines rules and mechanisms for nesting regions and for combining constructs.

19.1 Compound Directive Names

Unless explicitly specified otherwise, the *directive-name* of a compound directive concatenates two or more directive names, with an intervening separating character, the directive-name separator between each of them. Each directive name, as well as any concatenation of consecutive directive names and their directive-name separator, is a constituent-directive name. Any constituent-directive name that is not itself a compound-directive name is a leaf-directive name.

Let *directive-name-A* refer to the first leaf-directive name that appears in a compound-directive name, and let *directive-name-B* refer to the constituent-directive name that forms the remainder of the compound-directive name. If the construct named by *directive-name-B* can be immediately nested inside the construct named by *directive-name-A*, the compound-directive name is a combined-directive name, the name of combined directive. Otherwise, the compound-directive name is a composite-directive name. Unless explicitly specified otherwise, the syntax for a compound-directive name is <*compound-directive-name*>, as described in the following grammar:

<*compound-directive-name*> **:**
    <*combined-directive-name*>
    <*composite-directive-name*>

<*combined-directive-name*> **:**
    <*directive-name-A*><*separator*><*directive-name-B*>

<*directive-name-A*> **:**
    <*parallelism-generating-directive-name*>
    <*thread-selecting-directive-name*>

<*directive-name-B*> **:**
    <*composite-directive-name*>
    <*parallelism-generating-directive-name*>
    <*combined-parallelism-generating-directive-name*>
    <*partitioned-directive-name*>
    <*combined-partitioned-directive-name*>
    <*thread-selecting-directive-name*>

```
    <combined-thread-selecting-directive-name>

<composite-directive-name>:
    <loop-distributed-composite-construct-name>
    <simd-partitioned-composite-construct-name>

<loop-distributed-composite-construct-name>:
    <distribute-directive-name><separator><parallel-loop-directive-name>

<simd-partitioned-composite-construct-name>:
  <simd-partitionable-directive-name><separator><simd-directive-name>
```

where:

- *<composite-directive-name>* is a composite-directive name;
- *<parallelism-generating-directive-name>* is the name of a parallelism-generating construct;
- *<combined-parallelism-generating-directive-name>* is a *<combined-directive-name>* for which *<directive-name-A>* is a *<parallelism-generating-directive-name>*.
- *<thread-selecting-directive-name>* is the name of a thread-selecting construct;
- *<combined-thread-selecting-directive-name>* is a *<combined-directive-name>* for which *<directive-name-A>* is a *<thread-selecting-directive-name>*.
- *<partitioned-directive-name>* is the name of a partitioned construct;
- *<combined-partitioned-directive-name>* is a *<combined-directive-name>* for which *<directive-name-A>* is a *<partitioned-directive-name>*;
- *<distribute-directive-name>* is **`distribute`**;
- *<parallel-loop-directive-name>* is the name of a combined construct for which *<directive-name-A>* is **`parallel`** and *<directive-name-B>* is the name of a worksharing-loop construct or a composite directive for which *<directive-name-A>* is the name of a worksharing-loop construct;
- *<simd-partitionable-directive-name>* is the name of a SIMD-partitionable construct;
- *<simd-directive-name>* is **`simd`**.

C / C++

- *<separator>*, the directive-name separator, is white space.

C / C++

Fortran

- *<separator>*, the directive-name separator, is white space or a plus sign (i.e., '+').

Fortran

The section that defines any composite directive for which its composite-directive name is not composed from its leaf-directive names in the fashion described above, such as those that combine a series of directives into one directive, also specifies the composite-directive name and its leaf directives. Unless otherwise specified, those leaf directives may be specified by their leaf-directive names in a *directive-name-modifier*.

Restrictions

Restrictions to compound-directive names are as follows:

- Any given instance of a compound-directive name must use the same character for all instances of <*separator*>.
- Leaf-directive names that include spaces are not permitted in a compound-directive name; they must instead be specified with an underscore replacing each space in the directive name.
- The leaf-directive names of a given compound-directive name must be unique.
- The construct corresponding to <*directive-name-B*> must be permitted to be immediately nested inside the construct corresponding to <*directive-name-A*>.
- If the first leaf-directive name of <*directive-name-B*> is the name of a worksharing construct or a thread-selecting construct then <*directive-name-A*> must be **`parallel`**.
- If <*directive-name-A*> and the first leaf-directive name of <*directive-name-B*> are the names of task-generating constructs then their respective explicit task regions must not bind to the same parallel region.
- The compound construct named by a given compound-directive name must have at most one constituent construct that is a map-entering construct.
- The compound construct named by a given compound-directive name must have at most one constituent construct that is a map-exiting construct.

Fortran

- If a directive name is ambiguous due to the use of optional intervening spaces between leaf-directive names, the directive-name separator must be a plus sign.

Fortran

Cross References

- **`distribute`** Construct, see Section 13.7
- **`parallel`** Construct, see Section 12.1
- **`simd`** Construct, see Section 12.4

19.2 Clauses on Compound Constructs

This section specifies the handling of clauses on compound constructs and the handling of implicit clauses that arise from any variable with predetermined data-sharing attributes on more than one leaf construct. For any clause for which a *directive-name-modifier* is specified, the effect of the modifier is applied prior to any of the rules that are specified in this section. Some clauses are permitted only on a single leaf construct of the compound construct, in which case the effect is as if the clause is applied to that specific construct. Other clauses that are permitted on more than one leaf construct have the effect as if they are applied to a subset of those constructs, as detailed in this section. Unless otherwise specified, the effect of a clause on a compound directive is as if it is applied to all leaf constructs that permit it (i.e., it has the default all-constituents property).

Unless otherwise specified, certain clause properties determine how each clause with those properties applies to any constituent directives of a compound directive on which it appears. Regardless of any specified *directive-name-modifier*, the effect of any clause with the once-for-all-constituents property on a compound construct is as if it is applied once to the compound construct regardless of how many constituent constructs to which they may apply.

The effect of any clause with the all-privatizing property on a compound directive is as if it is applied to all leaf constructs that permit the clause and to which a data-sharing attribute clause that may create a private copy of the same list item is applied. Unless otherwise specified, the effect of any clause with the innermost-leaf property on a compound construct is as if it is applied only to the innermost leaf construct that permits it. Unless otherwise specified, the effect of any clause with the outermost-leaf property on a compound construct is as if it is applied only to the outermost leaf construct that permits it.

The effect of the **firstprivate** clause is as if it is applied to one or more leaf constructs as follows:

- To the **distribute** construct if it is among the constituent constructs;
- To the **teams** construct if it is among the constituent constructs and the **distribute** construct is not;
- To a worksharing construct that accepts the clause if one is among the constituent constructs;
- To the **taskloop** construct if it is among the constituent constructs;
- To the **parallel** construct if it is among the constituent construct and neither a **taskloop** construct nor a worksharing construct that accepts the clause is among them;
- To the **target** construct if it is among the constituent constructs and the same list item neither appears in a **lastprivate** clause nor is the base variable or base pointer of a list item that appears in a **map** clause.

If the **parallel** construct is among the constituent constructs and the effect is not as if the **firstprivate** clause is applied to it by the above rules, then the effect is as if the **shared** clause with the same list item is applied to the **parallel** construct. If the **teams** construct is among the constituent constructs and the effect is not as if the **firstprivate** clause is applied to it by the above rules, then the effect is as if the **shared** clause with the same list item is applied to the **teams** construct.

The effect of the **lastprivate** clause is as if it is applied to all leaf constructs that permit the clause. If the **parallel** construct is among the constituent constructs and the list item is not also specified in the **firstprivate** clause, then the effect of the **lastprivate** clause is as if the **shared** clause with the same list item is applied to the **parallel** construct. If the **teams** construct is among the constituent constructs and the list item is not also specified in the **firstprivate** clause, then the effect of the **lastprivate** clause is as if the **shared** clause with the same list item is applied to the **teams** construct. If the **target** construct is among the constituent constructs and the list item is not the base variable or base pointer of a list item that appears in a **map** clause, the effect of the **lastprivate** clause is as if the same list item appears in a **map** clause with a *map-type* of **tofrom**.

The effect of the **reduction** clause is as if it is applied to all leaf constructs that permit the clause, except for the following constructs:

- The **parallel** construct, when combined with the **sections**, worksharing-loop, **loop**, or **taskloop** construct; and
- The **teams** construct, when combined with the **loop** construct.

For the **parallel** and **teams** constructs above, the effect of the **reduction** clause instead is as if each list item or, for any list item that is an array item, its corresponding base array or corresponding base pointer appears in a **shared** clause for the construct. If the **task** *reduction-modifier* is specified, the effect is as if it only modifies the behavior of the **reduction** clause on the innermost leaf construct that accepts the modifier (see Section 7.6.10). If the **inscan** *reduction-modifier* is specified, the effect is as if it modifies the behavior of the **reduction** clause on all constructs of the compound construct to which the clause is applied and that accept the modifier. If a list item in a **reduction** clause on a compound target construct does not have the same base variable or base pointer as a list item in a **map** clause on the construct, then the effect is as if the list item in the **reduction** clause appears as a list item in a **map** clause with a *map-type* of **tofrom**.

The effect of the **linear** clause is as if it is applied to the innermost leaf construct. Additionally, if the list item is not the loop-iteration variable of a construct for which **simd** is a constituent construct, the effect on the outer leaf constructs is as if the list item was specified in **firstprivate** and **lastprivate** clauses on the compound construct, with the rules specified above applied. If a list item of the **linear** clause is the loop-iteration variable of a construct for which the **simd** construct is a leaf construct and the variable is not declared in the construct, the effect on the outer leaf constructs is as if the list item was specified in a **lastprivate** clause on the compound construct with the rules specified above applied.

If the clauses have expressions on them, such as for various clauses where the argument of the clause is an expression, or *lower-bound*, *length*, or *stride* expressions inside array sections (or *subscript* and *stride* expressions in *subscript-triplet* for Fortran), or *linear-step* or *alignment* expressions, the expressions are evaluated immediately before the construct to which the clause has been split or duplicated per the above rules (therefore inside of the outer leaf constructs). However, the expressions inside the **num_teams** and **thread_limit** clauses are always evaluated before the outermost leaf construct.

The restriction that a list item may not appear in more than one data-sharing attribute clause with the exception of specifying a variable in both **firstprivate** and **lastprivate** clauses applies after the clauses are split or duplicated per the above rules.

Restrictions

Restrictions to clauses on compound constructs are as follows:

- A clause that appears on a compound construct must apply to at least one of the leaf constructs per the rules defined in this section.

Cross References

- **distribute** Construct, see Section 13.7
- **firstprivate** Clause, see Section 7.5.4
- **lastprivate** Clause, see Section 7.5.5
- **linear** Clause, see Section 7.5.6
- **loop** Construct, see Section 13.8
- **map** Clause, see Section 7.9.6
- **num_teams** Clause, see Section 12.2.1
- **parallel** Construct, see Section 12.1
- **reduction** Clause, see Section 7.6.10
- **sections** Construct, see Section 13.3
- **shared** Clause, see Section 7.5.2
- **simd** Construct, see Section 12.4
- **target** Construct, see Section 15.8
- **taskloop** Construct, see Section 14.2
- **teams** Construct, see Section 12.2
- **thread_limit** Clause, see Section 15.3

19.3 Compound Construct Semantics

The semantics of combined constructs are identical to that of explicitly specifying the first construct containing one instance of the second construct and no other statements.

Most composite constructs compose constructs that otherwise cannot be immediately nested to apply multiple loop-nest-associated constructs to the same canonical loop nest. The semantics of each of these composite constructs first apply the semantics of the enclosing construct as specified by *directive-name-A* and any clauses that apply to it. For each task as appropriate for the semantics of *directive-name-A*, the application of its semantics yields a nested loop of depth two in which the outer loop iterates over the chunks assigned to that task and the inner loop iterates over the collapsed iteration of each chunk. The semantics of *directive-name-B* and any clauses that apply to it are then applied to that inner loop. If *directive-name-A* is **taskloop** and *directive-name-B* is **simd** then for the application of the **simd** construct, the effect of any **in_reduction** clause is as if a **reduction** clause with the same reduction operator and list items is present.

For all compound constructs, tool callbacks are invoked as if the leaf constructs were explicitly nested. All compound constructs for which a loop-nest-associated construct is a leaf construct are themselves loop-nest-associated constructs.

Restrictions

Restrictions to compound construct are as follows:

- The restrictions of all constituent directives apply.
- If **distribute** is a constituent-directive name, the **linear** clause may only be specified for loop-iteration variables of loops that are associated with the construct and the **ordered** clause must not be specified.

Cross References

- **distribute** Construct, see Section 13.7
- **in_reduction** Clause, see Section 7.6.12
- **linear** Clause, see Section 7.5.6
- **ordered** Clause, see Section 6.4.6
- **parallel** Construct, see Section 12.1
- **reduction** Clause, see Section 7.6.10
- **simd** Construct, see Section 12.4
- **taskloop** Construct, see Section 14.2

Part III

Runtime Library Routines

20 Runtime Library Definitions

This chapter defines the naming convention for the OpenMP API routines. It also defines several OpenMP types. The names of OpenMP API routines have an **omp_** prefix. Names that begin with the **ompx_** prefix are reserved for routines that are implementation defined extensions.

For each base language, a compliant implementation must supply a set of definitions for the OpenMP API routines and the OpenMP types that are used for their arguments and return values. The C/C++ header file (**omp.h**) and the Fortran module file (**omp_lib**) or the deprecated Fortran include file (**omp_lib.h**) provide these definitions and must contain a declaration for each routine and predefined identifier as well as a definition of each OpenMP type. In addition, each set of definitions may specify other implementation defined values.

C / C++

The routines are external functions with “C” linkage. C/C++ prototypes for the routines shall be provided in the **omp.h** header file.

C / C++

Fortran

The Fortran OpenMP API routines are external procedures. The return values of these routines are of default kind, unless otherwise specified. Interface declarations for the Fortran routines shall be provided in the form of a Fortran **module** named **omp_lib** or the deprecated Fortran **include** file named **omp_lib.h**. Whether the **omp_lib.h** file provides derived-type definitions or those routines that require an explicit interface is implementation defined. Whether the **include** file or the **module** file (or both) is provided is also implementation defined. Whether any of the routines that take an argument are extended with a generic interface so arguments of different **KIND** type can be accommodated is implementation defined.

Fortran

Restrictions

The following restrictions apply to all routines and OpenMP types:

C++

- Enumeration OpenMP types provided in the **omp.h** header file shall not be scoped enumeration types unless explicitly allowed.

C++

Fortran

- Routines may not be called from **PURE** or **ELEMENTAL** procedures.
- Routines may not be called in **DO CONCURRENT** constructs.

Fortran

20.1 Predefined Identifiers

Predefined Identifiers

Name	Value	Properties
omp_curr_progress_width	see below	*default*
omp_fill	see below	*default*
omp_initial_device	**-1**	constant
omp_invalid_device	**< -1**	constant
omp_num_args	see below	*default*
omp_unassigned_thread	**< -1**	constant
openmp_version	see below	constant, Fortran-only

In addition to the predefined identifiers of OpenMP types that are defined with their corresponding OpenMP type, the OpenMP API includes the predefined identifiers shown above. The predefined identifiers **omp_invalid_device** and **omp_unassigned_thread** have implementation defined values less than -1. The predefined identifier **omp_num_args** can only be used in parameter list items and is a context-specific value that evaluates to the number of parameters of the associated declaration plus any variadic arguments that were passed, if any, at a given procedure call site. The predefined identifier **omp_curr_progress_width** is a context-specific value that represents the maximum size, in terms of hardware threads, of a progress unit that is available to threads that are executing tasks in the current contention group.

The predefined identifier **omp_fill** is a context-specific value that can only be used as a list item of the **counts** clause. It represents the number of logical iterations of a logical iteration space that remain after removing those specified by the other list items.

Fortran

The predefined identifiers are represented as default integer named constants. The predefined identifier **openmp_version** has a value *yyyymm* where *yyyy* and *mm* are the year and month designations of the version of the OpenMP API that the implementation supports. This value matches that of the C preprocessor macro **_OPENMP**, when a macro preprocessor is supported (see Section 5.3).

Fortran

20.2 Routine Bindings

Unless otherwise specified, the binding task set of any routine region is its encountering task and the binding thread set of any routine region is the encountering thread. That is, the default binding properties for routines are the encountering-task binding property and the encountering-thread binding property. However, the binding task set for all lock routine regions is all tasks in the contention group so all of those routines have the all-contention-group-tasks binding property. Further, the binding region of any routine that has a binding region for any type of region that is relevant to that routine region is the innermost enclosing region of that type. The binding thread set of several routines is all threads or all threads on the current device. Those routine have the all-threads binding property or the all-device-threads binding property.

20.3 Routine Argument Properties

Similarly to directive and clause arguments, routine arguments have properties that often specify constraints on their values. For all routines, if an argument is specified that does not conform to the constraints implied by its properties then the behavior is implementation defined. Routine properties include the properties that apply to the arguments of directives and clauses with the same meanings. The default property for all routine arguments is the required property. Routine arguments that have the optional property may be omitted in base languages for which a default value is defined. In addition, routine argument properties include ones that correspond to aspects of their base language prototypes, as shown in Table 20.1.

TABLE 20.1: Routine Argument Properties

Property	**Property Description**
C/C++ pointer property	A pointer type in C/C++, an array in Fortran
intent(in) property	An **intent(in)** argument in Fortran and, if type corresponds to a pointer type but not pointer to **char**, a **const** argument in C/C++
intent(out) property	An **intent(out)** argument in Fortran
ISO C property	Binds to an ISO C type in Fortran
pointer property	A pointer type in C/C++ and an assumed-size array in Fortran
pointer-to-pointer property	A pointer-to-pointer type in C/C++
procedure property	A function pointer type in C/C++ and a procedure type in Fortran
value property	A **value** argument in Fortran

20.4 General OpenMP Types

This section describes general OpenMP types.

20.4.1 OpenMP intptr Type

Name: intptr Properties: omp	Base Type: c_intptr_t

Type Definition

C / C++

```
typedef intptr_t omp_intptr_t;
```

C / C++

Fortran

```
integer (kind=omp_c_intptr_t_kind)
```

Fortran

The **intptr** OpenMP type is a signed integer type that is capable of holding a pointer on any device, and is equivalent to **intptr_t** on platforms that provide it.

20.4.2 OpenMP uintptr Type

Name: uintptr Properties: C/C++-only, omp	Base Type: c_uintptr_t

Type Definition

C / C++

```
typedef uintptr_t omp_uintptr_t;
```

C / C++

The **uintptr** OpenMP type is an unsigned integer type that is capable of holding a pointer on any device, and is equivalent to **uintptr_t** on platforms that provide it.

20.5 OpenMP Parallel Region Support Types

This section describes OpenMP types that support parallel regions.

20.5.1 OpenMP sched Type

Name: sched Properties: omp	Base Type: enumeration

Values

Name	Value	Properties
omp_sched_static	**0x1**	omp
omp_sched_dynamic	**0x2**	omp
omp_sched_guided	**0x3**	omp
omp_sched_auto	**0x4**	omp
omp_sched_monotonic	**0x80000000u**	omp

Type Definition

C / C++

```
typedef enum omp_sched_t {
  omp_sched_static     = 0x1,
  omp_sched_dynamic    = 0x2,
  omp_sched_guided     = 0x3,
  omp_sched_auto       = 0x4,
  omp_sched_monotonic = 0x80000000u
} omp_sched_t;
```

C / C++

Fortran

```
integer (kind=omp_sched_kind), &
   parameter :: omp_sched_static = &
    int(Z'1', kind=omp_sched_kind)
integer (kind=omp_sched_kind), &
   parameter :: omp_sched_dynamic = &
    int(Z'2', kind=omp_sched_kind)
integer (kind=omp_sched_kind), &
   parameter :: omp_sched_guided = &
    int(Z'3', kind=omp_sched_kind)
integer (kind=omp_sched_kind), &
   parameter :: omp_sched_auto = int(Z'4', kind=omp_sched_kind)
integer (kind=omp_sched_kind), &
   parameter :: omp_sched_monotonic = &
    int(Z'80000000', kind=omp_sched_kind)
```

Fortran

The **sched** type is used in routines that modify or retrieve the value of the *run-sched-var* ICV. Each of **omp_sched_static**, **omp_sched_dynamic**, **omp_sched_guided**, and **omp_sched_auto** can be combined with **omp_sched_monotonic** by using the + or | operator in C/C++ or the + operator in Fortran. If the schedule type is combined with the **omp_sched_monotonic**, the value corresponds to a schedule that is modified with the **monotonic** *ordering-modifier*. Otherwise, the value corresponds to a schedule that is modified with the **nonmonotonic** *ordering-modifier*.

Cross References

- *run-sched-var* ICV, see Table 3.1

20.6 OpenMP Tasking Support Types

This section describes OpenMP types that support tasking mechanisms.

20.6.1 OpenMP `event_handle` Type

Name: `event_handle` **Properties:** named-handle, omp, opaque	**Base Type:** `implementation-defined-int`

Type Definition

C / C++

```
typedef <implementation-defined-integral> omp_event_handle_t;
```

C / C++

Fortran

```
integer (kind=omp_event_handle_kind)
```

Fortran

The **`event_handle`** OpenMP type is an opaque type that represents events related to detachable tasks.

20.7 OpenMP Interoperability Support Types

This section describes OpenMP types that support interoperability mechanisms.

20.7.1 OpenMP `interop` Type

Name: `interop` **Properties:** named-handle, omp, opaque	**Base Type:** `implementation-defined-int`

Predefined Identifiers

Name	Value	Properties
`omp_interop_none`	0	*default*

Type Definition

C / C++

```
typedef <implementation-defined-integral> omp_interop_t;
```

C / C++

Fortran

```
integer (kind=omp_interop_kind)
```

Fortran

The **interop** OpenMP type is an opaque type that represents OpenMP interoperability objects, which thus have the opaque property. Interoperability objects may be initialized, destroyed or otherwise used by an **interop** construct and may be initialized to **omp_interop_none**.

Cross References

- **interop** Construct, see Section 16.1

20.7.2 OpenMP `interop_fr` Type

Name: `interop_fr` **Properties:** omp	**Base Type:** enumeration

Values

Name	Value	Properties
omp_ifr_last	**N**	omp

Type Definition

C / C++

```
typedef enum omp_interop_fr_t {
  omp_ifr_last = N
} omp_interop_fr_t;
```

C / C++

Fortran

```
integer (kind=omp_interop_fr_kind), &
   parameter :: omp_ifr_last = N
```

Fortran

The **interop_fr** OpenMP type represents supported foreign runtime environments. Each value of the **interop_fr** OpenMP type that an implementation provides will be available as **omp_ifr_*name***, where *name* is the name of the foreign runtime environment. Available names include those that are listed in the OpenMP Additional Definitions document; implementation defined names may also be supported. The value of **omp_ifr_last** is defined as one greater than the value of the highest value of the supported foreign runtime environments that are listed in the aforementioned document or are implementation defined.

Cross References

- OpenMP Contexts, see Section 9.1
- **omp_get_num_devices** Routine, see Section 24.3

20.7.3 OpenMP interop_property Type

Name: interop_property Properties: omp	Base Type: enumeration

Values

Name	Value	Properties
omp_ipr_fr_id	-1	omp
omp_ipr_fr_name	-2	omp
omp_ipr_vendor	-3	omp
omp_ipr_vendor_name	-4	omp
omp_ipr_device_num	-5	omp
omp_ipr_platform	-6	omp
omp_ipr_device	-7	omp
omp_ipr_device_context	-8	omp
omp_ipr_targetsync	-9	omp
omp_ipr_first	-9	omp

Type Definition

C / C++

```
typedef enum omp_interop_property_t {
  omp_ipr_fr_id          = -1,
  omp_ipr_fr_name        = -2,
  omp_ipr_vendor         = -3,
  omp_ipr_vendor_name    = -4,
  omp_ipr_device_num     = -5,
  omp_ipr_platform       = -6,
  omp_ipr_device         = -7,
  omp_ipr_device_context = -8,
  omp_ipr_targetsync     = -9,
  omp_ipr_first          = -9
} omp_interop_property_t;
```

C / C++

Fortran

```
integer (kind=omp_interop_property_kind), &
   parameter :: omp_ipr_fr_id = -1
integer (kind=omp_interop_property_kind), &
   parameter :: omp_ipr_fr_name = -2
integer (kind=omp_interop_property_kind), &
   parameter :: omp_ipr_vendor = -3
integer (kind=omp_interop_property_kind), &
   parameter :: omp_ipr_vendor_name = -4
integer (kind=omp_interop_property_kind), &
   parameter :: omp_ipr_device_num = -5
integer (kind=omp_interop_property_kind), &
   parameter :: omp_ipr_platform = -6
integer (kind=omp_interop_property_kind), &
   parameter :: omp_ipr_device = -7
integer (kind=omp_interop_property_kind), &
   parameter :: omp_ipr_device_context = -8
integer (kind=omp_interop_property_kind), &
   parameter :: omp_ipr_targetsync = -9
integer (kind=omp_interop_property_kind), &
   parameter :: omp_ipr_first = -9
```

Fortran

The **interop_property** OpenMP type is used in interoperability routines to represent interoperability properties. OpenMP reserves all negative values for interoperability properties, as listed in Table 20.2; implementation defined interoperability properties may use non-negative values. The special interoperability property, **omp_ipr_first**, will always have the lowest **interop_property** value, which may change in future versions of this specification. Valid values and types for the properties that Table 20.2 lists are specified in the OpenMP Additional Definitions document or are implementation defined unless otherwise specified. The **Contexts** column of Table 20.2 lists the OpenMP context that is relevant to the value.

Cross References

- OpenMP Contexts, see Section 9.1
- **omp_get_num_devices** Routine, see Section 24.3

20.7.4 OpenMP interop_rc Type

Name: interop_rc **Properties:** omp	**Base Type:** enumeration

TABLE 20.2: Required Values of the **interop_property** OpenMP Type

Enum Name	Contexts	Name	Property
omp_ipr_fr_id	all	**fr_id**	An **intptr_t** value that represents the foreign runtime environment ID of context
omp_ipr_fr_name	all	**fr_name**	C string value that represents the name of the foreign runtime environment of context
omp_ipr_vendor	all	**vendor**	An **intptr_t** that represents the vendor of context
omp_ipr_vendor_name	all	**vendor_name**	C string value that represents the vendor of context
omp_ipr_device_num	all	**device_num**	The OpenMP device number for the device in the range 0 to **omp_get_num_devices** inclusive
omp_ipr_platform	*target*	**platform**	A foreign platform handle usually spanning multiple devices
omp_ipr_device	*target*	**device**	A foreign device handle
omp_ipr_device_context	*target*	**device_context**	A handle to an instance of a foreign device context
omp_ipr_targetsync	*targetsync*	**targetsync**	A handle to a synchronization object of a foreign execution context

Values

Name	Value	Properties
omp_irc_no_value	**1**	omp
omp_irc_success	**0**	omp
omp_irc_empty	**-1**	omp
omp_irc_out_of_range	**-2**	omp
omp_irc_type_int	**-3**	omp
omp_irc_type_ptr	**-4**	omp
omp_irc_type_str	**-5**	omp
omp_irc_other	**-6**	omp

Type Definition

C / C++

```
typedef enum omp_interop_rc_t {
  omp_irc_no_value     = 1,
  omp_irc_success      = 0,
  omp_irc_empty        = -1,
  omp_irc_out_of_range = -2,
  omp_irc_type_int     = -3,
```

TABLE 20.3: Required Values for the **interop_rc** OpenMP Type

Enum Name	Description
omp_irc_no_value	Valid but no meaningful value available
omp_irc_success	Successful, value is usable
omp_irc_empty	The provided interoperability object is equal to omp_interop_none
omp_irc_out_of_range	Property ID is out of range, see Table 20.2
omp_irc_type_int	Property type is int; use omp_get_interop_int
omp_irc_type_ptr	Property type is pointer; use omp_get_interop_ptr
omp_irc_type_str	Property type is string; use omp_get_interop_str
omp_irc_other	Other error; use omp_get_interop_rc_desc

```
  omp_irc_type_ptr      = -4,
  omp_irc_type_str      = -5,
  omp_irc_other         = -6
} omp_interop_rc_t;
```

C / C++

Fortran

```
integer (kind=omp_interop_rc_kind), &
   parameter :: omp_irc_no_value = 1
integer (kind=omp_interop_rc_kind), &
   parameter :: omp_irc_success = 0
integer (kind=omp_interop_rc_kind), &
   parameter :: omp_irc_empty = -1
integer (kind=omp_interop_rc_kind), &
   parameter :: omp_irc_out_of_range = -2
integer (kind=omp_interop_rc_kind), &
   parameter :: omp_irc_type_int = -3
integer (kind=omp_interop_rc_kind), &
   parameter :: omp_irc_type_ptr = -4
integer (kind=omp_interop_rc_kind), &
   parameter :: omp_irc_type_str = -5
integer (kind=omp_interop_rc_kind), &
   parameter :: omp_irc_other = -6
```

Fortran

The **interop_rc** OpenMP type is used in several interoperability routines to specify their results. Table 20.3 describes the values that this type must include.

Cross References

- OpenMP **interop** Type, see Section 20.7.1
- OpenMP **interop_property** Type, see Section 20.7.3
- **omp_get_interop_int** Routine, see Section 26.2
- **omp_get_interop_ptr** Routine, see Section 26.3
- **omp_get_interop_rc_desc** Routine, see Section 26.7
- **omp_get_interop_str** Routine, see Section 26.4

20.8 OpenMP Memory Management Types

This section describes OpenMP types that support memory management.

20.8.1 OpenMP allocator_handle Type

Name: **allocator_handle** **Properties:** omp	**Base Type:** enumeration

Values

Name	Value	Properties
omp_null_allocator	0	omp
omp_default_mem_alloc	1	omp
omp_large_cap_mem_alloc	2	omp
omp_const_mem_alloc	3	omp
omp_high_bw_mem_alloc	4	omp
omp_low_lat_mem_alloc	5	omp
omp_cgroup_mem_alloc	6	omp
omp_pteam_mem_alloc	7	omp
omp_thread_mem_alloc	8	omp

Type Definition

C / C++

```
typedef enum omp_allocator_handle_t {
  omp_null_allocator      = 0,
  omp_default_mem_alloc   = 1,
  omp_large_cap_mem_alloc = 2,
  omp_const_mem_alloc     = 3,
  omp_high_bw_mem_alloc   = 4,
  omp_low_lat_mem_alloc   = 5,
  omp_cgroup_mem_alloc    = 6,
```

```
    omp_pteam_mem_alloc      = 7,
    omp_thread_mem_alloc     = 8
} omp_allocator_handle_t;
```

C / C++

Fortran

```
integer (kind=omp_allocator_handle_kind), &
   parameter :: omp_null_allocator = 0
integer (kind=omp_allocator_handle_kind), &
   parameter :: omp_default_mem_alloc = 1
integer (kind=omp_allocator_handle_kind), &
   parameter :: omp_large_cap_mem_alloc = 2
integer (kind=omp_allocator_handle_kind), &
   parameter :: omp_const_mem_alloc = 3
integer (kind=omp_allocator_handle_kind), &
   parameter :: omp_high_bw_mem_alloc = 4
integer (kind=omp_allocator_handle_kind), &
   parameter :: omp_low_lat_mem_alloc = 5
integer (kind=omp_allocator_handle_kind), &
   parameter :: omp_cgroup_mem_alloc = 6
integer (kind=omp_allocator_handle_kind), &
   parameter :: omp_pteam_mem_alloc = 7
integer (kind=omp_allocator_handle_kind), &
   parameter :: omp_thread_mem_alloc = 8
```

Fortran

The **allocator_handle** OpenMP type represents an allocator as described in Table 8.3. This OpenMP type must be an implementation defined (for C++ possibly scoped) enum type and its valid constants must include those shown above.

20.8.2 OpenMP alloctrait Type

Name: alloctrait **Properties:** omp	**Base Type:** structure

Fields

Name	Type	Properties
key	alloctrait_key	omp
value	alloctrait_val	omp

Type Definition

C / C++

```
typedef struct omp_alloctrait_t {
  omp_alloctrait_key_t key;
  omp_alloctrait_val_t value;
} omp_alloctrait_t;
```

C / C++

Fortran

```
! omp_alloctrait might not be provided
! in deprecated include file omp_lib.h
type omp_alloctrait
   integer (kind=omp_alloctrait_key_kind) key
   integer (kind=omp_alloctrait_val_kind) value
end type omp_alloctrait;
```

Fortran

TABLE 20.4: Allowed Key-Values for **alloctrait** OpenMP Type

Trait	Key	Allowed Values
sync_hint	**omp_atk_sync_hint**	**omp_atv_contended**, **omp_atv_uncontended**, **omp_atv_serialized**, **omp_atv_private**
alignment	**omp_atk_alignment**	Positive property integer powers of 2
access	**omp_atk_access**	**omp_atv_all**, **omp_atv_memspace**, **omp_atv_device**, **omp_atv_cgroup**, **omp_atv_pteam**, **omp_atv_thread**
pool_size	**omp_atk_pool_size**	Any positive property integer
fallback	**omp_atk_fallback**	**omp_atv_default_mem_fb**, **omp_atv_null_fb**, **omp_atv_abort_fb**, **omp_atv_allocator_fb**

table continued on next page

table continued from previous page

Trait	Key	Allowed Values
fb_data	**omp_atk_fb_data**	An allocator handle
pinned	**omp_atk_pinned**	**omp_atv_true**, **omp_atv_false**
partition	**omp_atk_partition**	**omp_atv_environment**, **omp_atv_nearest**, **omp_atv_blocked**, **omp_atv_interleaved**, **omp_atv_partitioner**
pin_device	**omp_atk_pin_device**	Any conforming device number
preferred_device	**omp_atk_preferred_device**	Any conforming device number
target_access	**omp_atk_target_access**	**omp_atv_single**, **omp_atv_multiple**
atomic_scope	**omp_atk_atomic_scope**	**omp_atv_all**, **omp_atv_device**
part_size	**omp_atk_part_size**	Any positive property integer value
partitioner	**omp_atk_partitioner**	A memory partitioner handle
partitioner_arg	**omp_atk_partitioner_arg**	Any integer value

The **alloctrait** OpenMP type is a key-value pair that represents the name of an allocator trait, as the key, and its value (see Table 20.4).

Cross References

- Memory Allocators, see Section 8.2

20.8.3 OpenMP `alloctrait_key` Type

Name: **alloctrait_key** **Properties:** omp	**Base Type:** enumeration

Values

Name	Value	Properties
omp_atk_sync_hint	1	omp
omp_atk_alignment	2	omp
omp_atk_access	3	omp
omp_atk_pool_size	4	omp
omp_atk_fallback	5	omp
omp_atk_fb_data	6	omp
omp_atk_pinned	7	omp
omp_atk_partition	8	omp
omp_atk_pin_device	9	omp
omp_atk_preferred_device	10	omp
omp_atk_device_access	11	omp
omp_atk_target_access	12	omp
omp_atk_atomic_scope	13	omp
omp_atk_part_size	14	omp
omp_atk_partitioner	15	omp
omp_atk_partitioner_arg	16	omp

Type Definition

C / C++

```
typedef enum omp_alloctrait_key_t {
  omp_atk_sync_hint        = 1,
  omp_atk_alignment        = 2,
  omp_atk_access           = 3,
  omp_atk_pool_size        = 4,
  omp_atk_fallback         = 5,
  omp_atk_fb_data          = 6,
  omp_atk_pinned           = 7,
  omp_atk_partition        = 8,
  omp_atk_pin_device       = 9,
  omp_atk_preferred_device = 10,
  omp_atk_device_access    = 11,
  omp_atk_target_access    = 12,
  omp_atk_atomic_scope     = 13,
  omp_atk_part_size        = 14,
  omp_atk_partitioner      = 15,
  omp_atk_partitioner_arg  = 16
} omp_alloctrait_key_t;
```

C / C++

Fortran

```fortran
integer (kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_sync_hint = 1
integer (kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_alignment = 2
integer (kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_access = 3
integer (kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_pool_size = 4
integer (kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_fallback = 5
integer (kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_fb_data = 6
integer (kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_pinned = 7
integer (kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_partition = 8
integer (kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_pin_device = 9
integer (kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_preferred_device = 10
integer (kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_device_access = 11
integer (kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_target_access = 12
integer (kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_atomic_scope = 13
integer (kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_part_size = 14
integer (kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_partitioner = 15
integer (kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_partitioner_arg = 16
```

Fortran

The **alloctrait_key** OpenMP type represents an allocator trait as described in Table 20.4. The valid constants for this OpenMP type must include those shown above.

C++

The **omp.h** header file also defines a class template that models the memory allocator concept in the **omp::allocator** namespace for each value of the **alloctrait_key** OpenMP type. The names in this class do not include either the **omp_** prefix or the **_alloc** suffix.

C++

Cross References

- Memory Allocators, see Section 8.2

20.8.4 OpenMP alloctrait_value Type

Name: alloctrait_value	Base Type: enumeration
Properties: omp	

Values

Name	Value	Properties
omp_atv_default	-1	omp
omp_atv_false	0	omp
omp_atv_true	1	omp
omp_atv_contended	3	omp
omp_atv_uncontended	4	omp
omp_atv_serialized	5	omp
omp_atv_private	6	omp
omp_atv_device	7	omp
omp_atv_thread	8	omp
omp_atv_pteam	9	omp
omp_atv_cgroup	10	omp
omp_atv_default_mem_fb	11	omp
omp_atv_null_fb	12	omp
omp_atv_abort_fb	13	omp
omp_atv_allocator_fb	14	omp
omp_atv_environment	15	omp
omp_atv_nearest	16	omp
omp_atv_blocked	17	omp
omp_atv_interleaved	18	omp
omp_atv_all	19	omp
omp_atv_single	20	omp
omp_atv_multiple	21	omp
omp_atv_memspace	22	omp
omp_atv_partitioner	23	omp

Type Definition

C / C++

```
typedef enum omp_alloctrait_value_t {
  omp_atv_default         = -1,
  omp_atv_false           = 0,
  omp_atv_true            = 1,
  omp_atv_contended       = 3,
```

```
   omp_atv_uncontended    = 4,
   omp_atv_serialized     = 5,
   omp_atv_private        = 6,
   omp_atv_device         = 7,
   omp_atv_thread         = 8,
   omp_atv_pteam          = 9,
   omp_atv_cgroup         = 10,
   omp_atv_default_mem_fb = 11,
   omp_atv_null_fb        = 12,
   omp_atv_abort_fb       = 13,
   omp_atv_allocator_fb   = 14,
   omp_atv_environment    = 15,
   omp_atv_nearest        = 16,
   omp_atv_blocked        = 17,
   omp_atv_interleaved    = 18,
   omp_atv_all            = 19,
   omp_atv_single         = 20,
   omp_atv_multiple       = 21,
   omp_atv_memspace       = 22,
   omp_atv_partitioner    = 23
} omp_alloctrait_value_t;
```

C / C++

Fortran

```
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_default = -1
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_false = 0
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_true = 1
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_contended = 3
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_uncontended = 4
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_serialized = 5
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_private = 6
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_device = 7
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_thread = 8
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_pteam = 9
```

```
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_cgroup = 10
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_default_mem_fb = 11
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_null_fb = 12
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_abort_fb = 13
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_allocator_fb = 14
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_environment = 15
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_nearest = 16
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_blocked = 17
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_interleaved = 18
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_all = 19
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_single = 20
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_multiple = 21
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_memspace = 22
integer (kind=omp_alloctrait_value_kind), &
   parameter :: omp_atv_partitioner = 23
```

Fortran

The **`alloctrait_value`** OpenMP type represents semantic values of allocator traits as described in Table 20.4. The valid constants for this OpenMP type must include those shown above.

Cross References

- Memory Allocators, see Section 8.2

20.8.5 OpenMP `alloctrait_val` Type

Name: `alloctrait_val` **Properties:** omp	**Base Type:** `intptr`

Type Definition

C / C++

```
typedef omp_intptr_t omp_alloctrait_val_t;
```

C / C++

Fortran

```
integer (kind=c_intptr_t)
```

Fortran

The **alloctrait_val** OpenMP type represents the values that may be assigned to the **value** field of the **alloctrait_val** OpenMP type. Any of the semantic values of the **alloctrait_value** OpenMP type may be used for the **alloctrait_val** OpenMP type; in addition, other numeric values may be used for it as appropriate for the specified **key** of the **alloctrait** OpenMP type.

20.8.6 OpenMP mempartition Type

Name: mempartition Properties: named-handle, omp, opaque	Base Type: opaque

Type Definition

C / C++

```
typedef <implementation-defined> omp_mempartition_t;
```

C / C++

Fortran

```
integer (kind=omp_mempartition_kind)
```

Fortran

The **mempartition** OpenMP type is an opaque type that represents memory partitions.

20.8.7 OpenMP mempartitioner Type

Name: mempartitioner Properties: named-handle, omp, opaque	Base Type: opaque

Type Definition

C / C++

```
typedef <implementation-defined> omp_mempartitioner_t;
```

C / C++

Fortran

```
integer (kind=omp_mempartitioner_kind)
```

Fortran

The **mempartitioner** OpenMP type is an opaque type that represents memory partitioners.

20.8.8 OpenMP `mempartitioner_lifetime` Type

Name: `mempartitioner_lifetime` **Properties:** omp	**Base Type:** enumeration

Values

Name	Value	Properties
`omp_static_mempartition`	1	omp
`omp_allocator_mempartition`	2	omp
`omp_dynamic_mempartition`	3	omp

Type Definition

C / C++

```c
typedef enum omp_mempartitioner_lifetime_t {
  omp_static_mempartition    = 1,
  omp_allocator_mempartition = 2,
  omp_dynamic_mempartition   = 3
} omp_mempartitioner_lifetime_t;
```

C / C++

Fortran

```fortran
integer (kind=omp_mempartitioner_lifetime_kind), &
   parameter :: omp_static_mempartition = 1
integer (kind=omp_mempartitioner_lifetime_kind), &
   parameter :: omp_allocator_mempartition = 2
integer (kind=omp_mempartitioner_lifetime_kind), &
   parameter :: omp_dynamic_mempartition = 3
```

Fortran

The **`mempartitioner_lifetime`** OpenMP type represents the lifetime of a memory partitioner. The valid constants for the **`mempartitioner_lifetime`** OpenMP type must include those shown above.

20.8.9 OpenMP `mempartitioner_compute_proc` Type

Name: `mempartitioner_compute_proc` **Category:** subroutine pointer	**Properties:** iso_c_binding, omp

Arguments

Name	Type	Properties
memspace	memspace_handle	omp
allocation_size	c_size_t	iso_c, value
partitioner_arg	alloctrait_val	omp, value
partition	mempartition	C/C++ pointer, omp

Type Signature

C / C++

```
typedef void (*omp_mempartitioner_compute_proc_t) (
  omp_memspace_handle_t memspace, size_t allocation_size,
  omp_alloctrait_val_t partitioner_arg,
  omp_mempartition_t *partition);
```

C / C++

Fortran

```
abstract interface
  subroutine omp_mempartitioner_compute_proc_t(memspace, &
      allocation_size, partitioner_arg, partition) bind(c)
    use, intrinsic :: iso_c_binding, only : c_size_t
    integer (kind=omp_memspace_handle_kind) memspace
    integer (kind=c_size_t), value :: allocation_size
    integer (kind=omp_alloctrait_val_kind), value :: &
        partitioner_arg
    integer (kind=omp_mempartition_kind) partition
  end subroutine
end interface
```

Fortran

The **mempartitioner_compute_proc** OpenMP type represents a partition computation procedure. When used through the **omp_init_mempartition** and **omp_mempartition_set_part** routines, the procedure will be passed the following arguments in the listed order:

- The memory space associated with the allocator to be used for the memory allocation;
- The size of the allocation in bytes;
- If the **omp_atk_partitioner_arg** trait was specified for the allocator, its specified value, otherwise, the value zero; and
- A memory partition object to be initialized

If the sum of the sizes of the parts specified in the memory partition object after executing the procedure is not equal to the *allocation_size* argument, the behavior is unspecified.

If the associated memory partitioner has been created with a call to **omp_init_mempartitioner** with the value of the *lifetime* argument set to **omp_static_mempartition** then the memory partition object computed by an invocation to the procedure might be used for the allocations of any allocators that have the *partitioner* memory partitioner object associated with them if the allocations have the same size and the same memory space. The number of times that the *compute_proc* procedure is invoked is unspecified.

Cross References

- OpenMP **alloctrait_val** Type, see Section 20.8.5
- OpenMP **mempartition** Type, see Section 20.8.6
- OpenMP **memspace_handle** Type, see Section 20.8.11
- **omp_init_mempartition** Routine, see Section 27.5.3
- **omp_mempartition_set_part** Routine, see Section 27.5.5

20.8.10 OpenMP mempartitioner_release_proc Type

Name: **mempartitioner_release_proc** **Category:** subroutine pointer	**Properties:** iso_c_binding, omp

Arguments

Name	Type	Properties
partition	mempartition	C/C++ pointer, omp

Type Signature

C / C++

```
typedef void (*omp_mempartitioner_release_proc_t) (
  omp_mempartition_t *partition);
```

C / C++

Fortran

```
abstract interface
  subroutine omp_mempartitioner_release_proc_t(partition) &
      bind(c)
    integer (kind=omp_mempartition_kind) partition
  end subroutine
end interface
```

Fortran

The **mempartitioner_release_proc** OpenMP type represents a partition release procedure. When an implementation finishes using a memory partition object that was created with the procedure used as the *compute_proc* argument for a call to the **omp_init_mempartitioner** routine to which the represented release procedure was the *release_proc* argument, that release procedure will be called with the memory partition object as its argument. The procedure can then release the object and its resources using the **omp_destroy_mempartition** routine. The implementation will invoke the *release_proc* at most once for each memory partition object.

Cross References

- OpenMP **mempartition** Type, see Section 20.8.6
- **omp_init_mempartitioner** Routine, see Section 27.5.1

20.8.11 OpenMP memspace_handle Type

Name: memspace_handle **Properties:** omp	**Base Type:** enumeration

Values

Name	Value	Properties
omp_null_mem_space	0	omp
omp_default_mem_space	1	omp
omp_large_cap_mem_space	2	omp
omp_const_mem_space	3	omp
omp_high_bw_mem_space	4	omp
omp_low_lat_mem_space	5	omp

Type Definition

C / C++

```c
typedef enum omp_memspace_handle_t {
  omp_null_mem_space       = 0,
  omp_default_mem_space    = 1,
  omp_large_cap_mem_space = 2,
  omp_const_mem_space      = 3,
  omp_high_bw_mem_space    = 4,
  omp_low_lat_mem_space    = 5
} omp_memspace_handle_t;
```

C / C++

Fortran

```fortran
integer (kind=omp_memspace_handle_kind), &
   parameter :: omp_null_mem_space = 0
integer (kind=omp_memspace_handle_kind), &
   parameter :: omp_default_mem_space = 1
integer (kind=omp_memspace_handle_kind), &
   parameter :: omp_large_cap_mem_space = 2
integer (kind=omp_memspace_handle_kind), &
   parameter :: omp_const_mem_space = 3
integer (kind=omp_memspace_handle_kind), &
   parameter :: omp_high_bw_mem_space = 4
integer (kind=omp_memspace_handle_kind), &
   parameter :: omp_low_lat_mem_space = 5
```

Fortran

The **memspace_handle** OpenMP type represents an allocator as described in Table 8.1. This OpenMP type must be an implementation defined (for C++ possibly scoped) enum type and its valid constants must include those shown above.

Fortran

20.9 OpenMP Synchronization Types

This section describes OpenMP types related to synchronization, including locks.

20.9.1 OpenMP depend Type

Name: `depend` **Properties:** named-handle, omp, opaque	**Base Type:** `implementation-defined-int`

Type Definition

C / C++

```
typedef <implementation-defined-integral> omp_depend_t;
```

C / C++

Fortran

```
integer (kind=omp_depend_kind)
```

Fortran

The **depend** OpenMP type is an opaque type that represents depend objects.

20.9.2 OpenMP impex Type

Name: `impex` **Properties:** omp	**Base Type:** enumeration

Values

Name	Value	Properties
`omp_not_impex`	0	omp
`omp_import`	1	omp
`omp_export`	2	omp
`omp_impex`	3	omp

Type Definition

C / C++

```
typedef enum omp_impex_t {
  omp_not_impex = 0,
  omp_import    = 1,
  omp_export    = 2,
  omp_impex     = 3
} omp_impex_t;
```

C / C++

Fortran

```
integer (kind=omp_impex_kind), &
   parameter :: omp_not_impex = 0
integer (kind=omp_impex_kind), &
   parameter :: omp_import = 1
integer (kind=omp_impex_kind), &
   parameter :: omp_export = 2
integer (kind=omp_impex_kind), &
   parameter :: omp_impex = 3
```

Fortran

The **impex** OpenMP type is an enumeration type that is used to specify whether the child tasks of a task may form a task dependence with respect to its dependence-compatible tasks. In particular, it is used to identify whether a task is an importing task and/or an exporting task. The valid constants must include those shown above.

Cross References

- **transparent** Clause, see Section 17.9.6

20.9.3 OpenMP lock Type

Name: lock **Properties:** named-handle, opaque	**Base Type:** opaque

Type Definition

C / C++

```
typedef <implementation-defined> omp_lock_t;
```

C / C++

Fortran

```
integer (kind=omp_lock_kind)
```

Fortran

The **lock** OpenMP type is an opaque type that represents simple locks used in simple lock routines.

20.9.4 OpenMP `nest_lock` Type

Name: `nest_lock` Properties: named-handle, opaque	Base Type: `opaque`

Type Definition

C / C++

```
typedef <implementation-defined> omp_nest_lock_t;
```

C / C++

Fortran

```
integer (kind=omp_nest_lock_kind)
```

Fortran

The **`nest_lock`** OpenMP type is an opaque type that represents nestable locks used in nestable lock routines.

20.9.5 OpenMP `sync_hint` Type

Name: `sync_hint` Properties: omp	Base Type: enumeration

Values

Name	Value	Properties
`omp_sync_hint_none`	`0x0`	omp
`omp_sync_hint_uncontended`	`0x1`	omp
`omp_sync_hint_contended`	`0x2`	omp
`omp_sync_hint_nonspeculative`	`0x4`	omp
`omp_sync_hint_speculative`	`0x8`	omp

Type Definition

C / C++

```
typedef enum omp_sync_hint_t {
  omp_sync_hint_none           = 0x0,
  omp_sync_hint_uncontended    = 0x1,
  omp_sync_hint_contended      = 0x2,
  omp_sync_hint_nonspeculative = 0x4,
  omp_sync_hint_speculative    = 0x8
} omp_sync_hint_t;
```

C / C++

Fortran

```
integer (kind=omp_sync_hint_kind), &
   parameter :: omp_sync_hint_none = &
    int(Z'0', kind=omp_sync_hint_kind)
integer (kind=omp_sync_hint_kind), &
   parameter :: omp_sync_hint_uncontended = &
    int(Z'1', kind=omp_sync_hint_kind)
integer (kind=omp_sync_hint_kind), &
   parameter :: omp_sync_hint_contended = &
    int(Z'2', kind=omp_sync_hint_kind)
integer (kind=omp_sync_hint_kind), &
   parameter :: omp_sync_hint_nonspeculative = &
    int(Z'4', kind=omp_sync_hint_kind)
integer (kind=omp_sync_hint_kind), &
   parameter :: omp_sync_hint_speculative = &
    int(Z'8', kind=omp_sync_hint_kind)
```

Fortran

The **sync_hint** OpenMP type is used to specify synchronization hints. The **omp_init_lock_with_hint** and **omp_init_nest_lock_with_hint** routines provide hints about the expected dynamic behavior or suggested implementation of a lock. Synchronization hints may also be provided for **atomic** and **critical** directives by using the **hint** clause. The effect of a hint does not change the semantics of the associated construct or routine; if ignoring the hint changes the program semantics, the result is unspecified.

Synchronization hints can be combined by using the **+** or **|** operators in C/C++ or the **+** operator in Fortran. Combining **omp_sync_hint_none** with any other synchronization hint is equivalent to specifying the other synchronization hint.

The intended meaning of each synchronization hint is:

- **omp_sync_hint_uncontended**: low contention is expected in this operation, that is, few threads are expected to perform the operation simultaneously in a manner that requires synchronization;
- **omp_sync_hint_contended**: high contention is expected in this operation, that is, many threads are expected to perform the operation simultaneously in a manner that requires synchronization;
- **omp_sync_hint_speculative**: the programmer suggests that the operation should be implemented using speculative techniques such as transactional memory; and
- **omp_sync_hint_nonspeculative**: the programmer suggests that the operation should not be implemented using speculative techniques such as transactional memory.

Note – Future OpenMP specifications may add additional synchronization hints to the **sync_hint** OpenMP type. Implementers are advised to add implementation defined synchronization hints starting from the most significant bit of the type and to include the name of the implementation in the name of the added synchronization hint to avoid name conflicts with other OpenMP implementations.

Restrictions

Restrictions to the synchronization hints are as follows:

- The **omp_sync_hint_uncontended** and **omp_sync_hint_contended** values may not be combined.
- The **omp_sync_hint_nonspeculative** and **omp_sync_hint_speculative** values may not be combined.

Cross References

- **atomic** Construct, see Section 17.8.5
- **critical** Construct, see Section 17.2
- **hint** Clause, see Section 17.1
- **omp_init_lock_with_hint** Routine, see Section 28.1.3
- **omp_init_nest_lock_with_hint** Routine, see Section 28.1.4

20.10 OpenMP Affinity Support Types

This section describes OpenMP types that support affinity mechanisms.

20.10.1 OpenMP `proc_bind` Type

Name: **proc_bind** **Properties:** omp	**Base Type:** enumeration

Values

Name	Value	Properties
omp_proc_bind_false	**0**	omp
omp_proc_bind_true	**1**	omp
omp_proc_bind_primary	**2**	omp
omp_proc_bind_close	**3**	omp
omp_proc_bind_spread	**4**	omp

Type Definition

C / C++

```
typedef enum omp_proc_bind_t {
  omp_proc_bind_false   = 0,
  omp_proc_bind_true    = 1,
  omp_proc_bind_primary = 2,
  omp_proc_bind_close   = 3,
  omp_proc_bind_spread  = 4
} omp_proc_bind_t;
```

C / C++

Fortran

```
integer (kind=omp_proc_bind_kind), &
   parameter :: omp_proc_bind_false = 0
integer (kind=omp_proc_bind_kind), &
   parameter :: omp_proc_bind_true = 1
integer (kind=omp_proc_bind_kind), &
   parameter :: omp_proc_bind_primary = 2
integer (kind=omp_proc_bind_kind), &
   parameter :: omp_proc_bind_close = 3
integer (kind=omp_proc_bind_kind), &
   parameter :: omp_proc_bind_spread = 4
```

Fortran

The **proc_bind** OpenMP type is used in routines that modify or retrieve the value of the *bind-var* ICV. The valid constants for the **proc_bind** type must include those shown above.

Cross References

- *bind-var* ICV, see Table 3.1

20.11 OpenMP Resource Relinquishing Types

This section describes OpenMP types related to resource-relinquishing routines.

20.11.1 OpenMP pause_resource Type

Name: pause_resource **Properties:** omp	**Base Type:** enumeration

Values

Name	Value	Properties
omp_pause_soft	**1**	omp
omp_pause_hard	**2**	omp
omp_pause_stop_tool	**3**	omp

Type Definition

C / C++

```c
typedef enum omp_pause_resource_t {
  omp_pause_soft      = 1,
  omp_pause_hard      = 2,
  omp_pause_stop_tool = 3
} omp_pause_resource_t;
```

C / C++

Fortran

```fortran
integer (kind=omp_pause_resource_kind), &
   parameter :: omp_pause_soft = 1
integer (kind=omp_pause_resource_kind), &
   parameter :: omp_pause_hard = 2
integer (kind=omp_pause_resource_kind), &
   parameter :: omp_pause_stop_tool = 3
```

Fortran

The **pause_resource** OpenMP type is used in resource-relinquishing routines to specify the resources that the instance of the routine relinquishes. The valid constants for the **pause_resource** OpenMP type must include those shown above.

When specified and successful, the **omp_pause_hard** value results in a hard pause, which implies that the OpenMP state is not guaranteed to persist across the resource-relinquishing routine call. A hard pause may relinquish any data allocated by OpenMP on specified devices, including data allocated by device memory routines as well as data present on the devices as a result of a declare target directive or map-entering constructs. A hard pause may also relinquish any data associated with a **threadprivate** directive. When relinquished and when applicable, base language appropriate deallocation/finalization is performed. When relinquished and when applicable, mapped variables on a device will not be copied back from the device to the host device.

When specified and successful, the **omp_pause_soft** value results in a soft pause for which the OpenMP state is guaranteed to persist across the resource-relinquishing routine call, with the exception of any data associated with a **threadprivate** directive, which may be relinquished across the call. When relinquished and when applicable, base language appropriate deallocation/finalization is performed.

Note – A hard pause may relinquish more resources, but may resume processing regions more slowly. A soft pause allows regions to restart more quickly, but may relinquish fewer resources. An OpenMP implementation will reclaim resources as needed for regions encountered after the resource-relinquishing routine region. Since a hard pause may unmap data on the specified devices, appropriate mapping operations are required before using data on the specified devices after the resource-relinquishing routine region.

When specified and successful, the **omp_pause_stop_tool** value implies the effects described above for the **omp_pause_hard** value. Additionally, unless otherwise specified, the value implies that the implementation will shutdown the OMPT interface as if program execution is ending.

20.12 OpenMP Tool Types

This section describes OpenMP types that support the use of tools.

20.12.1 OpenMP control_tool Type

Name: control_tool **Properties:** omp	**Base Type:** enumeration

Values

Name	Value	Properties
omp_control_tool_start	1	omp
omp_control_tool_pause	2	omp
omp_control_tool_flush	3	omp
omp_control_tool_end	4	omp

Type Definition

C / C++

```
typedef enum omp_control_tool_t {
  omp_control_tool_start = 1,
  omp_control_tool_pause = 2,
  omp_control_tool_flush = 3,
  omp_control_tool_end   = 4
} omp_control_tool_t;
```

C / C++

Fortran

```
integer (kind=omp_control_tool_kind), &
   parameter :: omp_control_tool_start = 1
integer (kind=omp_control_tool_kind), &
   parameter :: omp_control_tool_pause = 2
integer (kind=omp_control_tool_kind), &
   parameter :: omp_control_tool_flush = 3
integer (kind=omp_control_tool_kind), &
   parameter :: omp_control_tool_end = 4
```

Fortran

The **control_tool** OpenMP type is used in tool support routines to specify tool commands. Table 20.5 describes the actions that standard commands request from a tool. The valid constants for the **control_tool** OpenMP type must include those shown above.

Tool-defined values for the **control_tool** OpenMP type must be greater than or equal to 64 and less than or equal to 2147483647 (**INT32_MAX**). Tools must ignore **control_tool** values that they are not explicitly designed to handle. Other values accepted by a tool for the **control_tool** OpenMP type are tool defined.

TABLE 20.5: Standard Tool Control Commands

Command	Action
omp_control_tool_start	Start or restart monitoring if it is off. If monitoring is already on, this command is idempotent. If monitoring has already been turned off permanently, this command will have no effect.
omp_control_tool_pause	Temporarily turn monitoring off. If monitoring is already off, it is idempotent.
omp_control_tool_flush	Flush any data buffered by a tool. This command may be applied whether monitoring is on or off.
omp_control_tool_end	Turn monitoring off permanently; the tool finalizes itself and flushes all output.

20.12.2 OpenMP control_tool_result Type

Name: control_tool_result **Properties:** omp	**Base Type:** enumeration

Values

Name	Value	Properties
omp_control_tool_notool	-2	omp
omp_control_tool_nocallback	-1	omp
omp_control_tool_success	0	omp
omp_control_tool_ignored	1	omp

Type Definition

C / C++

```
typedef enum omp_control_tool_result_t {
  omp_control_tool_notool     = -2,
  omp_control_tool_nocallback = -1,
  omp_control_tool_success    = 0,
  omp_control_tool_ignored    = 1
} omp_control_tool_result_t;
```

C / C++

Fortran

```fortran
integer (kind=omp_control_tool_result_kind), &
   parameter :: omp_control_tool_notool = -2
integer (kind=omp_control_tool_result_kind), &
   parameter :: omp_control_tool_nocallback = -1
integer (kind=omp_control_tool_result_kind), &
   parameter :: omp_control_tool_success = 0
integer (kind=omp_control_tool_result_kind), &
   parameter :: omp_control_tool_ignored = 1
```

Fortran

The **control_tool_result** OpenMP type is used in tool support routines to specify the results of tool commands. The valid constants for the **control_tool_result** OpenMP type must include those shown above.

21 Parallel Region Support Routines

This chapter describes routines that support execution of parallel regions, including routines to determine the number of OpenMP threads for parallel regions and that query the nesting of parallel regions at runtime.

21.1 `omp_set_num_threads` Routine

Name: `omp_set_num_threads` **Category:** subroutine	**Properties:** ICV-modifying

Arguments

Name	Type	Properties
num_threads	integer	positive

Prototypes

C / C++

```c
void omp_set_num_threads(int num_threads);
```

C / C++

Fortran

```fortran
subroutine omp_set_num_threads(num_threads)
  integer num_threads
```

Fortran

Effect

The effect of this routine is to set the value of the first element of the *nthreads-var* ICV of the current task to the value specified in the argument. Thus, the routine has the ICV modifying property, through which it affects the number of threads to be used for subsequent **`parallel`** regions that do not specify a **`num_threads`** clause.

Cross References

- *nthreads-var* ICV, see Table 3.1
- **`num_threads`** Clause, see Section 12.1.2
- **`parallel`** Construct, see Section 12.1
- Determining the Number of Threads for a **`parallel`** Region, see Section 12.1.1

21.2 omp_get_num_threads Routine

Name: **omp_get_num_threads** **Category:** function	**Properties:** *default*

Return Type

Name	Type	Properties
<*return type*>	integer	*default*

Prototypes

C / C++

```
int omp_get_num_threads(void);
```

C / C++

Fortran

```
integer function omp_get_num_threads()
```

Fortran

Effect

The **omp_get_num_threads** routine returns the number of threads in the team that is executing the parallel region to which the routine region binds.

21.3 omp_get_thread_num Routine

Name: **omp_get_thread_num** **Category:** function	**Properties:** *default*

Return Type

Name	Type	Properties
<*return type*>	integer	*default*

Prototypes

C / C++

```
int omp_get_thread_num(void);
```

C / C++

Fortran

```
integer function omp_get_thread_num()
```

Fortran

Effect

The **omp_get_thread_num** routine returns the thread number of the calling thread, within the team that is executing the parallel region to which the routine region binds. For assigned threads, the thread number is an integer between 0 and one less than the value returned by **omp_get_num_threads**, inclusive. The thread number of the primary thread of the team is 0. For unassigned threads, the thread number is the value **omp_unassigned_thread**.

Cross References

- Predefined Identifiers, see Section 20.1
- **omp_get_num_threads** Routine, see Section 21.2

21.4 omp_get_max_threads Routine

Name: omp_get_max_threads Category: function	Properties: ICV-retrieving

Return Type

Name	Type	Properties
<return type>	integer	*default*

Prototypes

C / C++

```
int omp_get_max_threads(void);
```

C / C++

Fortran

```
integer function omp_get_max_threads()
```

Fortran

Effect

The value returned by **omp_get_max_threads** is the value of the first element of the *nthreads-var* ICV of the current task; thus, the routine has the ICV retrieving property. Its return value is an upper bound on the number of threads that could be used to form a new team if a parallel region without a **num_threads** clause is encountered after execution returns from this routine.

Cross References

- *nthreads-var* ICV, see Table 3.1
- **num_threads** Clause, see Section 12.1.2
- **parallel** Construct, see Section 12.1
- Determining the Number of Threads for a **parallel** Region, see Section 12.1.1

21.5 omp_get_thread_limit Routine

Name: omp_get_thread_limit Category: function	Properties: ICV-retrieving

Return Type

Name	Type	Properties
<return type>	integer	*default*

Prototypes

C / C++

```
int omp_get_thread_limit(void);
```

C / C++

Fortran

```
integer function omp_get_thread_limit()
```

Fortran

Effect

The **omp_get_thread_limit** routine returns the value of the *thread-limit-var* ICV. Thus, it returns the maximum number of threads available to execute tasks in the current contention group.

Cross References

- *thread-limit-var* ICV, see Table 3.1

21.6 omp_in_parallel Routine

Name: omp_in_parallel **Category:** function	**Properties:** *default*

Return Type

Name	Type	Properties
<return type>	logical	*default*

Prototypes

C / C++

```
int omp_in_parallel(void);
```

C / C++

Fortran

```
logical function omp_in_parallel()
```

Fortran

Effect

The effect of the **omp_in_parallel** routine is to return *true* if the current task is enclosed by an active parallel region, and the **parallel** region is enclosed by the outermost initial task region on the device. That is, it returns *true* if the *active-levels-var* ICV is greater than zero. Otherwise, it returns *false*.

Cross References

- *active-levels-var* ICV, see Table 3.1
- **parallel** Construct, see Section 12.1

21.7 omp_set_dynamic Routine

Name: **omp_set_dynamic** Category: subroutine	**Properties:** ICV-modifying

Arguments

Name	Type	Properties
dynamic_threads	logical	*default*

Prototypes

C / C++

```
void omp_set_dynamic(int dynamic_threads);
```

C / C++

Fortran

```
subroutine omp_set_dynamic(dynamic_threads)
  logical dynamic_threads
```

Fortran

Effect

For implementations that support dynamic adjustment of the number of threads, if the argument to **omp_set_dynamic** evaluates to *true*, dynamic adjustment is enabled for the current task by setting the value of the *dyn-var* ICV to *true*; otherwise, dynamic adjustment is disabled for the current task by setting the value of the *dyn-var* ICV to *false*. For implementations that do not support dynamic adjustment of the number of threads, this routine has no effect: the value of *dyn-var* remains *false*.

Cross References

- *dyn-var* ICV, see Table 3.1

21.8 omp_get_dynamic Routine

Name: **omp_get_dynamic** Category: function	**Properties:** ICV-retrieving

Return Type

Name	Type	Properties
<return type>	logical	*default*

Prototypes

C / C++

```
int omp_get_dynamic(void);
```

C / C++

Fortran

```
logical function omp_get_dynamic()
```

Fortran

Effect

The **omp_get_dynamic** routine returns the value of the *dyn-var* ICV. Thus, this routine returns *true* if dynamic adjustment of the number of threads is enabled for the current task; otherwise, it returns *false*. If an implementation does not support dynamic adjustment of the number of threads, then this routine always returns *false*.

Cross References

- *dyn-var* ICV, see Table 3.1

21.9 omp_set_schedule Routine

Name: omp_set_schedule **Category:** subroutine	**Properties:** ICV-modifying

Arguments

Name	Type	Properties
kind	sched	omp
chunk_size	integer	*default*

Prototypes

C / C++

```
void omp_set_schedule(omp_sched_t kind, int chunk_size);
```

C / C++

Fortran

```
subroutine omp_set_schedule(kind, chunk_size)
  integer (kind=omp_sched_kind) kind
  integer chunk_size
```

Fortran

Effect

The effect of this routine is to set the value of the *run-sched-var* ICV of the current task to the values specified in the two arguments. Thus, the routine affects the schedule that is applied when **runtime** is used as the schedule type.

The schedule is set to the schedule type that is specified by the first argument *kind*. For the schedule types **omp_sched_static**, **omp_sched_dynamic**, and **omp_sched_guided**, the *chunk_size* is set to the value of the second argument, or to the default *chunk_size* if the value of the second argument is less than 1; for the schedule type **omp_sched_auto**, the second argument is ignored; for implementation defined schedule types, the values and associated meanings of the second argument are implementation defined.

Cross References

- *run-sched-var* ICV, see Table 3.1
- OpenMP **sched** Type, see Section 20.5.1

21.10 omp_get_schedule Routine

Name: omp_get_schedule Category: subroutine	Properties: ICV-retrieving

Arguments

Name	Type	Properties
kind	sched	C/C++ pointer, omp
chunk_size	integer	C/C++ pointer

Prototypes

C / C++

```c
void omp_get_schedule(omp_sched_t *kind, int *chunk_size);
```

C / C++

Fortran

```fortran
subroutine omp_get_schedule(kind, chunk_size)
  integer (kind=omp_sched_kind) kind
  integer chunk_size
```

Fortran

Effect

The **omp_get_schedule** routine returns the *run-sched-var* ICV in the task to which the routine binds. Thus, the routine returns the schedule that is applied when the **runtime** schedule type is used. The first argument *kind* returns the schedule type to be used. If the returned schedule type is **omp_sched_static**, **omp_sched_dynamic**, or **omp_sched_guided**, the second argument, *chunk_size*, returns the chunk size to be used, or a value less than 1 if the default chunk size is to be used. The value returned by the second argument is implementation defined for any other schedule types.

Cross References

- *run-sched-var* ICV, see Table 3.1
- OpenMP **sched** Type, see Section 20.5.1

21.11 omp_get_supported_active_levels Routine

Name: omp_get_supported_active_levels Category: function	Properties: *default*

Return Type

Name	Type	Properties
<return type>	integer	*default*

Prototypes

C / C++

```c
int omp_get_supported_active_levels(void);
```

C / C++

Fortran

```fortran
integer function omp_get_supported_active_levels()
```

Fortran

Effect

The **omp_get_supported_active_levels** routine returns the number of supported active levels. The *max-active-levels-var* ICV cannot have a value that is greater than this number. The value that the **omp_get_supported_active_levels** routine returns is implementation defined, but it must be greater than 0.

Cross References

- *max-active-levels-var* ICV, see Table 3.1

21.12 omp_set_max_active_levels Routine

Name: omp_set_max_active_levels Category: subroutine	Properties: ICV-modifying

Arguments

Name	Type	Properties
max_levels	integer	non-negative

Prototypes

C / C++

```
void omp_set_max_active_levels(int max_levels);
```

C / C++

Fortran

```
subroutine omp_set_max_active_levels(max_levels)
  integer max_levels
```

Fortran

Effect

The effect of this routine is to set the value of the *max-active-levels-var* ICV to the value specified in the argument. Thus, the routine limits the number of nested active parallel regions when a new nested **parallel** region is generated by the current task.

If the number of active levels requested exceeds the number of supported active levels, the value of the *max-active-levels-var* ICV will be set to the number of supported active levels. If the number of active levels requested is less than the value of the *active-levels-var* ICV, the value of the *max-active-levels-var* ICV will be set to an implementation defined value between the requested number and *active-levels-var*, inclusive.

Cross References

- *active-levels-var* ICV, see Table 3.1
- *max-active-levels-var* ICV, see Table 3.1
- **parallel** Construct, see Section 12.1

21.13 omp_get_max_active_levels Routine

Name: **omp_get_max_active_levels** **Category:** function	**Properties:** ICV-retrieving

Return Type

Name	Type	Properties
<return type>	integer	*default*

Prototypes

C / C++

```
int omp_get_max_active_levels(void);
```

C / C++

Fortran

```
integer function omp_get_max_active_levels()
```

Fortran

Effect

The **omp_get_max_active_levels** routine returns the value of the *max-active-levels-var* ICV. The current task may only generate an active parallel region if the returned value is greater than the value of the *active-levels-var* ICV.

Cross References

- *max-active-levels-var* ICV, see Table 3.1

21.14 omp_get_level Routine

Name: omp_get_level **Category:** function	**Properties:** ICV-retrieving

Return Type

Name	Type	Properties
<return type>	integer	*default*

Prototypes

C / C++

```
int omp_get_level(void);
```

C / C++

Fortran

```
integer function omp_get_level()
```

Fortran

Effect

The **omp_get_level** routine returns the value of the *levels-var* ICV. Thus, its effect is to return the number of nested **parallel** regions (whether active or inactive) that enclose the current task such that all of the **parallel** regions are enclosed by the outermost initial task region on the current device.

Cross References

- *levels-var* ICV, see Table 3.1
- **parallel** Construct, see Section 12.1

21.15 omp_get_ancestor_thread_num Routine

Name: omp_get_ancestor_thread_num **Category:** function	**Properties:** *default*

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
level	integer	*default*

Prototypes

C / C++

```
int omp_get_ancestor_thread_num(int level);
```

C / C++

Fortran

```
integer function omp_get_ancestor_thread_num(level)
  integer level
```

Fortran

Effect

The **omp_get_ancestor_thread_num** routine returns the thread number of the ancestor thread at a given nest level of the encountering thread or the thread number of the encountering thread. If the requested nest level is outside the range of 0 and the nest level of the encountering thread, as returned by the **omp_get_level** routine, the routine returns -1.

Note – When the **omp_get_ancestor_thread_num** routine is called with value of *level* =0, the routine always returns 0. If *level* =**omp_get_level**(), the routine has the same effect as the **omp_get_thread_num** routine.

Cross References

- **omp_get_level** Routine, see Section 21.14
- **omp_get_thread_num** Routine, see Section 21.3

21.16 omp_get_team_size Routine

Name: omp_get_team_size Category: function	Properties: *default*

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
level	integer	*default*

Prototypes

C / C++

```
int omp_get_team_size(int level);
```

C / C++

Fortran

```
integer function omp_get_team_size(level)
  integer level
```

Fortran

Effect

The **omp_get_team_size** routine returns the size of the current team to which the ancestor thread or the encountering task belongs. If the requested nested level is outside the range of 0 and the nested level of the encountering thread, as returned by the **omp_get_level** routine, the routine returns -1. Inactive parallel regions are regarded as active parallel regions executed with one thread.

Note – When the **omp_get_team_size** routine is called with a value of *level* =0, the routine always returns 1. If *level* =**omp_get_level**(), the routine has the same effect as the **omp_get_num_threads** routine.

Cross References

- **omp_get_level** Routine, see Section 21.14
- **omp_get_num_threads** Routine, see Section 21.2

21.17 omp_get_active_level Routine

Name: omp_get_active_level **Category:** function	**Properties:** ICV-retrieving

Return Type

Name	Type	Properties
<return type>	integer	*default*

Prototypes

C / C++

```
int omp_get_active_level(void);
```

C / C++

Fortran

```
integer function omp_get_active_level()
```

Fortran

Effect

The effect of the **`omp_get_active_level`** routine is to return the number of nested active **`parallel`** regions that enclose the current task such that all **`parallel`** regions are enclosed by the outermost initial task region on the current device. Thus, the routine returns the value of the *active-levels-var* ICV.

Cross References

- *active-levels-var* ICV, see Table 3.1
- **`parallel`** Construct, see Section 12.1

22 Teams Region Routines

This chapter describes routines that affect and monitor the league of teams that may execute a **teams** region.

22.1 omp_get_num_teams Routine

Name: `omp_get_num_teams` **Category:** function	**Properties:** ICV-retrieving, teams-nestable

Return Type

Name	Type	Properties
<return type>	integer	*default*

Prototypes

C / C++

```
int omp_get_num_teams(void);
```

C / C++

Fortran

```
integer function omp_get_num_teams()
```

Fortran

Effect

The **omp_get_num_teams** routine returns the value of the *league-size-var* ICV, which is the number of initial teams in the current **teams** region. The routine returns 1 if it is called from outside of a **teams** region.

Cross References

- *league-size-var* ICV, see Table 3.1
- **teams** Construct, see Section 12.2

22.2 omp_set_num_teams Routine

Name: omp_set_num_teams **Category:** subroutine	**Properties:** ICV-modifying

Arguments

Name	Type	Properties
num_teams	integer	non-negative

Prototypes

C / C++

```
void omp_set_num_teams(int num_teams);
```

C / C++

Fortran

```
subroutine omp_set_num_teams(num_teams)
  integer num_teams
```

Fortran

Effect

The effect of the **omp_set_num_teams** routine is to set the value of the *nteams-var* ICV of the host device to the value specified in the *num_teams* argument.

Restrictions

Restrictions to the **omp_set_num_teams** routine are as follows:

- An **omp_set_num_teams** region must be a strictly nested region of the implicit parallel region that surrounds the whole OpenMP program.

Cross References

- *nteams-var* ICV, see Table 3.1
- **num_teams** Clause, see Section 12.2.1
- **teams** Construct, see Section 12.2

22.3 omp_get_team_num Routine

Name: omp_get_team_num **Category:** function	**Properties:** ICV-retrieving, teams-nestable

Return Type

Name	Type	Properties
<return type>	integer	*default*

Prototypes

C / C++

```
int omp_get_team_num(void);
```

C / C++

Fortran

```
integer function omp_get_team_num()
```

Fortran

Effect

The **omp_get_team_num** routine returns the value of the *team-num-var* ICV, which is the team number of the current team and is an integer between 0 and one less than the value returned by **omp_get_num_teams**, inclusive. The routine returns 0 if it is called outside of a **teams** region.

Cross References

- *team-num-var* ICV, see Table 3.1
- **omp_get_num_teams** Routine, see Section 22.1
- **teams** Construct, see Section 12.2

22.4 omp_get_max_teams Routine

Name: omp_get_max_teams Category: function	Properties: ICV-retrieving

Return Type

Name	Type	Properties
<*return type*>	integer	*default*

Prototypes

C / C++

```
int omp_get_max_teams(void);
```

C / C++

Fortran

```
integer function omp_get_max_teams()
```

Fortran

Effect

The **omp_get_max_teams** routine returns the value of the *nteams-var* ICV of the current device. If positive, this value is also an upper bound on the number of teams that can be created by a **teams** construct without a **num_teams** clause that is encountered after execution returns from this routine.

Cross References

- *nteams-var* ICV, see Table 3.1
- **num_teams** Clause, see Section 12.2.1
- **teams** Construct, see Section 12.2

22.5 omp_get_teams_thread_limit Routine

Name: **omp_get_teams_thread_limit** **Category:** function	**Properties:** ICV-retrieving

Return Type

Name	Type	Properties
<return type>	integer	*default*

Prototypes

C / C++

```
int omp_get_teams_thread_limit(void);
```

C / C++

Fortran

```
integer function omp_get_teams_thread_limit()
```

Fortran

Effect

The **omp_get_teams_thread_limit** routine returns the value of the *teams-thread-limit-var* ICV, which is the maximum number of threads available to execute tasks in each contention group that a **teams** construct creates.

Cross References

- *teams-thread-limit-var* ICV, see Table 3.1
- **teams** Construct, see Section 12.2

22.6 omp_set_teams_thread_limit Routine

Name: **omp_set_teams_thread_limit** **Category:** subroutine	**Properties:** ICV-modifying

Arguments

Name	Type	Properties
thread_limit	integer	positive

Prototypes

C / C++

```
void omp_set_teams_thread_limit(int thread_limit);
```

C / C++

Fortran

```
subroutine omp_set_teams_thread_limit(thread_limit)
  integer thread_limit
```

Fortran

Effect

The **omp_set_teams_thread_limit** routine sets the value of the *teams-thread-limit-var* ICV to the value of the *thread_limit* argument and thus defines the maximum number of threads that can execute tasks in each contention group that a **teams** construct creates on the host device. If the value of *thread_limit* exceeds the number of threads that an implementation supports for each contention group created by a **teams** construct, the value of the *teams-thread-limit-var* ICV will be set to the number that is supported by the implementation.

Restrictions

Restrictions to the **omp_set_teams_thread_limit** routine are as follows:

- An **omp_set_num_teams** region must be a strictly nested region of the implicit parallel region that surrounds the whole OpenMP program.

Cross References

- *teams-thread-limit-var* ICV, see Table 3.1
- **teams** Construct, see Section 12.2
- **thread_limit** Clause, see Section 15.3

23 Tasking Support Routines

This chapter specifies OpenMP API routines that support task execution:

- Tasking routines that query general task execution properties; and
- The event routine to fulfill task dependences.

23.1 Tasking Routines

This section describes routines that pertain to OpenMP explicit tasks.

23.1.1 `omp_get_max_task_priority` Routine

Name: `omp_get_max_task_priority` **Category:** function	**Properties:** all-device-threads-binding, ICV-retrieving

Return Type

Name	Type	Properties
<*return type*>	integer	*default*

Prototypes

C / C++

```
int omp_get_max_task_priority(void);
```

C / C++

Fortran

```
integer function omp_get_max_task_priority()
```

Fortran

Effect

The **`omp_get_max_task_priority`** routine returns the value of the *max-task-priority-var* ICV, which determines the maximum value that can be specified in the **`priority`** clause.

Cross References

- *max-task-priority-var* ICV, see Table 3.1
- **`priority`** Clause, see Section 14.9

23.1.2 `omp_in_explicit_task` Routine

Name: **`omp_in_explicit_task`** **Category:** function	**Properties:** ICV-retrieving

Return Type

Name	Type	Properties
<return type>	logical	*default*

Prototypes

C / C++

```
int omp_in_explicit_task(void);
```

C / C++

Fortran

```
logical function omp_in_explicit_task()
```

Fortran

Effect

The **`omp_in_explicit_task`** routine returns the value of the *explicit-task-var* ICV, which indicates whether the encountering task is an explicit task region.

Cross References

- *explicit-task-var* ICV, see Table 3.1
- **`task`** Construct, see Section 14.1

23.1.3 `omp_in_final` Routine

Name: **`omp_in_final`** **Category:** function	**Properties:** ICV-retrieving

Return Type

Name	Type	Properties
<return type>	logical	*default*

Prototypes

C / C++

```
int omp_in_final(void);
```

C / C++

Fortran

```
logical function omp_in_final()
```

Fortran

Effect

The **omp_in_final** routine returns the value of the *final-task-var* ICV, which indicates whether the encountering task is a final task region.

Cross References

- **final** Clause, see Section 14.7
- *final-task-var* ICV, see Table 3.1
- **task** Construct, see Section 14.1

23.1.4 omp_is_free_agent Routine

Name: omp_is_free_agent Category: function	Properties: ICV-retrieving

Return Type

Name	Type	Properties
<return type>	logical	*default*

Prototypes

C / C++

```
int omp_is_free_agent(void);
```

C / C++

Fortran

```
logical function omp_is_free_agent()
```

Fortran

Effect

The **omp_is_free_agent** routine returns the value of the *free-agent-var* ICV, which indicates whether a free-agent thread is executing the enclosing task region at the time the routine is called.

Cross References

- *free-agent-var* ICV, see Table 3.1
- **task** Construct, see Section 14.1
- **threadset** Clause, see Section 14.8

23.1.5 omp_ancestor_is_free_agent Routine

Name: omp_ancestor_is_free_agent Category: function	Properties: *default*

Return Type and Arguments

Name	Type	Properties
<return type>	logical	*default*
level	integer	*default*

Prototypes

C / C++

```
int omp_ancestor_is_free_agent(int level);
```

C / C++

Fortran

```
logical function omp_ancestor_is_free_agent(level)
  integer level
```

Fortran

Effect

The **omp_ancestor_is_free_agent** routine returns *true* if the ancestor thread of the encountering thread is a free-agent thread, for a given nested level of the encountering thread; otherwise, it returns *false*. If the requested nesting level is outside the range of 0 and the nesting level of the current task, as returned by the **omp_get_level** routine, the routine returns *false*.

Note – When the **omp_ancestor_is_free_agent** routine is called with a value of *level* =**omp_get_level**, the routine has the same effect as the **omp_is_free_agent** routine.

Cross References

- **omp_get_level** Routine, see Section 21.14
- **omp_is_free_agent** Routine, see Section 23.1.4
- **task** Construct, see Section 14.1
- **threadset** Clause, see Section 14.8

23.2 Event Routine

This section describes routines that support OpenMP event objects.

23.2.1 omp_fulfill_event Routine

Name: **omp_fulfill_event** **Category:** subroutine	**Properties:** *default*

Arguments

Name	Type	Properties
event	event_handle	*default*

Prototypes

C / C++

```
void omp_fulfill_event(omp_event_handle_t event);
```

C / C++

Fortran

```
subroutine omp_fulfill_event(event)
  integer (kind=omp_event_handle_kind) event
```

Fortran

Effect

The effect of this routine is to fulfill the event associated with the *event* argument. The effect of fulfilling the event will depend on how the event object was created. The event object is destroyed and cannot be accessed after calling this routine, and the event handle becomes unassociated with any event object. This routine has no effect if the *event* argument corresponds to a completed task.

Execution Model Events

The *task-fulfill* event occurs in a thread that executes an **omp_fulfill_event** region before the event is fulfilled if the OpenMP event object was created by a **detach** clause on a task.

Tool Callbacks

A thread dispatches a registered **task_schedule** callback with NULL as its *next_task_data* argument while the argument *prior_task_data* binds to the detachable task for each occurrence of a *task-fulfill* event. If the *task-fulfill* event occurs before the detachable task finished execution of the associated structured block, the callback has **ompt_task_early_fulfill** as its *prior_task_status* argument; otherwise the callback has **ompt_task_late_fulfill** as its *prior_task_status* argument.

Restrictions

Restrictions to the **omp_fulfill_event** routine are as follows:

- The event that corresponds to the *event* argument must not have already been fulfilled.
- The event handle that the *event* argument identifies must have been created by the effect of a **detach** clause.
- The event handle passed to the routine must refer to an event object that was created by a thread in the same device as the thread that invoked the routine.
- An event handle must be fulfilled before execution continues beyond the next barrier of the current team after a **detach** clause creates the event that the *event* argument represents.

Cross References

- **detach** Clause, see Section 14.11
- OpenMP **event_handle** Type, see Section 20.6.1
- **task_schedule** Callback, see Section 34.5.2
- OMPT **task_status** Type, see Section 33.38

24 Device Information Routines

This chapter describes device-information routines, which are routines that have the device-information property. These routines modify or retrieve information that supports the use of the set of devices that are available to an OpenMP program.

Restrictions

Restrictions to device-information routines are as follows.

- Any *device_num* argument must be a conforming device number unless otherwise specified.

24.1 `omp_set_default_device` Routine

Name: **`omp_set_default_device`** **Category:** subroutine	**Properties:** device-information, ICV-modifying

Arguments

Name	Type	Properties
device_num	integer	*default*

Prototypes

C / C++

```c
void omp_set_default_device(int device_num);
```

C / C++

Fortran

```fortran
subroutine omp_set_default_device(device_num)
  integer device_num
```

Fortran

Effect

The effect of the **`omp_set_default_device`** routine is to set the value of the *default-device-var* ICV of the current task to the value specified in the *device-num* argument, thus determining the default target device. When called from within a **`target`** region, the effect of this routine is unspecified.

Cross References

- *default-device-var* ICV, see Table 3.1
- **`target`** Construct, see Section 15.8

24.2 `omp_get_default_device` Routine

Name: `omp_get_default_device` **Category:** function	**Properties:** device-information, ICV-retrieving

Return Type

Name	Type	Properties
<return type>	integer	*default*

Prototypes

C / C++

```
int omp_get_default_device(void);
```

C / C++

Fortran

```
integer function omp_get_default_device()
```

Fortran

Effect

The **`omp_get_default_device`** routine returns the value of the *default-device-var* ICV of the current task, which is the device number of the default target device. When called from within a **`target`** region the effect of this routine is unspecified.

Cross References

- *default-device-var* ICV, see Table 3.1
- **`target`** Construct, see Section 15.8

24.3 `omp_get_num_devices` Routine

Name: `omp_get_num_devices` **Category:** function	**Properties:** device-information, ICV-retrieving

Return Type

Name	Type	Properties
<return type>	integer	*default*

Prototypes

C / C++

```
int omp_get_num_devices(void);
```

C / C++

Fortran

```
integer function omp_get_num_devices()
```

Fortran

Effect

The **omp_get_num_devices** routine returns the value of the *num-devices-var* ICV, which is the number of available non-host devices onto which code or data may be offloaded. When called from within a **target** region the effect of this routine is unspecified.

Cross References

- *num-devices-var* ICV, see Table 3.1
- **target** Construct, see Section 15.8

24.4 omp_get_device_num Routine

Name: omp_get_device_num **Category:** function	**Properties:** device-information

Return Type

Name	Type	Properties
<return type>	integer	*default*

Prototypes

C / C++

```c
int omp_get_device_num(void);
```

Fortran

```fortran
integer function omp_get_device_num()
```

Effect

The **omp_get_device_num** routine returns the value of the *device-num-var* ICV, which is the device number of the device on which the encountering thread is executing. When called on the host device, it will return the same value as the **omp_get_initial_device** routine.

Cross References

- *device-num-var* ICV, see Table 3.1
- **target** Construct, see Section 15.8

24.5 omp_get_num_procs Routine

Name: omp_get_num_procs **Category:** function	**Properties:** all-device-threads-binding, device-information, ICV-retrieving

Return Type

Name	Type	Properties
<return type>	integer	*default*

Prototypes

C / C++

```
int omp_get_num_procs(void);
```

C / C++

Fortran

```
integer function omp_get_num_procs()
```

Fortran

Effect

The **omp_get_num_procs** routine returns the value of the *num-procs-var* ICV. Thus, this routine returns the number of processors that are available to the device at the time the routine is called. This value may change between the time that it is determined by the **omp_get_num_procs** routine and the time that it is read in the calling context due to system actions outside the control of the OpenMP implementation.

Cross References

- *num-procs-var* ICV, see Table 3.1

24.6 omp_get_max_progress_width Routine

Name: omp_get_max_progress_width Category: function	Properties: device-information

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
device_num	integer	*default*

Prototypes

C / C++

```
int omp_get_max_progress_width(int device_num);
```

C / C++

Fortran

```
integer function omp_get_max_progress_width(device_num)
  integer device_num
```

Fortran

Effect

The **omp_get_max_progress_width** routine returns the maximum size, in terms of hardware threads, of progress units on the device specified by *device_num*. When called from within a **target** region the effect of this routine is unspecified.

24.7 omp_get_device_from_uid Routine

Name: omp_get_device_from_uid Category: function	Properties: device-information

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
uid	char	pointer, intent(in)

Prototypes

C / C++

```
int omp_get_device_from_uid(const char *uid);
```

C / C++

Fortran

```
integer function omp_get_device_from_uid(uid)
  character(len=*), intent(in) :: uid
```

Fortran

Effect

The **omp_get_device_from_uid** routine returns the device number associated with the device specified by the *uid*; if no device with that *uid* is available, the value of **omp_invalid_device** is returned. When called from within a **target** region, the effect is unspecified.

Cross References

- *available-devices-var* ICV, see Table 3.1
- *default-device-var* ICV, see Table 3.1
- **omp_get_uid_from_device** Routine, see Section 24.8

24.8 omp_get_uid_from_device Routine

Name: omp_get_uid_from_device Category: function	Properties: device-information

Return Type and Arguments

Name	Type	Properties
<return type>	const char	pointer
device_num	integer	intent(in)

Prototypes

C / C++

```
const char *omp_get_uid_from_device(int device_num);
```

C / C++

Fortran

```
character(:) function omp_get_uid_from_device(device_num)
  pointer :: omp_get_uid_from_device
  integer, intent(in) :: device_num
```

Fortran

Effect

The **omp_get_uid_from_device** routine returns the implementation defined unique identifier string that identifies the device specified by *device_num*. If the *device_num* argument has a value of **omp_invalid_device**, the routine returns NULL. When called from within a **target** region, the effect is unspecified.

Cross References

- *available-devices-var* ICV, see Table 3.1
- *default-device-var* ICV, see Table 3.1
- **omp_get_device_from_uid** Routine, see Section 24.7

24.9 omp_is_initial_device Routine

Name: **omp_is_initial_device**	**Properties:** device-information
Category: function	

Return Type

Name	Type	Properties
<return type>	logical	*default*

Prototypes

C / C++

```
int omp_is_initial_device(void);
```

C / C++

Fortran

```
logical function omp_is_initial_device()
```

Fortran

Effect

The **omp_is_initial_device** routine returns *true* if the current task is executing on the host device; otherwise, it returns *false*.

24.10 omp_get_initial_device Routine

Name: omp_get_initial_device Category: function	Properties: device-information

Return Type

Name	Type	Properties
<return type>	integer	*default*

Prototypes

C / C++

```
int omp_get_initial_device(void);
```

C / C++

Fortran

```
integer function omp_get_initial_device()
```

Fortran

Effect

The effect of the **omp_get_initial_device** routine is to return the device number of the host device. The value of the device number is the value of **omp_initial_device** or the value returned by the **omp_get_num_devices** routine. When called from within a **target** region the effect of this routine is unspecified.

Cross References

- **target** Construct, see Section 15.8

24.11 omp_get_device_num_teams Routine

Name: omp_get_device_num_teams Category: function	Properties: device-information, ICV-retrieving

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
device_num	integer	*default*

Prototypes

C / C++

```c
int omp_get_device_num_teams(int device_num);
```

C / C++

Fortran

```fortran
integer function omp_get_device_num_teams(device_num)
  integer device_num
```

Fortran

Effect

The **omp_get_device_num_teams** routine returns the value of the *nteams-var* ICV in the device data environment of device *device_num*. Thus, the routine returns the number of teams that will be requested for a **teams** region on device *device_num* if the **num_teams** clause is not specified. If *device_num* is the device number of the host device, **omp_get_device_num_teams** is equivalent to **omp_get_num_teams**. If the *device_num* argument has the value of **omp_invalid_device** or is not a conforming device number, the routine returns zero. When called from within a **target** region, the effect of this routine is unspecified.

Cross References

- *nteams-var* ICV, see Table 3.1
- **num_teams** Clause, see Section 12.2.1
- **teams** Construct, see Section 12.2

24.12 omp_set_device_num_teams Routine

Name: omp_set_device_num_teams Category: subroutine	Properties: device-information, ICV-modifying

Arguments

Name	Type	Properties
num_teams	integer	non-negative
device_num	integer	*default*

Prototypes

C / C++

```c
void omp_set_device_num_teams(int num_teams, int device_num);
```

C / C++

Fortran

```fortran
subroutine omp_set_device_num_teams(num_teams, device_num)
  integer num_teams, device_num
```

Fortran

Effect

The effect of the **omp_set_device_num_teams** routine is to set the value of the *nteams-var* ICV of device *device_num* to the value specified in the *num_teams* argument. Thus, the routine determines the number of teams that will be requested for a **teams** region on device *device_num* if the **num_teams** clause is not specified. If *device_num* is the device number of the host device, **omp_set_device_num_teams** is equivalent to **omp_set_num_teams**. If the *device_num* argument has the value of **omp_invalid_device** or is not a conforming device number, runtime error termination occurs. When called from within a **target** region, the effect of this routine is unspecified.

Restrictions

Restrictions to the **omp_set_device_num_teams** routine are as follows:

- The routine must not execute concurrently with any device-affecting construct on device *device_num*.
- If device *device_num* is the host device, an **omp_set_device_num_teams** region must be a strictly nested region of the implicit parallel region that surrounds the whole OpenMP program.

Cross References

- *nteams-var* ICV, see Table 3.1
- **num_teams** Clause, see Section 12.2.1
- **teams** Construct, see Section 12.2

24.13 omp_get_device_teams_thread_limit Routine

Name: omp_get_device_teams_thread_limit **Category:** function	**Properties:** device-information, ICV-retrieving

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
device_num	integer	*default*

Prototypes

C / C++

```
int omp_get_device_teams_thread_limit(int device_num);
```

C / C++

Fortran

```
integer function omp_get_device_teams_thread_limit(device_num)
  integer device_num
```

Fortran

Effect

The **omp_get_device_teams_thread_limit** routine returns the value of the *teams-thread-limit-var* ICV in the device data environment of device *device_num*, which is the maximum number of threads available to execute tasks in each contention group that a **teams** construct creates on that device. If *device_num* is the device number of the host device, **omp_get_device_teams_thread_limit** is equivalent to **omp_get_teams_thread_limit**. If the *device_num* argument has the value of **omp_invalid_device** or is not a conforming device number, the routine returns zero. When called from within a **target** region, the effect of this routine is unspecified.

Cross References

- *teams-thread-limit-var* ICV, see Table 3.1
- **teams** Construct, see Section 12.2

24.14 omp_set_device_teams_thread_limit Routine

Name: omp_set_device_teams_thread_limit **Category:** subroutine	**Properties:** device-information, ICV-modifying

Arguments

Name	Type	Properties
thread_limit	integer	positive
device_num	integer	*default*

Prototypes

C / C++

```
void omp_set_device_teams_thread_limit(int thread_limit,
  int device_num);
```

C / C++

Fortran

```
subroutine omp_set_device_teams_thread_limit(thread_limit, &
  device_num)
  integer thread_limit, device_num
```

Fortran

Effect

The **omp_set_device_teams_thread_limit** routine sets the value of the *teams-thread-limit-var* ICV in the device data environment of device *device_num* to the value of the *thread_limit* argument and thus defines the maximum number of threads that can execute tasks in each contention group that a **teams** construct creates on that device. If the value of *thread_limit* exceeds the number of threads that an implementation supports for each contention group created by a **teams** construct on device *device_num*, the value of the *teams-thread-limit-var* ICV will be set to the number that is supported by the implementation. If *device_num* is the device number of the host device, **omp_set_device_teams_thread_limit** is equivalent to **omp_set_teams_thread_limit**. If the *device_num* argument has the value of **omp_invalid_device** or is not a conforming device number, runtime error termination occurs. When called from within a **target** region, the effect of this routine is unspecified.

Restrictions

Restrictions to the **omp_set_device_teams_thread_limit** routine are as follows:

- The routine must not execute concurrently with any device-affecting construct on device *device_num*.
- If device *device_num* is the host device, an **omp_set_device_teams_thread_limit** region must be a strictly nested region of the implicit parallel region that surrounds the whole OpenMP program.

Cross References

- *teams-thread-limit-var* ICV, see Table 3.1
- **teams** Construct, see Section 12.2
- **thread_limit** Clause, see Section 15.3

25 Device Memory Routines

This chapter describes device memory routines that support allocation of memory and management of pointers in the data environments of target devices, and therefore the routines have the device memory routine property.

If the *device_num*, *src_device_num*, or *dst_device_num* argument of a device memory routine has the value **omp_invalid_device**, runtime error termination is performed.

Device memory routines that are not device-memory-information routines execute as if part of a target task that is generated by the call to the routine. This target task, which is an included task if the routine is not an asynchronous device routine, is the generating task of the region associated with the routine. Since the target task provides the execution context for any execution that occurs on the device, it is the binding task set for the routine. Thus, all of these routines have the generating-task binding property.

Fortran

The Fortran version of all device memory routines have ISO C bindings so the routines have the ISO C binding property. Thus, each device memory routine requires an explicit interface and so might not be provided in the deprecated include file **omp_lib.h**.

Fortran

Execution Model Events

Events associated with a target task are the same as for the **task** construct defined in Section 14.1.

Tool Callbacks

Callbacks associated with events for target tasks are the same as for the **task** construct defined in Section 14.1; **(***flags* **& ompt_task_target)** always evaluates to *true* in the dispatched callback.

Restrictions

Restrictions to device memory routines are as follows:

- Any *device_num*, *src_device_num*, and *dst_device_num* arguments must be conforming device numbers.
- When called from within a **target** region, the effect is unspecified.

Cross References

- **target** Construct, see Section 15.8
- **task** Construct, see Section 14.1
- OMPT **task_flag** Type, see Section 33.37

25.1 Asynchronous Device Memory Routines

Some device memory routines have the asynchronous-device routine property. The execution of the target task that is generated by the call to an asynchronous device routines may be deferred. Task dependences are expressed with zero or more OpenMP depend objects. The dependences are specified by passing the number of depend objects followed by an array of the objects. The generated target task is not a dependent task if the program passes in a count of zero for *depobj_count*. The *depobj_list* argument is ignored if the value of *depobj_count* is zero.

Execution Model Events

Events associated with task dependences that result from *depobj_list* are the same as for a **depend** clause with the **depobj** *task-dependence-type* defined in Section 17.9.5.

Tool Callbacks

Callbacks associated with events for task dependences are the same as for the **depend** clause defined in Section 17.9.5.

Cross References

- **depend** Clause, see Section 17.9.5
- **depobj** Construct, see Section 17.9.3

25.2 Device Memory Information Routines

This section describes routines that have the device-memory-information routine property. These device-memory-information routines provide information about device pointers, which can be determined without directly accessing the target device; thus, they do not create a target task.

25.2.1 omp_target_is_present Routine

Name: **omp_target_is_present** **Category:** function	**Properties:** device-memory-information-routine, device-memory-routine, iso_c_binding

Return Type and Arguments

Name	Type	Properties
<return type>	c_int	*default*
ptr	c_ptr	intent(in), iso_c, value
device_num	c_int	iso_c, value

Prototypes

C / C++

```
int omp_target_is_present(const void *ptr, int device_num);
```

C / C++

Fortran

```
integer (kind=c_int) function omp_target_is_present(ptr, &
  device_num) bind(c)
  use, intrinsic :: iso_c_binding, only : c_int, c_ptr
  type (c_ptr), value, intent(in) :: ptr
  integer (kind=c_int), value :: device_num
```

Fortran

Effect

The **omp_target_is_present** routine returns a non-zero value if *device_num* refers to the host device or if *ptr* refers to storage that has corresponding storage in the device data environment of device *device_num*. Otherwise, the routine returns zero. If *ptr* is NULL. the routine returns zero. Thus, the **omp_target_is_present** routine tests whether a host pointer refers to storage that is mapped to a given device.

Restrictions

Restrictions to the **omp_target_is_present** routine are as follows:

- The value of *ptr* must be a valid host pointer or NULL.

25.2.2 omp_target_is_accessible Routine

Name: **omp_target_is_accessible** **Category:** function	**Properties:** device-memory-information-routine, device-memory-routine, iso_c_binding

Return Type and Arguments

Name	Type	Properties
<return type>	c_int	*default*
ptr	c_ptr	intent(in), iso_c, value
size	c_size_t	iso_c, positive, value
device_num	c_int	iso_c, value

Prototypes

C / C++

```
int omp_target_is_accessible(const void *ptr, size_t size,
  int device_num);
```

C / C++

Fortran

```
integer (kind=c_int) function omp_target_is_accessible(ptr, &
  size, device_num) bind(c)
  use, intrinsic :: iso_c_binding, only : c_int, c_ptr, &
    c_size_t
  type (c_ptr), value, intent(in) :: ptr
  integer (kind=c_size_t), value :: size
  integer (kind=c_int), value :: device_num
```

Fortran

Effect

The **omp_target_is_accessible** routine returns a non-zero value if the storage of *size* bytes that corresponds to the address range starting at the address given by *ptr* is accessible from device *device_num*. Otherwise, it returns zero. If *ptr* is NULL, the routine returns zero. The value of *ptr* is interpreted as an address in the address space of the specified device.

25.2.3 omp_get_mapped_ptr Routine

Name: **omp_get_mapped_ptr** Category: function	**Properties:** device-memory-information-routine, device-memory-routine, iso_c_binding

Return Type and Arguments

Name	Type	Properties
<return type>	c_ptr	*default*
ptr	c_ptr	intent(in), iso_c, value
device_num	c_int	iso_c, value

Prototypes

C / C++

```c
void *omp_get_mapped_ptr(const void *ptr, int device_num);
```

C / C++

Fortran

```fortran
type (c_ptr) function omp_get_mapped_ptr(ptr, device_num) &
  bind(c)
  use, intrinsic :: iso_c_binding, only : c_ptr, c_int
  type (c_ptr), value, intent(in) :: ptr
  integer (kind=c_int), value :: device_num
```

Fortran

Effect

The **omp_get_mapped_ptr** routine returns the associated device pointer for host pointer *ptr* on device *device_num*. A call to this routine for a pointer that is not NULL and does not have an associated pointer on the given device will return NULL. The routine returns NULL if unsuccessful. Otherwise it returns the device pointer, which is *ptr* if *device_num* specifies the host device.

Cross References

- **omp_get_initial_device** Routine, see Section 24.10

25.3 omp_target_alloc Routine

Name: **omp_target_alloc** Category: function	**Properties:** device-memory-routine, generating-task-binding, iso_c_binding

Return Type and Arguments

Name	Type	Properties
<return type>	c_ptr	*default*
size	c_size_t	iso_c, value
device_num	c_int	iso_c, value

Prototypes

C / C++

```
void *omp_target_alloc(size_t size, int device_num);
```

C / C++

Fortran

```
type (c_ptr) function omp_target_alloc(size, device_num) &
  bind(c)
  use, intrinsic :: iso_c_binding, only : c_ptr, c_size_t, &
    c_int
  integer (kind=c_size_t), value :: size
  integer (kind=c_int), value :: device_num
```

Fortran

Effect

The **omp_target_alloc** routine returns a device pointer that references the device address of a storage location of *size* bytes. The storage location is dynamically allocated in the device data environment of the device specified by *device_num*.

The **omp_target_alloc** routine returns NULL if it cannot dynamically allocate the memory in the device data environment or if *size* is 0. The device pointer returned by **omp_target_alloc** can be used in an **is_device_ptr** clause (see Section 7.5.7).

Execution Model Events

The *target-data-allocation-begin* event occurs before a thread initiates a data allocation on a target device. The *target-data-allocation-end* event occurs after a thread initiates a data allocation on a target device.

Tool Callbacks

A thread dispatches a registered **target_data_op_emi** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *target-data-allocation-begin* event in that thread. Similarly, a thread dispatches a registered **target_data_op_emi** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of a *target-data-allocation-end* event in that thread.

Restrictions

Restrictions to the **omp_target_alloc** routine are as follows:

- Freeing the storage returned by **omp_target_alloc** with any routine other than **omp_target_free** results in unspecified behavior.

C / C++

- Unless the **unified_address** clause appears on a **requires** directive in the compilation unit, pointer arithmetic is not supported on the device pointer returned by **omp_target_alloc**.

C / C++

Cross References

- **is_device_ptr** Clause, see Section 7.5.7
- **omp_target_free** Routine, see Section 25.4
- OMPT **scope_endpoint** Type, see Section 33.27
- **target_data_op_emi** Callback, see Section 35.7

25.4 omp_target_free Routine

Name: **omp_target_free** **Category:** subroutine	**Properties:** device-memory-routine, generating-task-binding, iso_c_binding

Arguments

Name	Type	Properties
device_ptr	c_ptr	iso_c, value
device_num	c_int	iso_c, value

Prototypes

C / C++

```
void omp_target_free(void *device_ptr, int device_num);
```

C / C++

Fortran

```
subroutine omp_target_free(device_ptr, device_num) bind(c)
  use, intrinsic :: iso_c_binding, only : c_ptr, c_int
  type (c_ptr), value :: device_ptr
  integer (kind=c_int), value :: device_num
```

Fortran

Effect

The **omp_target_free** routine frees the memory in the device data environment associated with *device_ptr*. If *device_ptr* is NULL, the operation is ignored.

Execution Model Events

The *target-data-free-begin* event occurs before a thread initiates a data free on a target device. The *target-data-free-end* event occurs after a thread initiates a data free on a target device.

Tool Callbacks

A thread dispatches a registered **target_data_op_emi** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *target-data-free-begin* event in that thread. Similarly, a thread dispatches a registered **target_data_op_emi** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of a *target-data-free-end* event in that thread.

Restrictions

Restrictions to the **omp_target_free** routine are as follows:

- The value of *device_ptr* must be NULL or have been returned by **omp_target_alloc**.

Cross References

- **omp_target_alloc** Routine, see Section 25.3
- OMPT **scope_endpoint** Type, see Section 33.27
- **target_data_op_emi** Callback, see Section 35.7

25.5 omp_target_associate_ptr Routine

Name: **omp_target_associate_ptr** **Category:** function	**Properties:** device-memory-routine, generating-task-binding, iso_c_binding

Return Type and Arguments

Name	Type	Properties
<return type>	c_int	*default*
host_ptr	c_ptr	intent(in), iso_c, value
device_ptr	c_ptr	intent(in), iso_c, value
size	c_size_t	iso_c, value
device_offset	c_size_t	iso_c, value
device_num	c_int	iso_c, value

Prototypes

C / C++

```
int omp_target_associate_ptr(const void *host_ptr,
  const void *device_ptr, size_t size, size_t device_offset,
  int device_num);
```

C / C++

Fortran

```
integer (kind=c_int) function omp_target_associate_ptr(host_ptr, &
  device_ptr, size, device_offset, device_num) bind(c)
  use, intrinsic :: iso_c_binding, only : c_int, c_ptr, &
    c_size_t
  type (c_ptr), value, intent(in) :: host_ptr, device_ptr
  integer (kind=c_size_t), value :: size, device_offset
  integer (kind=c_int), value :: device_num
```

Fortran

Effect

The **omp_target_associate_ptr** routine associates a device pointer in the device data environment of device *device_num* with a host pointer such that when the host device pointer appears in a subsequent **map** clause, the associated device pointer is used as the target for data motion associated with that host pointer. Thus, the **omp_target_associate_ptr** routine maps a device pointer, which may be returned from **omp_target_alloc** or implementation defined routine, to a host pointer. The *device_offset* argument specifies the offset into *device_ptr* that is used as the base address for the device side of the mapping. The reference count of the resulting mapping will be infinite. The association between the host pointer and the device pointer can be removed by using the **omp_target_disassociate_ptr** routine. The routine returns zero if successful. Otherwise it returns a non-zero value.

Only one device buffer can be associated with a given host pointer value and device number pair. Attempting to associate a second buffer will return non-zero. Associating the same pair of pointers on the same device with the same offset has no effect and returns zero. Associating pointers that share underlying storage will result in unspecified behavior. The **omp_target_is_present** routine can be used to test whether a given host pointer has a corresponding list item in the device data environment.

Execution Model Events

The *target-data-associate* event occurs before a thread initiates a device pointer association on a target device.

Tool Callbacks

A thread dispatches a registered **target_data_op_emi** callback with **ompt_scope_beginend** as its *endpoint* argument for each occurrence of a *target-data-associate* event in that thread.

Cross References

- **omp_target_alloc** Routine, see Section 25.3
- **omp_target_disassociate_ptr** Routine, see Section 25.6
- **omp_target_is_present** Routine, see Section 25.2.1

- OMPT **scope_endpoint** Type, see Section 33.27
- **target_data_op_emi** Callback, see Section 35.7

25.6 omp_target_disassociate_ptr Routine

Name: omp_target_disassociate_ptr **Category:** function	**Properties:** device-memory-routine, generating-task-binding, iso_c_binding

Return Type and Arguments

Name	Type	Properties
<return type>	c_int	*default*
ptr	c_ptr	intent(in), iso_c, value
device_num	c_int	iso_c, value

Prototypes

C / C++

```
int omp_target_disassociate_ptr(const void *ptr, int device_num);
```

C / C++

Fortran

```
integer (kind=c_int) function omp_target_disassociate_ptr(ptr, &
  device_num) bind(c)
  use, intrinsic :: iso_c_binding, only : c_int, c_ptr
  type (c_ptr), value, intent(in) :: ptr
  integer (kind=c_int), value :: device_num
```

Fortran

Effect

The **omp_target_disassociate_ptr** removes the associated device data on device *device_num* from the presence table for host pointer *ptr*. A call to this routine on a pointer that is not NULL and does not have associated data on the given device results in unspecified behavior. The reference count of the mapping is reduced to zero, regardless of its current value. The routine returns zero if successful. Otherwise it returns a non-zero value.

Execution Model Events

The *target-data-disassociate* event occurs before a thread initiates a device pointer disassociation on a target device.

Tool Callbacks

A thread dispatches a registered **target_data_op_emi** callback with **ompt_scope_beginend** as its *endpoint* argument for each occurrence of a *target-data-disassociate* event in that thread.

Cross References

- OMPT **scope_endpoint** Type, see Section 33.27
- **target_data_op_emi** Callback, see Section 35.7

25.7 Memory Copying Routines

This section describes memory-copying routines, which are routines that have the memory-copying property. These routines copy memory from the device data environment of a *src_device_num* device to the device data environment of a *dst_device_num* device. OpenMP provides two varieties of memory-copying routines: flat-memory-copying routines, which have the flat-memory-copying property; and rectangular-memory-copying routines, which have the rectangular-memory-copying property.

Each flat-memory-copying routine copies *length* bytes of memory at offset *src_offset* from *src* in the device data environment of device *src_device_num* to *dst* starting at offset *dst_offset* in the device data environment of device *dst_device_num*.

Each rectangular-memory-copying routine performs a copy between any combination of host pointers and device pointers. Specifically, the routine copies a rectangular subvolume from a multi-dimensional array *src*, in the device data environment of device *src_device_num*, to another multi-dimensional array *dst*, in the device data environment of device *dst_device_num*. The volume is specified in terms of the size of an element, number of dimensions, and constant arrays of length *num_dims*. The maximum number of dimensions supported is at least three; support for higher dimensionality is implementation defined. The *volume* array specifies the length, in number of elements, to copy in each dimension from *src* to *dst*. The *dst_offsets* (*src_offsets*) argument specifies the number of elements from the origin of *dst* (*src*) in elements. The *dst_dimensions* (*src_dimensions*) argument specifies the length of each dimension of *dst* (*src*).

An OpenMP program can determine the inclusive number of dimensions that an implementation supports for a rectangular-memory-copying routine by passing NULL for both *dst* and *src*. The routine returns the number of dimensions supported by the implementation for the specified device numbers. No copy operation is performed.

Fortran

Because the interface of each rectangular-memory-copying routine binds directly to a C language routine, each of these routines assumes C memory ordering.

Fortran

Each memory-copying routine contains a task scheduling point. These routines return zero on success and non-zero on failure.

Execution Model Events

The *target-data-op-begin* event occurs before a thread initiates a data transfer in a memory-copying routine region. The *target-data-op-end* event occurs after a thread initiates a data transfer in a memory-copying routine region.

Tool Callbacks

A thread dispatches a registered **target_data_op_emi** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *target-data-op-begin* event in that thread. Similarly, a thread dispatches a registered **target_data_op_emi** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of a *target-data-op-end* event in that thread. These callbacks occur in the context of the target task.

Restrictions

Restrictions to the memory-copying routines are as follows:

- The value of *src* must be a valid device pointer for the device *src_device_num*.
- The value of *dst* must be a valid device pointer for the device *dst_device_num*.
- The value of *num_dims* must be between 1 and the implementation defined limit, which must be at least three.
- The length of the offset (*src_offset* and *dst_offset*) and dimension (*src_dimensions* and *dst_dimensions*) arrays must be at least the value of *num_dims*.

Cross References

- OMPT **scope_endpoint** Type, see Section 33.27
- **target_data_op_emi** Callback, see Section 35.7

25.7.1 omp_target_memcpy Routine

Name: **omp_target_memcpy** **Category:** function	**Properties:** device-memory-routine, flat-memory-copying, generating-task-binding, iso_c_binding, memory-copying

Return Type and Arguments

Name	Type	Properties
<return type>	c_int	*default*
dst	c_ptr	iso_c, value
src	c_ptr	intent(in), iso_c, value
length	c_size_t	iso_c, value
dst_offset	c_size_t	iso_c, value
src_offset	c_size_t	iso_c, value
dst_device_num	c_int	iso_c, value
src_device_num	c_int	iso_c, value

Prototypes

C / C++

```
int omp_target_memcpy(void *dst, const void *src, size_t length,
  size_t dst_offset, size_t src_offset, int dst_device_num,
  int src_device_num);
```

C / C++

Fortran

```fortran
integer (kind=c_int) function omp_target_memcpy(dst, src, &
  length, dst_offset, src_offset, dst_device_num, &
  src_device_num) bind(c)
  use, intrinsic :: iso_c_binding, only : c_int, c_ptr, &
    c_size_t
  type (c_ptr), value :: dst
  type (c_ptr), value, intent(in) :: src
  integer (kind=c_size_t), value :: length, dst_offset, &
    src_offset
  integer (kind=c_int), value :: dst_device_num, src_device_num
```

Fortran

Effect

As a flat-memory-copying routine, the effect of the **omp_target_memcpy** routine is as described in Section 25.7. This effect includes the associated tool events and callbacks defined in that section.

Cross References

- Memory Copying Routines, see Section 25.7

25.7.2 omp_target_memcpy_rect Routine

Name: omp_target_memcpy_rect **Category:** function	**Properties:** device-memory-routine, generating-task-binding, iso_c_binding, memory-copying, rectangular-memory-copying

Return Type and Arguments

Name	Type	Properties
<return type>	c_int	*default*
dst	c_ptr	iso_c, value
src	c_ptr	intent(in), iso_c, value
element_size	c_size_t	iso_c, value
num_dims	c_int	iso_c, positive, value
volume	c_size_t	intent(in), iso_c, pointer
dst_offsets	c_size_t	intent(in), iso_c, pointer
src_offsets	c_size_t	intent(in), iso_c, pointer
dst_dimensions	c_size_t	intent(in), iso_c, pointer
src_dimensions	c_size_t	intent(in), iso_c, pointer
dst_device_num	c_int	iso_c, value
src_device_num	c_int	iso_c, value

Prototypes

C / C++

```
int omp_target_memcpy_rect(void *dst, const void *src,
  size_t element_size, int num_dims, const size_t *volume,
  const size_t *dst_offsets, const size_t *src_offsets,
  const size_t *dst_dimensions, const size_t *src_dimensions,
  int dst_device_num, int src_device_num);
```

C / C++

Fortran

```
integer (kind=c_int) function omp_target_memcpy_rect(dst, src, &
  element_size, num_dims, volume, dst_offsets, src_offsets, &
  dst_dimensions, src_dimensions, dst_device_num, &
  src_device_num) bind(c)
  use, intrinsic :: iso_c_binding, only : c_int, c_ptr, &
    c_size_t
  type (c_ptr), value :: dst
  type (c_ptr), value, intent(in) :: src
  integer (kind=c_size_t), value :: element_size
  integer (kind=c_int), value :: num_dims, dst_device_num, &
    src_device_num
  integer (kind=c_size_t), intent(in) :: volume(*), dst_offsets&
    (*), src_offsets(*), dst_dimensions(*), src_dimensions(*)
```

Fortran

Effect

As a rectangular-memory-copying routine, the effect of the **omp_target_memcpy_rect** routine is as described in Section 25.7. This effect includes the associated tool events and callbacks defined in that section.

Cross References

- Memory Copying Routines, see Section 25.7

25.7.3 omp_target_memcpy_async Routine

Name: omp_target_memcpy_async Category: function	Properties: asynchronous-device-routine, device-memory-routine, flat-memory-copying, generating-task-binding, iso_c_binding, memory-copying

Return Type and Arguments

Name	Type	Properties
<return type>	c_int	*default*
dst	c_ptr	iso_c, value
src	c_ptr	intent(in), iso_c, value
length	c_size_t	iso_c, value
dst_offset	c_size_t	iso_c, value
src_offset	c_size_t	iso_c, value
dst_device_num	c_int	iso_c, value
src_device_num	c_int	iso_c, value
depobj_count	c_int	iso_c, value
depobj_list	depend	optional, pointer

Prototypes

C / C++

```
int omp_target_memcpy_async(void *dst, const void *src,
  size_t length, size_t dst_offset, size_t src_offset,
  int dst_device_num, int src_device_num, int depobj_count,
  omp_depend_t *depobj_list);
```

C / C++

Fortran

```
integer (kind=c_int) function omp_target_memcpy_async(dst, src, &
  length, dst_offset, src_offset, dst_device_num, &
  src_device_num, depobj_count, depobj_list) bind(c)
  use, intrinsic :: iso_c_binding, only : c_int, c_ptr, &
    c_size_t
  type (c_ptr), value :: dst
  type (c_ptr), value, intent(in) :: src
  integer (kind=c_size_t), value :: length, dst_offset, &
    src_offset
  integer (kind=c_int), value :: dst_device_num, src_device_num, &
    depobj_count
  integer (kind=omp_depend_kind), optional :: depobj_list(*)
```

Fortran

Effect

As a flat-memory-copying routine, the effect of the **omp_target_memcpy_async** routine is as described in Section 25.7. This effect includes the tool events and callbacks defined in that section. As it is also an asynchronous device routine, the routine also includes the tool events and callbacks defined in Section 25.1.

Cross References

- Asynchronous Device Memory Routines, see Section 25.1
- Memory Copying Routines, see Section 25.7

25.7.4 omp_target_memcpy_rect_async Routine

Name: omp_target_memcpy_rect_async Category: function	Properties: asynchronous-device-routine, device-memory-routine, generating-task-binding, iso_c_binding, memory-copying, rectangular-memory-copying

Return Type and Arguments

Name	Type	Properties
<return type>	c_int	*default*
dst	c_ptr	iso_c, value
src	c_ptr	intent(in), iso_c, value
element_size	c_size_t	iso_c, value
num_dims	c_int	iso_c, positive, value
volume	c_size_t	intent(in), iso_c, pointer
dst_offsets	c_size_t	intent(in), iso_c, pointer
src_offsets	c_size_t	intent(in), iso_c, pointer
dst_dimensions	c_size_t	intent(in), iso_c, pointer
src_dimensions	c_size_t	intent(in), iso_c, pointer
dst_device_num	c_int	iso_c, value
src_device_num	c_int	iso_c, value
depobj_count	c_int	iso_c, value
depobj_list	depend	optional, pointer

Prototypes

C / C++

```
int omp_target_memcpy_rect_async(void *dst, const void *src,
  size_t element_size, int num_dims, const size_t *volume,
  const size_t *dst_offsets, const size_t *src_offsets,
  const size_t *dst_dimensions, const size_t *src_dimensions,
  int dst_device_num, int src_device_num, int depobj_count,
  omp_depend_t *depobj_list);
```

C / C++

Fortran

```fortran
integer (kind=c_int) function omp_target_memcpy_rect_async(dst, &
  src, element_size, num_dims, volume, dst_offsets, src_offsets, &
  dst_dimensions, src_dimensions, dst_device_num, &
  src_device_num, depobj_count, depobj_list) bind(c)
  use, intrinsic :: iso_c_binding, only : c_int, c_ptr, &
    c_size_t
  type (c_ptr), value :: dst
  type (c_ptr), value, intent(in) :: src
  integer (kind=c_size_t), value :: element_size
  integer (kind=c_int), value :: num_dims, dst_device_num, &
    src_device_num, depobj_count
  integer (kind=c_size_t), intent(in) :: volume(*), dst_offsets&
    (*), src_offsets(*), dst_dimensions(*), src_dimensions(*)
  integer (kind=omp_depend_kind), optional :: depobj_list(*)
```

Fortran

Effect

As a rectangular-memory-copying routine, the effect of the **omp_target_memcpy_rect_async** routine is as described in Section 25.7. This effect includes the tool events and callbacks defined in that section. As it is also an asynchronous device routine, the routine also includes the tool events and callbacks defined in Section 25.1.

Cross References

- Asynchronous Device Memory Routines, see Section 25.1
- Memory Copying Routines, see Section 25.7

25.8 Memory Setting Routines

This section describes the memory-setting routines, which are routines that have the memory-setting property. These routines fill memory in a device data environment with a given value. The effect of a memory-setting routine is to fill the first *count* bytes pointed to by *ptr* with the value *val* (converted to **unsigned char**) in the device data environment associated with device *device_num*. If *count* is zero, the routine has no effect. If *ptr* is NULL, the effect is unspecified. The memory-setting routines return *ptr*. Each memory-setting routine contains a task scheduling point.

Execution Model Events

The *target-data-op-begin* event occurs before a thread initiates filling the memory in a memory-setting routine region. The *target-data-op-end* event occurs after a thread initiates filling the memory in a memory-setting routine region.

Tool Callbacks

A thread dispatches a registered **target_data_op_emi** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *target-data-op-begin* event in that thread. Similarly, a thread dispatches a registered **target_data_op_emi** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of a *target-data-op-end* event in that thread. These callbacks occur in the context of the target task.

Restrictions

The restrictions to the memory-setting routines are as follows:

- The value of the *ptr* argument must be a valid pointer to device memory for the device denoted by the value of the *device_num* argument.

Cross References

- OMPT **scope_endpoint** Type, see Section 33.27
- **target_data_op_emi** Callback, see Section 35.7

25.8.1 omp_target_memset Routine

Name: **omp_target_memset** **Category:** function	**Properties:** device-memory-routine, generating-task-binding, iso_c_binding, memory-setting

Return Type and Arguments

Name	Type	Properties
<return type>	c_ptr	*default*
ptr	c_ptr	iso_c, value
val	c_int	iso_c, value
count	c_size_t	iso_c, value
device_num	c_int	iso_c, value

Prototypes

C / C++

```
void *omp_target_memset(void *ptr, int val, size_t count,
  int device_num);
```

C / C++

Fortran

```
type (c_ptr) function omp_target_memset(ptr, val, count, &
  device_num) bind(c)
  use, intrinsic :: iso_c_binding, only : c_ptr, c_int, &
    c_size_t
  type (c_ptr), value :: ptr
  integer (kind=c_int), value :: val, device_num
  integer (kind=c_size_t), value :: count
```

Fortran

Effect

As a memory-setting routine, the effect of the **omp_target_memset** routine is as described in Section 25.8. This effect includes the tool events and callbacks defined in that section.

Cross References

- Memory Setting Routines, see Section 25.8

25.8.2 omp_target_memset_async Routine

Name: **omp_target_memset_async** **Category:** function	**Properties:** asynchronous-device-routine, device-memory-routine, generating-task-binding, iso_c_binding, memory-setting

Return Type and Arguments

Name	Type	Properties
<return type>	c_ptr	*default*
ptr	c_ptr	iso_c, value
val	c_int	iso_c, value
count	c_size_t	iso_c, value
device_num	c_int	iso_c, value
depobj_count	c_int	iso_c, value
depobj_list	depend	optional, pointer

Prototypes

C / C++

```
void  *omp_target_memset_async(void *ptr, int val, size_t count,
  int device_num, int depobj_count, omp_depend_t *depobj_list);
```

C / C++

Fortran

```
type (c_ptr) function omp_target_memset_async(ptr, val, count, &
  device_num, depobj_count, depobj_list) bind(c)
  use, intrinsic :: iso_c_binding, only : c_ptr, c_int, &
    c_size_t
  type (c_ptr), value :: ptr
  integer (kind=c_int), value :: val, device_num, depobj_count
  integer (kind=c_size_t), value :: count
  integer (kind=omp_depend_kind), optional :: depobj_list(*)
```

Fortran

Effect

As a memory-setting routine, the effect of the **omp_target_memset_async** routine is as described in Section 25.8. This effect includes the tool events and callbacks defined in that section. As it is also an asynchronous device routine, the routine also includes the tool events and callbacks defined in Section 25.1.

Cross References

- Asynchronous Device Memory Routines, see Section 25.1
- Memory Setting Routines, see Section 25.8

26 Interoperability Routines

This section describes interoperability routines, which have the interoperability-routine property. These routines provide mechanisms to inspect the properties associated with an interoperability object. Each interoperability routine takes an *interop* argument of the **interop** OpenMP type. Most interoperability routines also take a *property_id* argument of the **interop_property** OpenMP type and a *ret_code* argument of (pointer to) **interop_rc** OpenMP type.

Interoperability-property-retrieving routines, which have the interoperability-property-retrieving property, retrieve an interoperability property from an interoperability object. For these routines, if a non-null pointer is passed to the *ret_code* argument, an **interop_rc** OpenMP type value that indicates the return code is stored in the object to which *ret_code* points. If an error occurred, the stored value is negative and matches the error as defined in Table 20.3. On success, **omp_irc_success** is stored. If no error occurred but no meaningful value can be returned, **omp_irc_no_value** is stored.

Interoperability-property-retrieving routines return the requested interoperability property, if available, and zero if an error occurs or no value is available. If the *interop* argument is **omp_interop_none**, an empty error occurs. If the *property_id* argument is greater than or equal to **omp_get_num_interop_properties(***interop***)** or less than **omp_ipr_first**, an out-of-range error occurs. If the requested property value is not convertible into a value of the type that the specific interoperability-property-retrieving routine retrieves, a type error occurs.

Restrictions

Restrictions to interoperability routines are as follows:

- Providing an invalid interoperability object for the *interop* argument results in unspecified behavior.
- For any interoperability routine that returns a pointer, memory referenced by the pointer is managed by the OpenMP implementation and should not be freed or modified and memory referenced by that pointer cannot be accessed after the interoperability object that was used to obtain the pointer is destroyed.

Cross References

- OpenMP Interoperability Support Types, see Section 20.7

26.1 omp_get_num_interop_properties Routine

Name: omp_get_num_interop_properties **Category:** function	**Properties:** interoperability-routine

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
interop	interop	intent(in)

Prototypes

C / C++

```
int omp_get_num_interop_properties(const omp_interop_t interop);
```

C / C++

Fortran

```
integer function omp_get_num_interop_properties(interop)
  integer (kind=omp_interop_kind), intent(in) :: interop
```

Fortran

Effect

The **omp_get_num_interop_properties** routine returns the number of implementation defined interoperability properties available for *interop*. The total number of properties available for *interop* is the returned value minus **omp_ipr_first**.

Cross References

- OpenMP **interop** Type, see Section 20.7.1

26.2 omp_get_interop_int Routine

Name: omp_get_interop_int **Category:** function	**Properties:** interoperability-property-retrieving, interoperability-routine

Return Type and Arguments

Name	Type	Properties
<return type>	intptr	*default*
interop	interop	omp, opaque, intent(in)
property_id	interop_property	omp
ret_code	interop_rc	omp, intent(out), optional

Prototypes

C / C++

```
omp_intptr_t  *omp_get_interop_int(const omp_interop_t interop,
  omp_interop_property_t property_id, omp_interop_rc_t *ret_code);
```

C / C++

Fortran

```
integer (kind=c_intptr_t) function omp_get_interop_int(interop, &
  property_id, ret_code)
  use, intrinsic :: iso_c_binding, only : c_intptr_t
  integer (kind=omp_interop_kind), intent(in) :: interop
  integer (kind=omp_interop_property_kind) property_id
  integer (kind=omp_interop_rc_kind), intent(out), optional :: &
    ret_code
```

Fortran

Effect

The **omp_get_interop_int** routine is an interoperability-property-retrieving routine that retrieves an interoperability property of integer type, if available.

Cross References

- OpenMP **interop** Type, see Section 20.7.1
- OpenMP **interop_property** Type, see Section 20.7.3
- OpenMP **interop_rc** Type, see Section 20.7.4
- **omp_get_num_interop_properties** Routine, see Section 26.1

26.3 omp_get_interop_ptr Routine

Name: omp_get_interop_ptr **Category:** function	**Properties:** interoperability-property-retrieving, interoperability-routine

Return Type and Arguments

Name	Type	Properties
<return type>	c_ptr	*default*
interop	interop	omp, opaque, intent(in)
property_id	interop_property	omp
ret_code	interop_rc	omp, intent(out), optional

Prototypes

C / C++

```
void *omp_get_interop_ptr(const omp_interop_t interop,
  omp_interop_property_t property_id, omp_interop_rc_t *ret_code);
```

C / C++

Fortran

```
type (c_ptr) function omp_get_interop_ptr(interop, property_id, &
  ret_code)
  use, intrinsic :: iso_c_binding, only : c_ptr
  integer (kind=omp_interop_kind), intent(in) :: interop
  integer (kind=omp_interop_property_kind) property_id
  integer (kind=omp_interop_rc_kind), intent(out), optional :: &
    ret_code
```

Fortran

Effect

The **omp_get_interop_ptr** routine is an interoperability-property-retrieving routine that retrieves an interoperability property of pointer type, if available.

Cross References

- OpenMP **interop** Type, see Section 20.7.1
- OpenMP **interop_property** Type, see Section 20.7.3
- OpenMP **interop_rc** Type, see Section 20.7.4
- **omp_get_num_interop_properties** Routine, see Section 26.1

26.4 omp_get_interop_str Routine

Name: **omp_get_interop_str** **Category:** function	**Properties:** interoperability-property-retrieving, interoperability-routine

Return Type and Arguments

Name	Type	Properties
<return type>	const char	pointer
interop	interop	omp, opaque, intent(in)
property_id	interop_property	omp
ret_code	interop_rc	omp, intent(out), optional

Prototypes

C / C++

```
const char *omp_get_interop_str(const omp_interop_t interop,
  omp_interop_property_t property_id, omp_interop_rc_t *ret_code);
```

C / C++

Fortran

```
character(:) function omp_get_interop_str(interop, property_id, &
  ret_code)
  pointer :: omp_get_interop_str
  integer (kind=omp_interop_kind), intent(in) :: interop
  integer (kind=omp_interop_property_kind) property_id
  integer (kind=omp_interop_rc_kind), intent(out), optional :: &
    ret_code
```

Fortran

Effect

The **omp_get_interop_str** routine is an interoperability-property-retrieving routine that retrieves an interoperability string property type as a string, if available.

Cross References

- OpenMP **interop** Type, see Section 20.7.1
- OpenMP **interop_property** Type, see Section 20.7.3
- OpenMP **interop_rc** Type, see Section 20.7.4
- **omp_get_num_interop_properties** Routine, see Section 26.1

26.5 omp_get_interop_name Routine

Name: omp_get_interop_name **Category:** function	**Properties:** interoperability-routine

Return Type and Arguments

Name	Type	Properties
<return type>	const char	pointer
interop	interop	omp, opaque, intent(in)
property_id	interop_property	omp

Prototypes

C / C++

```
const char  *omp_get_interop_name(const omp_interop_t interop,
  omp_interop_property_t property_id);
```

C / C++

Fortran

```
character(:) function omp_get_interop_name(interop,  property_id)
  pointer :: omp_get_interop_name
  integer (kind=omp_interop_kind), intent(in) :: interop
  integer (kind=omp_interop_property_kind) property_id
```

Fortran

Effect

The **omp_get_interop_name** routine returns, as a string, the name of the interoperability property identified by *property_id*. Property names for non-implementation defined interoperability properties are listed in Table 20.2. If the *property_id* is less than **omp_ipr_first** or greater than or equal to **omp_get_num_interop_properties**(*interop*), NULL is returned.

Cross References

- OpenMP **interop** Type, see Section 20.7.1
- OpenMP **interop_property** Type, see Section 20.7.3
- **omp_get_num_interop_properties** Routine, see Section 26.1

26.6 omp_get_interop_type_desc Routine

Name: **omp_get_interop_type_desc** **Category:** function	**Properties:** interoperability-routine

Return Type and Arguments

Name	Type	Properties
<return type>	const char	pointer
interop	interop	omp, opaque, intent(in)
property_id	interop_property	omp

Prototypes

C / C++

```
const char *omp_get_interop_type_desc(
  const omp_interop_t interop, omp_interop_property_t property_id);
```

C / C++

Fortran

```
character(:) function omp_get_interop_type_desc(interop, &
  property_id)
  pointer :: omp_get_interop_type_desc
  integer (kind=omp_interop_kind), intent(in) :: interop
  integer (kind=omp_interop_property_kind) property_id
```

Fortran

Effect

The **omp_get_interop_type_desc** routine returns a string that describes the type of the interoperability property identified by *property_id* in human-readable form. The description may contain a valid type declaration, possibly followed by a description or name of the type. If *interop* has the value **omp_interop_none**, or if the *property_id* is less than **omp_ipr_first** or greater than or equal to **omp_get_num_interop_properties**(*interop*), NULL is returned.

Cross References

- OpenMP **interop** Type, see Section 20.7.1
- OpenMP **interop_property** Type, see Section 20.7.3
- **omp_get_num_interop_properties** Routine, see Section 26.1

26.7 omp_get_interop_rc_desc Routine

Name: omp_get_interop_rc_desc Category: function	Properties: interoperability-routine

Return Type and Arguments

Name	Type	Properties
<return type>	const char	pointer
interop	interop	omp, opaque, intent(in)
ret_code	interop_rc	omp

Prototypes

C / C++

```
const char *omp_get_interop_rc_desc(const omp_interop_t interop,
  omp_interop_rc_t ret_code);
```

C / C++

Fortran

```
character(:) function omp_get_interop_rc_desc(interop, ret_code)
  pointer :: omp_get_interop_rc_desc
  integer (kind=omp_interop_kind), intent(in) :: interop
  integer (kind=omp_interop_rc_kind) ret_code
```

Fortran

Effect

The **omp_get_interop_rc_desc** routine returns a string that describes the return code *ret_code* associated with an interoperability object in human-readable form.

Restrictions

Restrictions to the **omp_get_interop_rc_desc** routine are as follows:

- The behavior of the routine is unspecified if *ret_code* was not last written by an interoperability routine invoked with the interoperability object *interop*.

Cross References

- OpenMP **interop** Type, see Section 20.7.1
- OpenMP **interop_property** Type, see Section 20.7.3
- OpenMP **interop_rc** Type, see Section 20.7.4
- **omp_get_num_interop_properties** Routine, see Section 26.1

27 Memory Management Routines

This chapter describes OpenMP memory-management routines, which are OpenMP API routines that have the memory-management-routine property. These routines support memory management on the current device.

Fortran

The Fortran versions of the memory-management routines require an explicit interface and thus might not be provided in the deprecated include file **`omp_lib.h`**.

Fortran

27.1 Memory Space Retrieving Routines

This section describes the memory-space-retrieving routines, which are routines that have the memory-space-retrieving property. Each of these routines returns a handle to a memory space that represents a set of storage resources accessible by one or more devices. For each storage resource the following requirements are true:

- The storage resource is accessible by each of the devices selected by the routine; and
- The storage resource is part of the memory space represented by the *memspace* argument in each of the devices selected by the routine.

If no set of storage resources matches the above requirements then the special value **`omp_null_mem_space`** is returned. These routines have the all-device-threads binding property for each device selected by the routine. Thus, the binding thread set for a region that corresponds to a memory-space-retrieving routine is all threads on the devices selected by the routine.

The memory spaces returned by these routines are target memory spaces if any of the selected devices is not the current device.

For any memory-space-retrieving routine that takes a *devs* argument, if the array to which the argument points has more than *ndevs* values, the additional values are ignored.

Restrictions

The restrictions to memory-space-retrieving routines are as follows:

- These routines must only be invoked on the host device.
- The *memspace* argument must be one of the predefined memory spaces.
- For any memory-space-retrieving routine that has a *devs* argument, the argument must point to an array that contains at least *ndevs* values.

- For any memory-space-retrieving routine that has a *dev* or *devs* argument, the value of the *dev* argument the *ndevs* values of the array to which *devs* points must be conforming device numbers.

Cross References

- Memory Spaces, see Section 8.1
- **requires** Directive, see Section 10.5
- **target** Construct, see Section 15.8

27.1.1 omp_get_devices_memspace Routine

Name: **omp_get_devices_memspace** **Category:** function	**Properties:** all-device-threads-binding, memory-management-routine, memory-space-retrieving

Return Type and Arguments

Name	Type	Properties
<return type>	memspace_handle	*default*
ndevs	integer	intent(in), positive
devs	integer	intent(in), pointer
memspace	memspace_handle	intent(in), omp

Prototypes

C / C++

```
omp_memspace_handle_t omp_get_devices_memspace(int ndevs,
  const int *devs, omp_memspace_handle_t memspace);
```

C / C++

Fortran

```
integer (kind=omp_memspace_handle_kind) function &
  omp_get_devices_memspace(ndevs, devs, memspace)
  integer, intent(in) :: ndevs, devs(*)
  integer (kind=omp_memspace_handle_kind), intent(in) :: memspace
```

Fortran

Effect

The **omp_get_devices_memspace** routine is a memory-space-retrieving routine. The devices selected by the routine are those specified in the *devs* argument.

Cross References

- Memory Space Retrieving Routines, see Section 27.1
- OpenMP **memspace_handle** Type, see Section 20.8.11

27.1.2 `omp_get_device_memspace` Routine

Name: `omp_get_device_memspace` **Category:** function	**Properties:** all-device-threads-binding, memory-management-routine, memory-space-retrieving

Return Type and Arguments

Name	Type	Properties
<return type>	memspace_handle	*default*
dev	integer	intent(in)
memspace	memspace_handle	intent(in), omp

Prototypes

C / C++

```c
omp_memspace_handle_t omp_get_device_memspace(int dev,
  omp_memspace_handle_t memspace);
```

C / C++

Fortran

```fortran
integer (kind=omp_memspace_handle_kind) function &
  omp_get_device_memspace(dev, memspace)
  integer, intent(in) :: dev
  integer (kind=omp_memspace_handle_kind), intent(in) :: memspace
```

Fortran

Effect

The **`omp_get_device_memspace`** routine is a memory-space-retrieving routine. The device selected by the routine is the device specified in the *dev* argument.

Cross References

- Memory Space Retrieving Routines, see Section 27.1
- OpenMP **`memspace_handle`** Type, see Section 20.8.11

27.1.3 `omp_get_devices_and_host_memspace` Routine

Name: `omp_get_devices_and_host_memspace` **Category:** function	**Properties:** all-device-threads-binding, memory-management-routine, memory-space-retrieving

Return Type and Arguments

Name	Type	Properties
<return type>	memspace_handle	*default*
ndevs	integer	intent(in), positive
devs	integer	intent(in), pointer
memspace	memspace_handle	intent(in), omp

Prototypes

C / C++

```
omp_memspace_handle_t omp_get_devices_and_host_memspace(
  int ndevs, const int *devs, omp_memspace_handle_t memspace);
```

C / C++

Fortran

```
integer (kind=omp_memspace_handle_kind) function &
  omp_get_devices_and_host_memspace(ndevs, devs, memspace)
  integer, intent(in) :: ndevs, devs(*)
  integer (kind=omp_memspace_handle_kind), intent(in) :: memspace
```

Fortran

Effect

The **omp_get_devices_and_host_memspace** routine is a memory-space-retrieving routine. The devices selected by the routine are the host device and those specified in the *devs* argument.

Cross References

- Memory Space Retrieving Routines, see Section 27.1
- OpenMP **memspace_handle** Type, see Section 20.8.11

27.1.4 omp_get_device_and_host_memspace Routine

Name: **omp_get_device_and_host_memspace** **Category:** function	**Properties:** all-device-threads-binding, memory-management-routine, memory-space-retrieving

Return Type and Arguments

Name	Type	Properties
<return type>	memspace_handle	*default*
dev	integer	intent(in)
memspace	memspace_handle	intent(in), omp

Prototypes

C / C++

```
omp_memspace_handle_t omp_get_device_and_host_memspace(int dev,
  omp_memspace_handle_t memspace);
```

C / C++

Fortran

```
integer (kind=omp_memspace_handle_kind) function &
  omp_get_device_and_host_memspace(dev, memspace)
  integer, intent(in) :: dev
  integer (kind=omp_memspace_handle_kind), intent(in) :: memspace
```

Fortran

Effect

The **omp_get_device_and_host_memspace** routine is a memory-space-retrieving routine. The devices selected by the routine are the host device and the device specified in the *dev* argument.

Cross References

- Memory Space Retrieving Routines, see Section 27.1
- OpenMP **memspace_handle** Type, see Section 20.8.11

27.1.5 omp_get_devices_all_memspace Routine

Name: **omp_get_devices_all_memspace** **Category:** function	**Properties:** all-device-threads-binding, memory-management-routine, memory-space-retrieving

Return Type and Arguments

Name	Type	Properties
<*return type*>	memspace_handle	*default*
memspace	memspace_handle	intent(in), omp

Prototypes

C / C++

```c
omp_memspace_handle_t omp_get_devices_all_memspace(
  omp_memspace_handle_t memspace);
```

C / C++

Fortran

```fortran
integer (kind=omp_memspace_handle_kind) function &
  omp_get_devices_all_memspace(memspace)
  integer (kind=omp_memspace_handle_kind), intent(in) :: memspace
```

Fortran

Effect

The **omp_get_devices_all_memspace** routine is a memory-space-retrieving routine. The devices selected by the routine are all available devices.

Cross References

- Memory Space Retrieving Routines, see Section 27.1
- OpenMP **memspace_handle** Type, see Section 20.8.11

27.2 omp_get_memspace_num_resources Routine

Name: **omp_get_memspace_num_resources** **Category:** function	**Properties:** all-device-threads-binding, memory-management-routine

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
memspace	memspace_handle	intent(in), omp

Prototypes

C / C++

```
int omp_get_memspace_num_resources(
  omp_memspace_handle_t memspace);
```

C / C++

Fortran

```
integer function omp_get_memspace_num_resources(memspace)
  integer (kind=omp_memspace_handle_kind), intent(in) :: memspace
```

Fortran

Effect

The **omp_get_memspace_num_resources** routine is a memory-management routine that returns the number of distinct storage resources that are associated with the memory space represented by the *memspace* handle.

Restrictions

The restrictions to the **omp_get_memspace_num_resources** routine are as follows:

- The *memspace* argument must be a valid memory space.

Cross References

- Memory Spaces, see Section 8.1
- OpenMP **memspace_handle** Type, see Section 20.8.11

27.3 omp_get_memspace_pagesize Routine

Name: omp_get_memspace_pagesize **Category:** function	**Properties:** all-device-threads-binding, iso_c_binding, memory-management-routine

Return Type and Arguments

Name	Type	Properties
<return type>	c_size_t	*default*
memspace	memspace_handle	intent(in), omp

Prototypes

C / C++

```
size_t omp_get_memspace_pagesize(omp_memspace_handle_t memspace);
```

C / C++

Fortran

```
integer (kind=c_size_t) function omp_get_memspace_pagesize(&
  memspace) bind(c)
  use, intrinsic :: iso_c_binding, only : c_size_t
  integer (kind=omp_memspace_handle_kind), intent(in) :: memspace
```

Fortran

Effect

The **omp_get_memspace_pagesize** routine is a memory-management routine that returns the page size that the memory space represented by the *memspace* handle supports.

Restrictions

The restrictions to the **omp_get_memspace_pagesize** routine are as follows:

- The *memspace* argument must be a valid memory space.

Cross References

- Memory Spaces, see Section 8.1
- OpenMP **memspace_handle** Type, see Section 20.8.11

27.4 omp_get_submemspace Routine

Name: omp_get_submemspace **Category:** function	**Properties:** all-device-threads-binding, memory-management-routine

Return Type and Arguments

Name	Type	Properties
<return type>	memspace_handle	*default*
memspace	memspace_handle	intent(in), omp
num_resources	integer	intent(in), non-negative
resources	integer	intent(in), pointer

Prototypes

C / C++

```
omp_memspace_handle_t omp_get_submemspace(
  omp_memspace_handle_t memspace, int num_resources,
  const int *resources);
```

C / C++

Fortran

```
integer (kind=omp_memspace_handle_kind) function &
  omp_get_submemspace(memspace, num_resources, resources)
  integer (kind=omp_memspace_handle_kind), intent(in) :: memspace
  integer, intent(in) :: num_resources, resources(*)
```

Fortran

Effect

The **omp_get_submemspace** routine is a memory-management routine that returns a new memory space that contains a subset of the resources of the original memory space. The new memory space represents only the resources of the memory space represented by the *memspace* handle that are specified by the *resources* argument. If *num_resources* is zero or a memory space cannot be created for the requested resources, the special value **omp_null_mem_space** is returned.

Restrictions

The restrictions to the **omp_get_submemspace** routine are as follows:

- The *memspace* argument must be a valid memory space.
- The *resources* array must contain at least as many entries as specified by the *num_resources* argument.
- The value of each entry of the *resources* array must be between 0 and one less than the number of resources associated with the memory space represented by the *memspace* argument.

Cross References

- Memory Spaces, see Section 8.1
- OpenMP **memspace_handle** Type, see Section 20.8.11

27.5 OpenMP Memory Partitioning Routines

This section describes the memory-partitioning routines, which are routines that have the memory-partitioning property. These routines provide mechanisms to create and to use memory partitioners.

27.5.1 `omp_init_mempartitioner` Routine

Name: **omp_init_mempartitioner** Category: subroutine	Properties: all-device-threads-binding, memory-management-routine, memory-partitioning

Arguments

Name	Type	Properties
partitioner	mempartitioner	C/C++ pointer, omp, intent(out)
lifetime	mempartitioner_lifetime	omp, intent(in)
compute_proc	mempartitioner_compute_proc	omp, procedure
release_proc	mempartitioner_release_proc	omp, procedure

Prototypes

C / C++

```
void omp_init_mempartitioner(omp_mempartitioner_t *partitioner,
  omp_mempartitioner_lifetime_t lifetime,
  omp_mempartitioner_compute_proc_t compute_proc,
  omp_mempartitioner_release_proc_t release_proc);
```

C / C++

Fortran

```
subroutine omp_init_mempartitioner(partitioner, lifetime, &
  compute_proc, release_proc)
  integer (kind=omp_mempartitioner_kind), intent(out) :: &
    partitioner
  integer (kind=omp_mempartitioner_lifetime_kind), &
    intent(in) :: lifetime
  procedure (omp_mempartitioner_compute_proc_t) compute_proc
  procedure (omp_mempartitioner_release_proc_t) release_proc
```

Fortran

Effect

The **omp_init_mempartitioner** routine initializes the memory partitioner that the *partitioner* object represents with the lifetime specified by the *lifetime* argument, and the *compute_proc* partition computation procedure and the *release_proc* partition release procedure.

Once initialized the *partitioner* object can be associated with an allocator when the allocator is initialized with **omp_init_allocator** by using the **omp_atk_partitioner** trait. If the **omp_atk_partition** allocator trait is set to **omp_atv_partitioner**, then, for allocations that use the allocator, the number of memory parts of an allocation and how they are distributed across the storage resources are defined by a memory partition object that must be initialized in the *compute_proc* provided in this routine through calls to the **omp_init_mempartition** and **omp_mempartition_set_part** routines.

If the value of the *lifetime* argument is **omp_allocator_mempartition** then the memory partition object that is created through the *compute_proc* procedure might be used for all allocations of an allocator that has the same allocation size. If the value of the *lifetime* argument is **omp_dynamic_mempartition** then a memory partition object will be initialized for every allocation.

Restrictions

The restrictions to the **omp_init_mempartitioner** routine are as follows:

- The memory partitioner represented by the *partitioner* argument must be in the uninitialized state.

Cross References

- Memory Allocators, see Section 8.2
- Memory Spaces, see Section 8.1
- OpenMP **mempartitioner** Type, see Section 20.8.7
- OpenMP **mempartitioner_compute_proc** Type, see Section 20.8.9
- OpenMP **mempartitioner_lifetime** Type, see Section 20.8.8
- OpenMP **mempartitioner_release_proc** Type, see Section 20.8.10

27.5.2 omp_destroy_mempartitioner Routine

Name: **omp_destroy_mempartitioner** **Category:** subroutine	**Properties:** all-device-threads-binding, memory-management-routine, memory-partitioning

Arguments

Name	Type	Properties
partitioner	mempartitioner	C/C++ pointer, omp, intent(in)

Prototypes

C / C++

```
void omp_destroy_mempartitioner(
  const omp_mempartitioner_t *partitioner);
```

C / C++

Fortran

```
subroutine omp_destroy_mempartitioner(partitioner)
  integer (kind=omp_mempartitioner_kind), intent(in) :: &
    partitioner
```

Fortran

Effect

The effect of the **omp_destroy_mempartitioner** routine is to uninitialize a memory partitioner. Thus, the routine changes the state of the memory partitioner object represented by the *partitioner* argument to uninitialized and releases all resources associated with it.

Restrictions

The restrictions to the **omp_destroy_mempartitioner** routine are as follows:

- The memory partitioner represented by the *partitioner* argument must be in the initialized state.
- Any allocator that references the memory partitioner object represented by the *partitioner* argument must be destroyed before this routine is called.

Cross References

- Memory Allocators, see Section 8.2
- OpenMP **mempartitioner** Type, see Section 20.8.7

27.5.3 omp_init_mempartition Routine

Name: **omp_init_mempartition** **Category:** subroutine	**Properties:** all-device-threads-binding, iso_c_binding, memory-management-routine, memory-partitioning

Arguments

Name	Type	Properties
partition	mempartition	C/C++ pointer, omp, intent(out)
nparts	c_size_t	intent(in), iso_c, intent(in)
user_data	c_ptr	intent(in), iso_c, intent(in)

Prototypes

C / C++

```
void omp_init_mempartition(omp_mempartition_t *partition,
  size_t nparts, const void *user_data);
```

C / C++

Fortran

```
subroutine omp_init_mempartition(partition, nparts, user_data) &
  bind(c)
  use, intrinsic :: iso_c_binding, only : c_size_t, c_ptr
  integer (kind=omp_mempartition_kind), intent(out) :: partition
  integer (kind=c_size_t), intent(in) :: nparts
  type (c_ptr), intent(in) :: user_data
```

Fortran

Effect

The effect of the **omp_init_mempartition** routine is to initialize a memory partition object. Thus, the routine sets the memory partition object indicated by the *partition* argument to represent a memory partition of *nparts* parts and associates the user data indicated by the *user_data* argument with it.

Restrictions

The restrictions to the **omp_init_mempartition** routine are as follows:

- The memory partition represented by the *partition* argument must be in the uninitialized state.
- This routine must only be called by a procedure that is associated with the memory partitioner object that allocated the memory partition indicated by the *partition* argument.

Cross References

- OpenMP Memory Management Types, see Section 20.8
- OpenMP **mempartitioner** Type, see Section 20.8.7

27.5.4 omp_destroy_mempartition Routine

Name: **omp_destroy_mempartition** **Category:** subroutine	**Properties:** all-device-threads-binding, memory-management-routine, memory-partitioning

Arguments

Name	Type	Properties
partition	mempartition	C/C++ pointer, omp, intent(in)

Prototypes

C / C++

```
void omp_destroy_mempartition(
  const omp_mempartition_t *partition);
```

C / C++

Fortran

```
subroutine omp_destroy_mempartition(partition)
  integer (kind=omp_mempartition_kind), intent(in) :: partition
```

Fortran

Effect

The effect of the **`omp_destroy_mempartition`** routine is to uninitialize a memory partition object. Thus, the routine releases the memory partition indicated by the *partition* argument and all resources associated with it.

Restrictions

The restrictions to the **`omp_destroy_mempartition`** routine are as follows:

- The memory partition represented by the *partition* argument must be in the initialized state.
- This routine must only be called by a procedure that is associated with the memory partitioner object that allocated the memory partition indicated by the *partition* argument.

Cross References

- OpenMP Memory Management Types, see Section 20.8
- OpenMP **`mempartitioner`** Type, see Section 20.8.7

27.5.5 `omp_mempartition_set_part` Routine

Name: `omp_mempartition_set_part` **Category:** function	**Properties:** all-device-threads-binding, iso_c_binding, memory-management-routine, memory-partitioning

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
partition	mempartition	C/C++ pointer, omp, intent(out)
part	c_size_t	intent(in), iso_c
resource	integer	intent(in), iso_c
size	c_size_t	intent(in), iso_c

Prototypes

C / C++

```
int omp_mempartition_set_part(omp_mempartition_t *partition,
    size_t part, int resource, size_t size);
```

C / C++

Fortran

```
integer function omp_mempartition_set_part(partition, part, &
  resource, size) bind(c)
  use, intrinsic :: iso_c_binding, only : c_size_t
  integer (kind=omp_mempartition_kind), intent(out) :: partition
  integer (kind=c_size_t), intent(in) :: part, size
  integer, intent(in) :: resource
```

Fortran

Effect

The effect of the **omp_mempartition_set_part** routine is to define the size and resource of a given part of a memory partition. Thus the routine defines the part number indicated by the *part* argument of the memory partition object indicated by the *partition* argument to be associated to the resource indicated by the *resource* argument and to be of size indicated by the *size* argument.

The size of all parts of a memory partition, except the last one, need to be a multiple of the page size that the memory space where the memory is being allocated supports. If the specified *size* cannot be supported by the specified *resource*, this routine returns negative one. Otherwise, it returns zero.

Restrictions

The restrictions to the **omp_mempartition_set_part** routine are as follows:

- The memory partition represented by the *partition* argument must be in the initialized state.
- This routine must only be called by a procedure that is associated with the memory partitioner object that allocated the memory partition indicated by the *partition* argument.

Cross References

- Memory Spaces, see Section 8.1
- OpenMP Memory Management Types, see Section 20.8
- OpenMP **mempartitioner** Type, see Section 20.8.7

27.5.6 omp_mempartition_get_user_data Routine

Name: **omp_mempartition_get_user_data** **Category:** function	**Properties:** all-device-threads-binding, iso_c_binding, memory-management-routine, memory-partitioning

Return Type and Arguments

Name	Type	Properties
<return type>	c_ptr	*default*
partition	mempartition	intent(in), C/C++ pointer, omp

Prototypes

C / C++

```
void *omp_mempartition_get_user_data(
  const omp_mempartition_t *partition);
```

C / C++

Fortran

```
type (c_ptr) function omp_mempartition_get_user_data(partition) &
  bind(c)
  use, intrinsic :: iso_c_binding, only : c_ptr
  integer (kind=omp_mempartition_kind), intent(in) :: partition
```

Fortran

Effect

The effect of the **`omp_mempartition_get_user_data`** routine is to retrieve the user data that was associated with the memory partition when it was created. Thus, the routine returns the data associated with the memory partition object indicated by the *partition* argument.

Restrictions

The restrictions to the **`omp_mempartition_get_user_data`** routine are as follows:

- The memory partition represented by the *partition* argument must be in the initialized state.
- This routine must only be called by a procedure that is associated with the memory partitioner object that allocated the memory partition indicated by the *partition* argument.

Cross References

- OpenMP Memory Management Types, see Section 20.8
- OpenMP **`mempartitioner`** Type, see Section 20.8.7

27.6 `omp_init_allocator` Routine

Name: `omp_init_allocator` **Category:** function	**Properties:** all-device-threads-binding, memory-management-routine

Return Type and Arguments

Name	Type	Properties
<return type>	allocator_handle	*default*
memspace	memspace_handle	intent(in), omp
ntraits	integer	intent(in)
traits	alloctrait	intent(in), pointer, omp

Prototypes

C / C++

```
omp_allocator_handle_t omp_init_allocator(
  omp_memspace_handle_t memspace, int ntraits,
  const omp_alloctrait_t *traits);
```

C / C++

Fortran

```
integer (kind=omp_allocator_handle_kind) function &
  omp_init_allocator(memspace, ntraits, traits)
  integer (kind=omp_memspace_handle_kind), intent(in) :: memspace
  integer, intent(in) :: ntraits
  integer (kind=omp_alloctrait_kind), intent(in) :: traits(*)
```

Fortran

Effect

The **omp_init_allocator** routine creates a new allocator that is associated with the *memspace* memory space and returns a handle to it. All allocations through the created allocator will behave according to the allocator traits specified in the *traits* argument. The number of traits in the *traits* argument is specified by the *ntraits* argument. If the special **omp_atv_default** value is used for a given trait, then its value will be the default value specified in Table 8.2 for that trait.

If *memspace* has the value **omp_null_mem_space**, the effect of this routine will be as if the value of *memspace* was **omp_default_mem_space**. If *memspace* is **omp_default_mem_space** and the *traits* argument is an empty set, this routine will always return a handle to an allocator. Otherwise, if an allocator based on the requirements cannot be created then the special **omp_null_allocator** handle is returned.

Restrictions

The restrictions to the **omp_init_allocator** routine are as follows:

- Each allocator trait must be specified at most once.
- The *memspace* argument must be a valid memory space handle or the value **omp_null_mem_space**.
- If the *ntraits* argument is positive then the *traits* argument must specify at least *ntraits* traits.
- The use of an allocator returned by this routine on devices other than the one on which it was created results in unspecified behavior.
- Unless a **requires** directive with the **dynamic_allocators** clause is present in the same compilation unit, using this routine in a **target** region results in unspecified behavior.
- If the *memspace* handle represents a target memory space, the values **omp_atv_device**, **omp_atv_cgroup**, **omp_atv_pteam** or **omp_atv_thread** must not be specified for the **omp_atk_access** allocator trait.

Cross References

- OpenMP **`allocator_handle`** Type, see Section 20.8.1
- Memory Allocators, see Section 8.2
- Memory Spaces, see Section 8.1
- OpenMP **`memspace_handle`** Type, see Section 20.8.11
- **`requires`** Directive, see Section 10.5
- **`target`** Construct, see Section 15.8

27.7 `omp_destroy_allocator` Routine

Name: `omp_destroy_allocator` **Category:** subroutine	**Properties:** all-device-threads-binding, memory-management-routine

Arguments

Name	Type	Properties
allocator	allocator_handle	intent(in), omp

Prototypes

C / C++

```
void omp_destroy_allocator(omp_allocator_handle_t allocator);
```

C / C++

Fortran

```
subroutine omp_destroy_allocator(allocator)
  integer (kind=omp_allocator_handle_kind), intent(in) :: &
    allocator
```

Fortran

Effect

The **`omp_destroy_allocator`** routine releases all resources used to implement the *allocator* handle. If *allocator* is **`omp_null_allocator`** then this routine has no effect.

Restrictions

The restrictions to the **`omp_destroy_allocator`** routine are as follows:

- The *allocator* argument must not represent a predefined memory allocator.
- Accessing any memory allocated by the *allocator* after this call results in unspecified behavior.
- Unless a **`requires`** directive with the **`dynamic_allocators`** clause is present in the same compilation unit, using this routine in a **`target`** region results in unspecified behavior.

Cross References

- OpenMP **allocator_handle** Type, see Section 20.8.1
- Memory Allocators, see Section 8.2
- **requires** Directive, see Section 10.5
- **target** Construct, see Section 15.8

27.8 Memory Allocator Retrieving Routines

This section describes the memory-allocator-retrieving routines, which are routines that have the memory-allocator-retrieving property. Each of these routines returns a handle to a predefined memory allocator that represents the default memory allocator for a given device for a certain kind of memory. If the implementation does not have a predefined allocator that satisfies the request, then the special value **omp_null_allocator** is returned. For any memory-allocator-retrieving routine that takes a *devs* argument, if the array to which the argument points has more than *ndevs* values, the additional values are ignored. Each of these routines returns an allocator that may be used anywhere that requires a predefined allocator specified in Table 8.3. The allocator is associated with a target memory space if any of the selected devices is not the current device.

Restrictions

The restrictions to memory-allocator-retrieving routines are as follows:

- These routines must only be invoked on the host device.
- The *memspace* argument must not be one of the predefined memory spaces.
- For any memory-allocator-retrieving routine that has a *devs* argument, the argument must point to an array that contains at least *ndevs* values.
- For any memory-allocator-retrieving routine that has a *dev* or *devs* argument, the value of the *dev* argument the *ndevs* values of the array to which *devs* points must be conforming device numbers.

Cross References

- Memory Allocators, see Section 8.2
- Memory Spaces, see Section 8.1

27.8.1 omp_get_devices_allocator Routine

Name: omp_get_devices_allocator **Category:** function	**Properties:** all-device-threads-binding, memory-management-routine, memory-allocator-retrieving

Return Type and Arguments

Name	Type	Properties
<return type>	allocator_handle	*default*
ndevs	integer	intent(in), positive
devs	integer	intent(in), pointer
memspace	memspace_handle	intent(in), omp

Prototypes

C / C++

```
omp_allocator_handle_t omp_get_devices_allocator(int ndevs,
  const int *devs, omp_memspace_handle_t memspace);
```

C / C++

Fortran

```
integer (kind=omp_allocator_handle_kind) function &
  omp_get_devices_allocator(ndevs, devs, memspace)
  integer, intent(in) :: ndevs, devs(*)
  integer (kind=omp_memspace_handle_kind), intent(in) :: memspace
```

Fortran

Effect

The **omp_get_devices_allocator** routine is a memory-allocator-retrieving routine. The devices selected by the routine are those specified in the *devs* argument.

Cross References

- OpenMP **allocator_handle** Type, see Section 20.8.1
- Memory Allocator Retrieving Routines, see Section 27.8
- OpenMP **memspace_handle** Type, see Section 20.8.11

27.8.2 omp_get_device_allocator Routine

Name: omp_get_device_allocator **Category:** function	**Properties:** all-device-threads-binding, memory-management-routine, memory-allocator-retrieving

Return Type and Arguments

Name	Type	Properties
<return type>	allocator_handle	*default*
dev	integer	intent(in)
memspace	memspace_handle	intent(in), omp

Prototypes

C / C++

```
omp_allocator_handle_t omp_get_device_allocator(int dev,
  omp_memspace_handle_t memspace);
```

C / C++

Fortran

```
integer (kind=omp_allocator_handle_kind) function &
  omp_get_device_allocator(dev, memspace)
  integer, intent(in) :: dev
  integer (kind=omp_memspace_handle_kind), intent(in) :: memspace
```

Fortran

Effect

The **omp_get_device_allocator** routine is a memory-allocator-retrieving routine. The device selected by the routine is the device specified in the *dev* argument.

Cross References

- OpenMP **allocator_handle** Type, see Section 20.8.1
- Memory Allocator Retrieving Routines, see Section 27.8
- OpenMP **memspace_handle** Type, see Section 20.8.11

27.8.3 omp_get_devices_and_host_allocator Routine

Name: **omp_get_devices_and_host_allocator** **Category:** function	**Properties:** all-device-threads-binding, memory-management-routine, memory-allocator-retrieving

Return Type and Arguments

Name	Type	Properties
<return type>	allocator_handle	*default*
ndevs	integer	intent(in), positive
devs	integer	intent(in), pointer
memspace	memspace_handle	intent(in), omp

Prototypes

C / C++

```
omp_allocator_handle_t omp_get_devices_and_host_allocator(
  int ndevs, const int *devs, omp_memspace_handle_t memspace);
```

C / C++

Fortran

```
integer (kind=omp_allocator_handle_kind) function &
  omp_get_devices_and_host_allocator(ndevs, devs, memspace)
  integer, intent(in) :: ndevs, devs(*)
  integer (kind=omp_memspace_handle_kind), intent(in) :: memspace
```

Fortran

Effect

The **omp_get_devices_and_host_allocator** routine is a memory-allocator-retrieving routine. The devices selected by the routine are the host device and those specified in the *devs* argument.

Cross References

- OpenMP **allocator_handle** Type, see Section 20.8.1
- Memory Allocator Retrieving Routines, see Section 27.8
- OpenMP **memspace_handle** Type, see Section 20.8.11

27.8.4 omp_get_device_and_host_allocator Routine

Name: **omp_get_device_and_host_allocator** **Category:** function	**Properties:** all-device-threads-binding, memory-management-routine, memory-allocator-retrieving

Return Type and Arguments

Name	Type	Properties
<return type>	allocator_handle	*default*
dev	integer	intent(in)
memspace	memspace_handle	intent(in), omp

Prototypes

C / C++

```
omp_allocator_handle_t omp_get_device_and_host_allocator(int dev,
  omp_memspace_handle_t memspace);
```

C / C++

Fortran

```
integer (kind=omp_allocator_handle_kind) function &
  omp_get_device_and_host_allocator(dev, memspace)
  integer, intent(in) :: dev
  integer (kind=omp_memspace_handle_kind), intent(in) :: memspace
```

Fortran

Effect

The **omp_get_device_and_host_allocator** routine is a memory-allocator-retrieving routine. The devices selected by the routine are the host device and the device specified in the *dev* argument.

Cross References

- OpenMP **allocator_handle** Type, see Section 20.8.1
- Memory Allocator Retrieving Routines, see Section 27.8
- OpenMP **memspace_handle** Type, see Section 20.8.11

27.8.5 omp_get_devices_all_allocator Routine

Name: **omp_get_devices_all_allocator** **Category:** function	**Properties:** all-device-threads-binding, memory-management-routine, memory-allocator-retrieving

Return Type and Arguments

Name	Type	Properties
<return type>	allocator_handle	*default*
memspace	memspace_handle	intent(in), omp

Prototypes

C / C++

```
omp_allocator_handle_t omp_get_devices_all_allocator(
  omp_memspace_handle_t memspace);
```

C / C++

Fortran

```
integer (kind=omp_allocator_handle_kind) function &
  omp_get_devices_all_allocator(memspace)
  integer (kind=omp_memspace_handle_kind), intent(in) :: memspace
```

Fortran

Effect

The **omp_get_devices_all_allocator** routine is a memory-allocator-retrieving routine. The devices selected by the routine are all available devices.

Cross References

- OpenMP **allocator_handle** Type, see Section 20.8.1
- Memory Space Retrieving Routines, see Section 27.1
- OpenMP **memspace_handle** Type, see Section 20.8.11

27.9 omp_set_default_allocator Routine

Name: **omp_set_default_allocator** **Category:** subroutine	**Properties:** binding-implicit-task-binding, memory-management-routine

Arguments

Name	Type	Properties
allocator	allocator_handle	omp, intent(in)

Prototypes

C / C++

```
void omp_set_default_allocator(omp_allocator_handle_t allocator);
```

C / C++

Fortran

```
subroutine omp_set_default_allocator(allocator)
  integer (kind=omp_allocator_handle_kind), intent(in) :: &
    allocator
```

Fortran

Effect

The effect of the **omp_set_default_allocator** is to set the value of the *def-allocator-var* ICV of the binding implicit task to the value specified in the *allocator* argument. Thus, it sets the default memory allocator to be used by allocation calls, **allocate** clauses and **allocate** and **allocators** directives that do not specify an allocator. This routine has the binding-implicit-task binding property so the binding task set for an **omp_set_default_allocator** region is the binding implicit task.

Restrictions

The restrictions to the **omp_set_default_allocator** routine are as follows:

- The *allocator* argument must be a valid memory allocator handle.

Cross References

- **allocate** Clause, see Section 8.6
- **allocate** Directive, see Section 8.5
- OpenMP **allocator_handle** Type, see Section 20.8.1
- **allocators** Construct, see Section 8.7
- Memory Allocators, see Section 8.2
- *def-allocator-var* ICV, see Table 3.1

27.10 omp_get_default_allocator Routine

Name: omp_get_default_allocator Category: function	Properties: binding-implicit-task-binding, memory-management-routine

Return Type

Name	Type	Properties
<return type>	allocator_handle	*default*

Prototypes

C / C++

```
omp_allocator_handle_t omp_get_default_allocator(void);
```

C / C++

Fortran

```
integer (kind=omp_allocator_handle_kind) function &
  omp_get_default_allocator()
```

Fortran

Effect

The **omp_get_default_allocator** routine returns the value of the *def-allocator-var* ICV of the binding implicit task, which is a handle to the memory allocator to be used by allocation calls, **allocate** clauses and **allocate** and **allocators** directives that do not specify an allocator. This routine has the binding-implicit-task binding property, so the binding task set for an **omp_get_default_allocator** region is the binding implicit task.

Cross References

- **allocate** Clause, see Section 8.6
- **allocate** Directive, see Section 8.5
- OpenMP **allocator_handle** Type, see Section 20.8.1
- **allocators** Construct, see Section 8.7
- Memory Allocators, see Section 8.2
- *def-allocator-var* ICV, see Table 3.1

27.11 Memory Allocating Routines

This section describes the memory-allocating routines, which are routines that have the memory-allocating-routine property. Each of these routines requests a memory allocation from the memory allocator that its *allocator* argument specifies. If the *allocator* argument is **omp_null_allocator**, the routine uses the memory allocator specified by the *def-allocator-var* ICV of the binding implicit task. Upon success, these routines return a pointer to the allocated memory. Otherwise, the behavior that the **omp_atk_fallback** trait of the allocator specifies is followed. Pointers returned by these routines are considered device pointers if at least one of the devices associated with the allocator that the *allocator* argument represents is not the current device.

OpenMP provides several kinds of memory-allocating routines. The memory allocated by raw-memory-allocating routines, which have the raw-memory-allocating-routine property, is uninitialized. The memory allocated by zeroed-memory-allocating routines, which have the zeroed-memory-allocating-routine property, is set to zero before the routine returns.

The memory allocated by aligned-memory-allocating routines, which have the aligned-memory-allocating-routine property, is byte-aligned to at least the maximum of the alignment required by **malloc**, the **omp_atk_alignment** trait of the allocator and the value of their *alignment* argument. The memory allocated by all other memory-allocating routines is byte-aligned to at least the maximum of the alignment required by **malloc** and the **omp_atk_alignment** trait of the allocator.

Raw-memory-allocating routines request a memory allocation of *size* bytes from the specified memory allocator. Zeroed-memory-allocating routines request a memory allocation for an array of *nmemb* elements, each of which has a size of *size* bytes. If any of the *size* or *nmemb* arguments are zero, these routines return NULL.

Memory-reallocating routines deallocate the memory to which the *ptr* argument points and request a new memory allocation of *size* bytes from the memory allocator that is specified by the *allocator* argument. If the *free_allocator* argument is **omp_null_allocator**, the implementation will determine that value automatically. If the *allocator* argument is **omp_null_allocator**, the behavior is as if the memory allocator that allocated the memory to which *ptr* argument points is passed to the *allocator* argument. Upon success, each of these routines returns a (possibly moved) pointer to the allocated memory and the contents of the new object will be the same as that of the old object prior to deallocation, up to the minimum size of the old allocated size and *size*. Any bytes in the new object beyond the old allocated size will have unspecified values. If the allocation failed, the behavior that the **omp_atk_fallback** trait of the *allocator* specifies will be followed. If *ptr* is NULL, a memory-reallocating routine behaves the same as a raw-memory-allocating routine with the same *size* and *allocator* arguments. If *size* is zero, a memory-reallocating routine returns NULL and the old allocation is deallocated. If *size* is not zero, the old allocation will be deallocated if and only if the routine returns a non-null value.

C++

The C++ version of all memory-allocating routines have the overloaded property since they are overloaded routines for which the *allocator* argument may be omitted, in which case the effect is as if **omp_null_allocator** is specified.

C++

Restrictions

The restrictions to memory-allocating routines are as follows:

- Unless the **unified_address** clause is specified or the current device is an associated device of the allocator, pointer arithmetic is not supported on the pointer that a memory-allocating routine returns.
- Each *allocator* and *free_allocator* argument must be a constant expression that evaluates to a handle that represents a predefined memory allocator.
- The value of the *alignment* argument to an aligned-memory-allocating routine must be a power of two.
- The value of a *size* argument to an aligned-memory-allocating routine must be a multiple of the *alignment* argument.
- The value of the *ptr* argument to a memory-reallocating routine must have been returned by a memory-allocating routine.
- If the *free_allocator* argument is specified for a memory-reallocating routine, it must be the memory allocator to which the previous allocation request was made.
- Using a memory-reallocating routine on memory that was already deallocated or that was allocated by an allocator that has already been destroyed with **omp_destroy_allocator** results in unspecified behavior.
- Unless a **requires** directive with the **dynamic_allocators** clause is present in the same compilation unit, memory-allocating routines that appear in **target** regions must not pass **omp_null_allocator** as the *allocator* or *free_allocator* argument.

Cross References

- Memory Allocators, see Section 8.2
- *def-allocator-var* ICV, see Table 3.1
- **omp_destroy_allocator** Routine, see Section 27.7
- **requires** Directive, see Section 10.5
- **target** Construct, see Section 15.8

27.11.1 omp_alloc Routine

Name: omp_alloc Category: function	Properties: iso_c_binding, memory-allocating-routine, memory-management-routine, overloaded, raw-memory-allocating-routine

Return Type and Arguments

Name	Type	Properties
<return type>	c_ptr	*default*
size	c_size_t	iso_c, value
allocator	allocator_handle	value, omp

Prototypes

C

```c
void *omp_alloc(size_t size, omp_allocator_handle_t allocator);
```

C

C++

```cpp
void *omp_alloc(size_t size,
  omp_allocator_handle_t allocator = omp_null_allocator);
```

C++

Fortran

```fortran
type (c_ptr) function omp_alloc(size, allocator) bind(c)
  use, intrinsic :: iso_c_binding, only : c_ptr, c_size_t
  integer (kind=c_size_t), value :: size
  integer (kind=omp_allocator_handle_kind), value :: allocator
```

Fortran

Effect

The **omp_alloc** routine is a raw-memory-allocating routine.

Cross References

- OpenMP **allocator_handle** Type, see Section 20.8.1
- Memory Allocating Routines, see Section 27.11

27.11.2 omp_aligned_alloc Routine

Name: omp_aligned_alloc **Category:** function	**Properties:** aligned-memory-allocating-routine, iso_c_binding, memory-allocating-routine, memory-management-routine, overloaded, raw-memory-allocating-routine

Return Type and Arguments

Name	Type	Properties
<return type>	c_ptr	*default*
alignment	c_size_t	iso_c, value
size	c_size_t	iso_c, value
allocator	allocator_handle	value, omp

Prototypes

C

```c
void *omp_aligned_alloc(size_t alignment, size_t size,
  omp_allocator_handle_t allocator);
```

C

C++

```cpp
void *omp_aligned_alloc(size_t alignment, size_t size,
  omp_allocator_handle_t allocator = omp_null_allocator);
```

C++

Fortran

```fortran
type (c_ptr) function omp_aligned_alloc(alignment, size, &
  allocator) bind(c)
  use, intrinsic :: iso_c_binding, only : c_ptr, c_size_t
  integer (kind=c_size_t), value :: alignment, size
  integer (kind=omp_allocator_handle_kind), value :: allocator
```

Fortran

Effect

The **omp_aligned_alloc** routine is a raw-memory-allocating routine and an aligned-memory-allocating routine.

Cross References

- OpenMP **allocator_handle** Type, see Section 20.8.1
- Memory Allocating Routines, see Section 27.11

27.11.3 omp_calloc Routine

Name: **omp_calloc** **Category:** function	**Properties:** iso_c_binding, memory-allocating-routine, memory-management-routine, overloaded, zeroed-memory-allocating-routine

Return Type and Arguments

Name	Type	Properties
<return type>	c_ptr	*default*
nmemb	c_size_t	iso_c, value
size	c_size_t	iso_c, value
allocator	allocator_handle	value, omp

Prototypes

C

```c
void *omp_calloc(size_t nmemb, size_t size,
  omp_allocator_handle_t allocator);
```

C

C++

```cpp
void *omp_calloc(size_t nmemb, size_t size,
  omp_allocator_handle_t allocator = omp_null_allocator);
```

C++

Fortran

```fortran
type (c_ptr) function omp_calloc(nmemb, size, allocator) &
  bind(c)
  use, intrinsic :: iso_c_binding, only : c_ptr, c_size_t
  integer (kind=c_size_t), value :: nmemb, size
  integer (kind=omp_allocator_handle_kind), value :: allocator
```

Fortran

Effect

The **omp_calloc** routine is a zeroed-memory-allocating routines.

Cross References

- OpenMP **allocator_handle** Type, see Section 20.8.1
- Memory Allocating Routines, see Section 27.11

27.11.4 omp_aligned_calloc Routine

Name: `omp_aligned_calloc` **Category:** function	**Properties:** aligned-memory-allocating-routine, iso_c_binding, memory-allocating-routine, memory-management-routine, overloaded, zeroed-memory-allocating-routine

Return Type and Arguments

Name	Type	Properties
<return type>	c_ptr	*default*
alignment	c_size_t	iso_c, value
nmemb	c_size_t	iso_c, value
size	c_size_t	iso_c, value
allocator	allocator_handle	value, omp

Prototypes

C

```c
void *omp_aligned_calloc(size_t alignment, size_t nmemb,
  size_t size, omp_allocator_handle_t allocator);
```

C

C++

```cpp
void *omp_aligned_calloc(size_t alignment, size_t nmemb,
  size_t size,
  omp_allocator_handle_t allocator = omp_null_allocator);
```

C++

Fortran

```fortran
type (c_ptr) function omp_aligned_calloc(alignment, nmemb, size, &
  allocator) bind(c)
  use, intrinsic :: iso_c_binding, only : c_ptr, c_size_t
  integer (kind=c_size_t), value :: alignment, nmemb, size
  integer (kind=omp_allocator_handle_kind), value :: allocator
```

Fortran

Effect

The **`omp_aligned_calloc`** routine is a zeroed-memory-allocating routine and an aligned-memory-allocating routine.

Cross References

- OpenMP **`allocator_handle`** Type, see Section 20.8.1
- Memory Allocating Routines, see Section 27.11

27.11.5 omp_realloc Routine

Name: **omp_realloc** **Category:** function	**Properties:** iso_c_binding, memory-allocating-routine, memory-management-routine, memory-reallocating-routine, overloaded

Return Type and Arguments

Name	Type	Properties
<return type>	c_ptr	*default*
ptr	c_ptr	iso_c, value
size	c_size_t	iso_c, value
allocator	allocator_handle	value, omp
free_allocator	allocator_handle	value, omp

Prototypes

C

```c
void *omp_realloc(void *ptr, size_t size,
  omp_allocator_handle_t allocator,
  omp_allocator_handle_t free_allocator);
```

C

C++

```cpp
void *omp_realloc(void *ptr, size_t size,
  omp_allocator_handle_t allocator = omp_null_allocator,
  omp_allocator_handle_t free_allocator = omp_null_allocator);
```

C++

Fortran

```fortran
type (c_ptr) function omp_realloc(ptr, size, allocator, &
  free_allocator) bind(c)
  use, intrinsic :: iso_c_binding, only : c_ptr, c_size_t
  type (c_ptr), value :: ptr
  integer (kind=c_size_t), value :: size
  integer (kind=omp_allocator_handle_kind), value :: allocator, &
    free_allocator
```

Fortran

Effect

The **omp_realloc** routine is a memory-reallocating routine.

Cross References

- OpenMP **allocator_handle** Type, see Section 20.8.1
- Memory Allocating Routines, see Section 27.11

27.12 `omp_free` Routine

Name: `omp_free` Category: subroutine	**Properties:** iso_c_binding, memory-management-routine, overloaded

Arguments

Name	Type	Properties
ptr	c_ptr	iso_c, value
allocator	allocator_handle	value, omp

Prototypes

C

```
void omp_free(void *ptr, omp_allocator_handle_t allocator);
```

C

C++

```
void omp_free(void *ptr,
  omp_allocator_handle_t allocator = omp_null_allocator);
```

C++

Fortran

```
subroutine omp_free(ptr, allocator) bind(c)
  use, intrinsic :: iso_c_binding, only : c_ptr
  type (c_ptr), value :: ptr
  integer (kind=omp_allocator_handle_kind), value :: allocator
```

Fortran

Effect

The **`omp_free`** routine deallocates the memory to which the *ptr* argument points. If the *allocator* argument is **`omp_null_allocator`**, the implementation will determine that value automatically. If *ptr* is NULL, no operation is performed.

C++

The C++ version of the **`omp_free`** routine has the overloaded property since it is an overloaded routine for which the *allocator* argument may be omitted, in which case the effect is as if **`omp_null_allocator`** is specified.

C++

Restrictions

The restrictions to the **omp_free** routine are as follows:

- The *ptr* argument must have been returned by a memory-allocating routine.
- If the *allocator* argument is specified it must be the memory allocator to which the allocation request was made.
- Using **omp_free** on memory that was already deallocated or that was allocated by an allocator that has already been destroyed with **omp_destroy_allocator** results in unspecified behavior.

Cross References

- OpenMP **allocator_handle** Type, see Section 20.8.1
- Memory Allocating Routines, see Section 27.11
- Memory Allocators, see Section 8.2
- **omp_destroy_allocator** Routine, see Section 27.7

28 Lock Routines

This chapter describes general-purpose lock routines that can be used for synchronization via mutual exclusion. These routines with the lock property operate on OpenMP locks that are represented by OpenMP lock variables. OpenMP lock variables must be accessed only through the lock routines; OpenMP programs that otherwise access OpenMP lock variables are non-conforming.

A lock can be in one of the following lock states: *uninitialized*; *unlocked*; or *locked*. If a lock is in the unlocked state, a task can acquire the lock by executing a lock-acquiring routine, a routine that has the lock-acquiring property, through which it changes the lock state to the locked state. The task that acquires the lock is then said to *own* the lock. A task that owns a lock can release it by executing a lock-releasing routine, a routine that has the lock-releasing property, through which it returns the lock state to the unlocked state. An OpenMP program in which a task executes a lock-releasing routine on a lock that is owned by another task is non-conforming.

OpenMP supports two types of locks: simple locks and nestable locks. A nestable lock can be acquired (i.e., set) multiple times by the same task before being released (i.e., unset); a simple lock cannot be acquired if it is already owned by the task trying to set it. Simple lock variables are associated with simple locks and can only be passed to simple lock routines (routines that have the simple lock property). Nestable lock variables are associated with nestable locks and can only be passed to nestable lock routines (routines that have the nestable lock property).

Each type of lock can also have a synchronization hint that contains information about the intended usage of the lock by the OpenMP program. The effect of the hint is implementation defined. An OpenMP implementation can use this hint to select a usage-specific lock, but hints do not change the mutual exclusion semantics of locks. A compliant implementation can safely ignore the hint.

Constraints on the lock state and ownership of the lock accessed by each of the lock routines are described with the routine. If these constraints are not met, the behavior of the routine is unspecified.

The lock routines access an OpenMP lock variable such that they always read and update its most current value. An OpenMP program does not need to include explicit **`flush`** directives to ensure that the value of a lock is consistent among different tasks.

Restrictions

Restrictions to OpenMP lock routines are as follows:

- The use of the same lock in different contention groups results in unspecified behavior.

28.1 Lock Initializing Routines

Lock-initializing routines are routines with the lock-initializing property. These routines initialize the lock to the unlocked state; that is, no task owns the lock. In addition, the nesting count for a nestable lock is set to zero.

Restrictions

Restrictions to lock-initializing routines are as follows:

- A lock-initializing routine must not access a lock that is not in the uninitialized state.

28.1.1 omp_init_lock Routine

Name: omp_init_lock Category: subroutine	Properties: all-contention-group-tasks-binding, lock-initializing, simple-lock

Arguments

Name	Type	Properties
svar	lock	C/C++ pointer, omp

Prototypes

C / C++

```
void omp_init_lock(omp_lock_t *svar);
```

C / C++

Fortran

```
subroutine omp_init_lock(svar)
  integer (kind=omp_lock_kind) svar
```

Fortran

Effect

The **omp_init_lock** routine is a lock-initializing routine.

Execution Model Events

The *lock-init* event occurs in a thread that executes an **omp_init_lock** region after initialization of the lock, but before it finishes the region.

Tool Callbacks

A thread dispatches a registered **lock_init** callback with **omp_sync_hint_none** as the *hint* argument and **ompt_mutex_lock** as the *kind* argument for each occurrence of a *lock-init* event in that thread. This callback occurs in the task that encounters the routine.

Cross References

- OpenMP **lock** Type, see Section 20.9.3
- **lock_init** Callback, see Section 34.7.9
- OMPT **mutex** Type, see Section 33.20

28.1.2 omp_init_nest_lock Routine

Name: omp_init_nest_lock Category: subroutine	Properties: all-contention-group-tasks-binding, lock-initializing, nestable-lock

Arguments

Name	Type	Properties
nvar	nest_lock	C/C++ pointer, omp

Prototypes

C / C++

```
void omp_init_nest_lock(omp_nest_lock_t *nvar);
```

C / C++

Fortran

```
subroutine omp_init_nest_lock(nvar)
  integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Effect

The **omp_init_nest_lock** routine is a lock-initializing routine.

Execution Model Events

The *nest-lock-init* event occurs in a thread that executes an **omp_init_nest_lock** region after initialization of the lock, but before it finishes the region.

Tool Callbacks

A thread dispatches a registered **lock_init** callback with **omp_sync_hint_none** as the *hint* argument and **ompt_mutex_nest_lock** as the *kind* argument for each occurrence of a *nest-lock-init* event in that thread. This callback occurs in the task that encounters the routine.

Cross References

- **lock_init** Callback, see Section 34.7.9
- OMPT **mutex** Type, see Section 33.20
- OpenMP **nest_lock** Type, see Section 20.9.4

28.1.3 omp_init_lock_with_hint Routine

Name: omp_init_lock_with_hint Category: subroutine	Properties: all-contention-group-tasks-binding, lock-initializing, simple-lock

Arguments

Name	Type	Properties
svar	lock	C/C++ pointer, omp
hint	sync_hint	omp

Prototypes

C / C++

```c
void omp_init_lock_with_hint(omp_lock_t *svar,
  omp_sync_hint_t hint);
```

C / C++

Fortran

```fortran
subroutine omp_init_lock_with_hint(svar, hint)
  integer (kind=omp_lock_kind) svar
  integer (kind=omp_sync_hint_kind) hint
```

Fortran

Effect

The **omp_init_lock_with_hint** routine is a lock-initializing routine.

Execution Model Events

The *lock-init-with-hint* event occurs in a thread that executes an **omp_init_lock_with_hint** region after initialization of the lock, but before it finishes the region.

Tool Callbacks

A thread dispatches a registered **lock_init** callback with the same value for its *hint* argument as the *hint* argument of the call to **omp_init_lock_with_hint** and **ompt_mutex_lock** as the *kind* argument for each occurrence of a *lock-init-with-hint* event in that thread. This callback occurs in the task that encounters the routine.

Cross References

- OpenMP **lock** Type, see Section 20.9.3
- **lock_init** Callback, see Section 34.7.9
- OMPT **mutex** Type, see Section 33.20
- OpenMP **sync_hint** Type, see Section 20.9.5

28.1.4 omp_init_nest_lock_with_hint Routine

Name: omp_init_nest_lock_with_hint **Category:** subroutine	**Properties:** all-contention-group-tasks-binding, lock-initializing, nestable-lock

Arguments

Name	Type	Properties
nvar	nest_lock	C/C++ pointer, omp
hint	sync_hint	omp

Prototypes

C / C++

```
void omp_init_nest_lock_with_hint(omp_nest_lock_t *nvar,
  omp_sync_hint_t hint);
```

C / C++

Fortran

```
subroutine omp_init_nest_lock_with_hint(nvar, hint)
  integer (kind=omp_nest_lock_kind) nvar
  integer (kind=omp_sync_hint_kind) hint
```

Fortran

Effect

The **omp_init_nest_lock_with_hint** routine is a lock-initializing routine.

Execution Model Events

The *nest-lock-init-with-hint* event occurs in a thread that executes an **omp_init_nest_lock** region after initialization of the lock, but before it finishes the region.

Tool Callbacks

A thread dispatches a registered **lock_init** callback with the same value for its *hint* argument as the *hint* argument of the call to **omp_init_nest_lock_with_hint** and **ompt_mutex_nest_lock** as the *kind* argument for each occurrence of a *nest-lock-init-with-hint* event in that thread This callback occurs in the task that encounters the routine.

Cross References

- **lock_init** Callback, see Section 34.7.9
- OMPT **mutex** Type, see Section 33.20
- OpenMP **nest_lock** Type, see Section 20.9.4
- OpenMP **sync_hint** Type, see Section 20.9.5

28.2 Lock Destroying Routines

Lock-destroying routines are routines with the lock-destroying property. These routines deactivate the lock by setting it to the uninitialized state.

Restrictions

Restrictions to lock-destroying routines are as follows:

- A lock-destroying routine must not access a lock that is not in the unlocked state.

28.2.1 `omp_destroy_lock` Routine

Name: `omp_destroy_lock` **Category:** subroutine	**Properties:** all-contention-group-tasks-binding, lock-destroying, simple-lock

Arguments

Name	Type	Properties
svar	lock	C/C++ pointer, omp

Prototypes

C / C++

```
void omp_destroy_lock(omp_lock_t *svar);
```

C / C++

Fortran

```
subroutine omp_destroy_lock(svar)
  integer (kind=omp_lock_kind) svar
```

Fortran

Effect

The **`omp_destroy_lock`** routine is a lock-destroying routine.

Execution Model Events

The *lock-destroy* event occurs in a thread that executes an **`omp_destroy_lock`** region before it finishes the region.

Tool Callbacks

A thread dispatches a registered **`lock_destroy`** callback with **`ompt_mutex_lock`** as the *kind* argument for each occurrence of a *lock-destroy* event in that thread. This callback occurs in the task that encounters the routine.

Cross References

- OpenMP **lock** Type, see Section 20.9.3
- **lock_destroy** Callback, see Section 34.7.11
- OMPT **mutex** Type, see Section 33.20

28.2.2 omp_destroy_nest_lock Routine

Name: **omp_destroy_nest_lock** **Category:** subroutine	**Properties:** all-contention-group-tasks-binding, lock-destroying, nestable-lock

Arguments

Name	Type	Properties
nvar	nest_lock	C/C++ pointer, omp

Prototypes

C / C++

```
void omp_destroy_nest_lock(omp_nest_lock_t *nvar);
```

C / C++

Fortran

```
subroutine omp_destroy_nest_lock(nvar)
  integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Effect

The **omp_destroy_nest_lock** routine is a lock-destroying routine.

Execution Model Events

The *nest-lock-destroy* event occurs in a thread that executes an **omp_destroy_nest_lock** region before it finishes the region.

Tool Callbacks

A thread dispatches a registered **lock_destroy** callback with **ompt_mutex_nest_lock** as the *kind* argument for each occurrence of a *nest-lock-destroy* event in that thread. This occurs in the task that encounters the routine.

Cross References

- **lock_destroy** Callback, see Section 34.7.11
- OMPT **mutex** Type, see Section 33.20
- OpenMP **nest_lock** Type, see Section 20.9.4

28.3 Lock Acquiring Routines

Lock-acquiring routines are routines with the lock-acquiring property. These routines provide a means of setting locks. The encountering task region behaves as if it was suspended until the lock can be acquired by this task.

Note – The semantics of lock-acquiring routine are specified *as if* they serialize execution of the region guarded by the lock. However, implementations may implement them in other ways provided that the isolation properties are respected so that the actual execution delivers a result that could arise from some serialization.

Restrictions

Restrictions to lock-acquiring routines are as follows:

- A lock-acquiring routine must not access a lock that is in the uninitialized state.

28.3.1 omp_set_lock Routine

Name: `omp_set_lock` **Category:** subroutine	**Properties:** all-contention-group-tasks-binding, lock-acquiring, simple-lock

Arguments

Name	Type	Properties
svar	lock	C/C++ pointer, omp

Prototypes

C / C++

```
void omp_set_lock(omp_lock_t *svar);
```

C / C++

Fortran

```
subroutine omp_set_lock(svar)
  integer (kind=omp_lock_kind) svar
```

Fortran

Effect

A simple lock is available when it is in the unlocked state. Ownership of the lock is granted to the task that executes the routine.

Execution Model Events

The *lock-acquire* event occurs in a thread that executes an **omp_set_lock** region before the associated lock is requested. The *lock-acquired* event occurs in a thread that executes an **omp_set_lock** region after it acquires the associated lock but before it finishes the region.

Tool Callbacks

A thread dispatches a registered **mutex_acquire** callback for each occurrence of a *lock-acquire* event in that thread. A thread dispatches a registered **mutex_acquired** callback for each occurrence of a *lock-acquired* event in that thread. These callbacks occur in the task that encounters the **omp_set_lock** routine and their *kind* argument is **ompt_mutex_lock**.

Restrictions

Restrictions to the **omp_set_lock** routine are as follows:

- A task must not already own the lock that it accesses with a call to **omp_set_lock** (or deadlock will result).

Cross References

- OpenMP **lock** Type, see Section 20.9.3
- OMPT **mutex** Type, see Section 33.20
- **mutex_acquire** Callback, see Section 34.7.8
- **mutex_acquired** Callback, see Section 34.7.12

28.3.2 omp_set_nest_lock Routine

Name: omp_set_nest_lock Category: subroutine	Properties: all-contention-group-tasks-binding, lock-acquiring, nestable-lock

Arguments

Name	Type	Properties
nvar	nest_lock	C/C++ pointer, omp

Prototypes

C / C++

```
void omp_set_nest_lock(omp_nest_lock_t *nvar);
```

C / C++

Fortran

```
subroutine omp_set_nest_lock(nvar)
  integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Effect

A nestable lock is available if it is in the unlocked state or if it is already owned by the task that executes the routine. The task that executes the routine is granted, or retains, ownership of the lock, and the nesting count for the lock is incremented.

Execution Model Events

The *nest-lock-acquire* event occurs in a thread that executes an **omp_set_nest_lock** region before the associated lock is requested. The *nest-lock-acquired* event occurs in a thread that executes an **omp_set_nest_lock** region if the task did not already own the lock, after it acquires the associated lock but before it finishes the region. The *nest-lock-owned* event occurs in a task when it already owns the lock and executes an **omp_set_nest_lock** region. The *nest-lock-owned* event occurs after the nesting count is incremented but before the task finishes the region.

Tool Callbacks

A thread dispatches a registered **mutex_acquire** callback for each occurrence of a *nest-lock-acquire* event in that thread. A thread dispatches a registered **mutex_acquired** callback for each occurrence of a *nest-lock-acquired* event in that thread. A thread dispatches a registered **nest_lock** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *nest-lock-owned* event in that thread. These callbacks occur in the task that encounters the **omp_set_nest_lock** routine and their *kind* argument is **ompt_mutex_nest_lock**.

Cross References

- OMPT **mutex** Type, see Section 33.20
- **mutex_acquire** Callback, see Section 34.7.8
- **mutex_acquired** Callback, see Section 34.7.12
- **nest_lock** Callback, see Section 34.7.14
- OpenMP **nest_lock** Type, see Section 20.9.4
- OMPT **scope_endpoint** Type, see Section 33.27

28.4 Lock Releasing Routines

Lock-releasing routines are routines with the lock-releasing property. These routines provide a means of unsetting locks. If the effect of a lock-releasing routine changes the lock state to the unlocked state and one or more task regions were effectively suspended because the lock was unavailable, the effect is that one task is chosen and given ownership of the lock.

Restrictions

Restrictions to lock-releasing routines are as follows:

- A lock-releasing routine must not access a lock that is not in the locked state.
- A lock-releasing routine must not access a lock that is owned by a task other than the encountering task.

28.4.1 omp_unset_lock Routine

Name: **omp_unset_lock** **Category:** subroutine	**Properties:** all-contention-group-tasks-binding, lock-releasing, simple-lock

Arguments

Name	Type	Properties
svar	lock	C/C++ pointer, omp

Prototypes

C / C++

```
void omp_unset_lock(omp_lock_t *svar);
```

C / C++

Fortran

```
subroutine omp_unset_lock(svar)
  integer (kind=omp_lock_kind) svar
```

Fortran

Effect

The **omp_unset_lock** routine changes the lock state to the unlocked state.

Execution Model Events

The *lock-release* event occurs in a thread that executes an **omp_unset_lock** region after it releases the associated lock but before it finishes the region.

Tool Callbacks

A thread dispatches a registered **mutex_released** callback with **ompt_mutex_lock** as the *kind* argument for each occurrence of a *lock-release* event in that thread. This callback occurs in the encountering task.

Cross References

- OpenMP **lock** Type, see Section 20.9.3
- OMPT **mutex** Type, see Section 33.20
- **mutex_released** Callback, see Section 34.7.13

28.4.2 omp_unset_nest_lock Routine

Name: omp_unset_nest_lock Category: subroutine	Properties: all-contention-group-tasks-binding, lock-releasing, nestable-lock

Arguments

Name	Type	Properties
nvar	nest_lock	C/C++ pointer, omp

Prototypes

C / C++

```
void omp_unset_nest_lock(omp_nest_lock_t *nvar);
```

C / C++

Fortran

```
subroutine omp_unset_nest_lock(nvar)
  integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Effect

The **omp_unset_nest_lock** routine decrements the nesting count and, if the resulting nesting count is zero, changes the lock state to the unlocked state.

Execution Model Events

The *nest-lock-release* event occurs in a thread that executes an **omp_unset_nest_lock** region after it releases the associated lock but before it finishes the region. The *nest-lock-held* event occurs in a thread that executes an **omp_unset_nest_lock** region before it finishes the region when the thread still owns the lock after the nesting count is decremented.

Tool Callbacks

A thread dispatches a registered **mutex_released** callback with **ompt_mutex_nest_lock** as the *kind* argument for each occurrence of a *nest-lock-release* event in that thread. A thread dispatches a registered **nest_lock** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of a *nest-lock-held* event in that thread. These callbacks occur in the encountering task.

Cross References

- OMPT **mutex** Type, see Section 33.20
- **mutex_released** Callback, see Section 34.7.13
- **nest_lock** Callback, see Section 34.7.14
- OpenMP **nest_lock** Type, see Section 20.9.4
- OMPT **scope_endpoint** Type, see Section 33.27

28.5 Lock Testing Routines

Lock-testing routines are routines with the lock-testing property. These routines attempt to acquire a lock in the same manner as lock-acquiring routines, except that they do not suspend execution of the encountering task

Restrictions

Restrictions on lock-testing routines are as follows.

- A lock-testing routine must not access a lock that is in the uninitialized state.

28.5.1 `omp_test_lock` Routine

Name: `omp_test_lock` **Category:** function	**Properties:** all-contention-group-tasks-binding, lock-testing, simple-lock

Return Type and Arguments

Name	Type	Properties
<return type>	logical	*default*
svar	lock	C/C++ pointer, omp

Prototypes

C / C++

```
int omp_test_lock(omp_lock_t *svar);
```

C / C++

Fortran

```
logical function omp_test_lock(svar)
  integer (kind=omp_lock_kind) svar
```

Fortran

Effect

The **`omp_test_lock`** routine returns *true* if it successfully acquires the lock; otherwise, it returns *false*.

Execution Model Events

The *lock-test* event occurs in a thread that executes an **`omp_test_lock`** region before the associated lock is tested. The *lock-test-acquired* event occurs in a thread that executes an **`omp_test_lock`** region before it finishes the region if the associated lock was acquired.

Tool Callbacks

A thread dispatches a registered **`mutex_acquire`** callback for each occurrence of a *lock-test* event in that thread. A thread dispatches a registered **`mutex_acquired`** callback for each occurrence of a *lock-test-acquired* event in that thread. These callbacks occur in the encountering task and their *kind* argument is **`ompt_mutex_test_lock`**.

Restrictions

Restrictions to **omp_test_lock** routines are as follows:

- An **omp_test_lock** routine must not access a lock that is already owned by the encountering task.

Cross References

- OpenMP **lock** Type, see Section 20.9.3
- OMPT **mutex** Type, see Section 33.20
- **mutex_acquire** Callback, see Section 34.7.8
- **mutex_acquired** Callback, see Section 34.7.12

28.5.2 omp_test_nest_lock Routine

Name: **omp_test_nest_lock** Category: function	**Properties:** all-contention-group-tasks-binding, lock-testing, nestable-lock

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
nvar	nest_lock	C/C++ pointer, omp

Prototypes

C / C++

```
int omp_test_nest_lock(omp_nest_lock_t *nvar);
```

C / C++

Fortran

```
integer function omp_test_nest_lock(nvar)
  integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Effect

The **omp_test_nest_lock** routine returns the new nesting count if it successfully sets the lock; otherwise, it returns zero.

Execution Model Events

The *nest-lock-test* event occurs in a thread that executes an **omp_test_nest_lock** region before the associated lock is tested. The *nest-lock-test-acquired* event occurs in a thread that executes an **omp_test_nest_lock** region before it finishes the region if the associated lock was acquired and the thread did not already own the lock. The *nest-lock-owned* event occurs in a thread that executes an **omp_test_nest_lock** region before it finishes the region after the nesting count is incremented if the thread already owned the lock.

Tool Callbacks

A thread dispatches a registered **mutex_acquire** callback for each occurrence of a *nest-lock-test* event in that thread. A thread dispatches a registered **mutex_acquired** callback for each occurrence of a *nest-lock-test-acquired* event in that thread. A thread dispatches a registered **nest_lock** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *nest-lock-owned* event in that thread. These callbacks occur in the encountering task and their *kind* argument is **ompt_mutex_test_nest_lock**.

Cross References

- OMPT **mutex** Type, see Section 33.20
- **mutex_acquire** Callback, see Section 34.7.8
- **mutex_acquired** Callback, see Section 34.7.12
- **nest_lock** Callback, see Section 34.7.14
- OpenMP **nest_lock** Type, see Section 20.9.4
- OMPT **scope_endpoint** Type, see Section 33.27

29 Thread Affinity Routines

This chapter describes routines that specify and obtain information about thread affinity policies, which govern the placement of threads in the execution environment of OpenMP programs.

29.1 omp_get_proc_bind Routine

Name: omp_get_proc_bind **Category:** function	**Properties:** ICV-retrieving

Return Type

Name	Type	Properties
<*return type*>	proc_bind	*default*

Prototypes

C / C++

```
omp_proc_bind_t omp_get_proc_bind(void);
```

C / C++

Fortran

```
integer (kind=omp_proc_bind_kind) function omp_get_proc_bind()
```

Fortran

Effect

The effect of this routine is to return the value of the first element of the *bind-var* ICV of the current task, which will be used for the subsequent nested **parallel** regions that do not specify a **proc_bind** clause. See Section 12.1.3 for the rules that govern the thread affinity policy.

Cross References

- Controlling OpenMP Thread Affinity, see Section 12.1.3
- *bind-var* ICV, see Table 3.1
- **parallel** Construct, see Section 12.1
- OpenMP **proc_bind** Type, see Section 20.10.1

29.2 omp_get_num_places Routine

Name: `omp_get_num_places` **Category:** function	**Properties:** all-device-threads-binding

Return Type

Name	Type	Properties
<return type>	integer	*default*

Prototypes

C / C++

```c
int omp_get_num_places(void);
```

C / C++

Fortran

```fortran
integer function omp_get_num_places()
```

Fortran

Effect

The **omp_get_num_places** routine returns the number of places in the place list. This value is equivalent to the number of places in the *place-partition-var* ICV in the execution environment of the initial task.

Cross References

- *place-partition-var* ICV, see Table 3.1

29.3 omp_get_place_num_procs Routine

Name: `omp_get_place_num_procs` **Category:** function	**Properties:** all-device-threads-binding, ICV-retrieving

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
place_num	integer	*default*

Prototypes

C / C++

```c
int omp_get_place_num_procs(int place_num);
```

C / C++

Fortran

```fortran
integer function omp_get_place_num_procs(place_num)
  integer place_num
```

Fortran

Effect

The **omp_get_place_num_procs** routine returns the number of processors associated with the place numbered *place_num* as per the *place-partition-var* ICV. The routine returns zero when *place_num* is negative or is greater than or equal to the value returned by **omp_get_num_places**.

Cross References

- *place-partition-var* ICV, see Table 3.1
- **omp_get_num_places** Routine, see Section 29.2

29.4 omp_get_place_proc_ids Routine

Name: **omp_get_place_proc_ids** Category: subroutine	**Properties:** all-device-threads-binding, ICV-retrieving

Arguments

Name	Type	Properties
place_num	integer	*default*
ids	integer	pointer

Prototypes

C / C++

```c
void omp_get_place_proc_ids(int place_num, int *ids);
```

C / C++

Fortran

```fortran
subroutine omp_get_place_proc_ids(place_num, ids)
  integer place_num, ids(*)
```

Fortran

Effect

The **omp_get_place_proc_ids** routine returns the numerical identifiers of each processor associated with the place numbered *place_num* as per the *place-partition-var* ICV. The numerical identifiers are non-negative and their meaning is implementation defined. The numerical identifiers are returned in the array *ids* and their order in the array is implementation defined. The array must be sufficiently large to contain **omp_get_place_num_procs**(*place_num*) integers; otherwise, the behavior is unspecified. The routine has no effect when *place_num* has a negative value or a value greater than or equal to **omp_get_num_places**.

Cross References

- **OMP_PLACES**, see Section 4.1.6
- **omp_get_num_places** Routine, see Section 29.2
- **omp_get_place_num_procs** Routine, see Section 29.3

29.5 omp_get_place_num Routine

Name: omp_get_place_num Category: function	Properties: *default*

Return Type

Name	Type	Properties
<return type>	integer	*default*

Prototypes

C / C++

```
int omp_get_place_num(void);
```

C / C++

Fortran

```
integer function omp_get_place_num()
```

Fortran

Effect

When the encountering thread is bound to a place, the **omp_get_place_num** routine returns the place number associated with the thread. The returned value is between zero and one less than the value returned by **omp_get_num_places**, inclusive. When the encountering thread is not bound to a place, the routine returns -1.

Cross References

- **omp_get_num_places** Routine, see Section 29.2

29.6 omp_get_partition_num_places Routine

Name: omp_get_partition_num_places Category: function	Properties: ICV-retrieving

Return Type

Name	Type	Properties
<return type>	integer	*default*

Prototypes

C / C++

```
int omp_get_partition_num_places(void);
```

C / C++

Fortran

```
integer function omp_get_partition_num_places()
```

Fortran

Effect

The **omp_get_partition_num_places** routine returns the number of places in the *place-partition-var* ICV.

Cross References

- *place-partition-var* ICV, see Table 3.1

29.7 omp_get_partition_place_nums Routine

Name: omp_get_partition_place_nums **Category:** subroutine	**Properties:** ICV-retrieving

Arguments

Name	Type	Properties
place_nums	integer	pointer

Prototypes

C / C++

```
void omp_get_partition_place_nums(int *place_nums);
```

C / C++

Fortran

```
subroutine omp_get_partition_place_nums(place_nums)
  integer place_nums(*)
```

Fortran

Effect

The **omp_get_partition_place_nums** routine returns the list of place numbers that correspond to the places in the *place-partition-var* ICV of the innermost implicit task. The array must be sufficiently large to contain **omp_get_partition_num_places** integers; otherwise, the behavior is unspecified.

Cross References

- *place-partition-var* ICV, see Table 3.1
- **omp_get_partition_num_places** Routine, see Section 29.6

29.8 omp_set_affinity_format Routine

Name: omp_set_affinity_format **Category:** subroutine	**Properties:** ICV-modifying

Arguments

Name	Type	Properties
format	char	pointer, intent(in)

Prototypes

C / C++

```
void omp_set_affinity_format(const char *format);
```

C / C++

Fortran

```
subroutine omp_set_affinity_format(format)
  character(len=*), intent(in) :: format
```

Fortran

Effect

The **omp_set_affinity_format** routine sets the affinity format to be used on the device by setting the value of the *affinity-format-var* ICV. The value of the ICV is set by copying the character string specified by the *format* argument into the ICV on the current device.

This routine has the described effect only when called from a sequential part of the program. When called from within a **parallel** or **teams** region, the effect of this routine is implementation defined.

When called from a sequential part of the program, the binding thread set for an **omp_set_affinity_format** region is the encountering thread. When called from within any **parallel** or **teams** region, the binding thread set (and binding region, if required) for the **omp_set_affinity_format** region is implementation defined.

Restrictions

Restrictions to the **omp_set_affinity_format** routine are as follows:

- When called from within a **target** region the effect is unspecified.

Cross References

- **OMP_AFFINITY_FORMAT**, see Section 4.3.5
- **OMP_DISPLAY_AFFINITY**, see Section 4.3.4
- Controlling OpenMP Thread Affinity, see Section 12.1.3
- *affinity-format-var* ICV, see Table 3.1
- **parallel** Construct, see Section 12.1
- **teams** Construct, see Section 12.2

29.9 omp_get_affinity_format Routine

Name: omp_get_affinity_format Category: function	Properties: ICV-retrieving

Return Type and Arguments

Name	Type	Properties
<return type>	size_t	*default*
buffer	char	pointer, intent(out)
size	size_t	*default*

Prototypes

C / C++

```
size_t omp_get_affinity_format(char *buffer, size_t size);
```

C / C++

Fortran

```
integer function omp_get_affinity_format(buffer)
  character(len=*), intent(out) :: buffer
```

Fortran

Effect

C / C++

The **omp_get_affinity_format** routine returns the number of characters in the *affinity-format-var* ICV on the current device, excluding the terminating null byte (**'\0'**) and, if *size* is non-zero, writes the value of the *affinity-format-var* ICV on the current device to *buffer* followed by a null byte. If the return value is larger or equal to *size*, the affinity format specification is truncated, with the terminating null byte stored to ***buffer*** **[*size*-1]**. If *size* is zero, nothing is stored and *buffer* may be NULL.

C / C++

Fortran

The **omp_get_affinity_format** routine returns the number of characters that are required to hold the *affinity-format-var* ICV on the current device and writes the value of the *affinity-format-var* ICV on the current device to *buffer*. If the return value is larger than **len(*buffer*)**, the affinity format specification is truncated.

Fortran

If the *buffer* argument does not conform to the specified format then the result is implementation defined.

When called from a sequential part of the program, the binding thread set for an **omp_get_affinity_format** region is the encountering thread. When called from within any **parallel** or **teams** region, the binding thread set (and binding region, if required) for the **omp_get_affinity_format** region is implementation defined.

Restrictions

Restrictions to the **omp_get_affinity_format** routine are as follows:

- When called from within a **target** region the effect is unspecified.

Cross References

- *affinity-format-var* ICV, see Table 3.1
- **parallel** Construct, see Section 12.1
- **target** Construct, see Section 15.8
- **teams** Construct, see Section 12.2

29.10 omp_display_affinity Routine

Name: omp_display_affinity Category: subroutine	Properties: *default*

Arguments

Name	Type	Properties
format	char	pointer, intent(in)

Prototypes

C / C++

```
void omp_display_affinity(const char *format);
```

C / C++

Fortran

```
subroutine omp_display_affinity(format)
  character(len=*), intent(in) :: format
```

Fortran

Effect

The **omp_display_affinity** routine prints the thread affinity information of the encountering thread in the format specified by the *format* argument, followed by a *new-line*. If the *format* is NULL (for C/C++) or a zero-length string (for Fortran and C/C++), the value of the *affinity-format-var* ICV is used. If the *format* argument does not conform to the specified format then the result is implementation defined.

Restrictions

Restrictions to the **omp_display_affinity** routine are as follows:

- When called from within a **target** region the effect is unspecified.

Cross References

- *affinity-format-var* ICV, see Table 3.1
- **target** Construct, see Section 15.8

29.11 omp_capture_affinity Routine

Name: omp_capture_affinity **Category:** function	**Properties:** *default*

Return Type and Arguments

Name	Type	Properties
<return type>	size_t	*default*
buffer	char	pointer, intent(out)
size	size_t	*default*
format	char	pointer, intent(in)

Prototypes

C / C++

```
size_t omp_capture_affinity(char *buffer, size_t size,
  const char *format);
```

C / C++

Fortran

```
integer function omp_capture_affinity(buffer, format)
  character(len=*), intent(out) :: buffer
  character(len=*), intent(in) :: format
```

Fortran

Effect

C / C++

The **omp_capture_affinity** routine returns the number of characters in the entire thread affinity information string excluding the terminating null byte (**'\0'**). If *size* is non-zero, it writes the thread affinity information of the encountering thread in the format specified by the *format* argument into the character string *buffer* followed by a null byte. If the return value is larger or equal to *size*, the thread affinity information string is truncated, with the terminating null byte stored to ***buffer* [*size*−1]**. If *size* is zero, nothing is stored and *buffer* may be NULL. If the *format* is NULL or a zero-length string, the value of the *affinity-format-var* ICV is used.

C / C++

Fortran

The **omp_capture_affinity** routine returns the number of characters required to hold the entire thread affinity information string and prints the thread affinity information of the encountering thread into the character string *buffer* with the size of **len(*buffer*)** in the format specified by the *format* argument. If the *format* is a zero-length string, the value of the *affinity-format-var* ICV is used. If the return value is larger than **len(*buffer*)**, the thread affinity information string is truncated. If the *format* is a zero-length string, the value of the *affinity-format-var* ICV is used.

Fortran

If the *format* argument does not conform to the specified format then the result is implementation defined.

Restrictions

Restrictions to the **omp_capture_affinity** routine are as follows:

- When called from within a **target** region the effect is unspecified.

Cross References

- *affinity-format-var* ICV, see Table 3.1
- **target** Construct, see Section 15.8

30 Execution Control Routines

This chapter describes the OpenMP API routines that control the execution state of the OpenMP implementation and provide information about that state. These routines include:

- Routines that monitor and control cancellation;
- Resource-relinquishing routines that free resources used by the OpenMP program;
- Routines that support timing measurements of OpenMP programs; and
- The environment display routine that displays the initial values of ICVs.

30.1 `omp_get_cancellation` Routine

Name: `omp_get_cancellation` **Category:** function	**Properties:** ICV-retrieving

Return Type

Name	Type	Properties
<*return type*>	logical	*default*

Prototypes

C / C++

```
int omp_get_cancellation(void);
```

C / C++

Fortran

```
logical function omp_get_cancellation()
```

Fortran

Effect

The **`omp_get_cancellation`** routine returns the value of the *cancel-var* ICV. Thus, it returns *true* if cancellation is enabled and otherwise it returns *false*.

Cross References

- *cancel-var* ICV, see Table 3.1

30.2 Resource Relinquishing Routines

This section describes routines that have the resource-relinquishing property. Each resource-relinquishing routine region implies a barrier. Each resource-relinquishing routine returns zero in case of success, and non-zero otherwise.

Tool Callbacks

If the tool is not allowed to interact with the specified device after encountering the resource-relinquishing routine, then the runtime must call the tool finalizer for that device.

Restrictions

Restrictions to resource-relinquishing routines are as follows:

- A resource-relinquishing routine region may not be nested in any explicit region.
- A resource-relinquishing routine may only be called when all explicit tasks that do not bind to the implicit parallel region to which the encountering thread binds have finalized execution.

30.2.1 `omp_pause_resource` Routine

Name: `omp_pause_resource` **Category:** function	**Properties:** all-tasks-binding, resource-relinquishing

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
kind	pause_resource	*default*
device_num	integer	*default*

Prototypes

C / C++

```
int omp_pause_resource(omp_pause_resource_t kind, int device_num);
```

C / C++

Fortran

```
integer function omp_pause_resource(kind, device_num)
  integer (kind=omp_pause_resource_kind) kind
  integer device_num
```

Fortran

Effect

The **`omp_pause_resource`** routine allows the runtime to relinquish resources used by OpenMP on the specified device. The *device_num* argument indicates the device that will be paused. If the device number has the value **`omp_invalid_device`**, runtime error termination is performed.

The binding task set for a **omp_pause_resource** routine region is all tasks on the specified device. That is, this routines has the all-device-tasks binding property. If **omp_pause_stop_tool** is specified for a non-host device, the effect is the same as for **omp_pause_hard** and (unlike for the host device) does not shutdown the OMPT interface.

Restrictions

Restrictions to the **omp_pause_resource** routine are as follows:

- The *device_num* argument must be a conforming device number.

Cross References

- Predefined Identifiers, see Section 20.1
- OpenMP **pause_resource** Type, see Section 20.11.1

30.2.2 omp_pause_resource_all Routine

Name: omp_pause_resource_all **Category:** function	**Properties:** all-tasks-binding, resource-relinquishing

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
kind	pause_resource	*default*

Prototypes

C / C++

```c
int omp_pause_resource_all(omp_pause_resource_t kind);
```

C / C++

Fortran

```fortran
integer function omp_pause_resource_all(kind)
  integer (kind=omp_pause_resource_kind) kind
```

Fortran

Effect

The **omp_pause_resource_all** routine allows the runtime to relinquish resources used by OpenMP on all devices. It is equivalent to calling the **omp_pause_resource** routine once for each available device, including the host device. The binding task set for a **omp_pause_resource_all** routine region is all tasks in the OpenMP program. That is, this routine has the all-tasks binding property.

Cross References

- **omp_pause_resource** Routine, see Section 30.2.1
- OpenMP **pause_resource** Type, see Section 20.11.1

30.3 Timing Routines

This section describes routines that support a portable wall clock timer.

30.3.1 omp_get_wtime Routine

Name: omp_get_wtime **Category:** function	**Properties:** *default*

Return Type

Name	Type	Properties
<return type>	double	*default*

Prototypes

C / C++

```
double omp_get_wtime(void);
```

C / C++

Fortran

```
double precision function omp_get_wtime()
```

Fortran

Effect

The **omp_get_wtime** routine returns a value equal to the elapsed wall clock time in seconds since some *time-in-the-past*. The actual *time-in-the-past* is arbitrary, but it is guaranteed not to change during the execution of an OpenMP program. The time returned is a *per-thread time*, so it is not required to be globally consistent across all threads that participate in an OpenMP program.

30.3.2 omp_get_wtick Routine

Name: omp_get_wtick **Category:** function	**Properties:** *default*

Return Type

Name	Type	Properties
<return type>	double	*default*

Prototypes

C / C++

```
double omp_get_wtick(void);
```

C / C++

Fortran

```
double precision function omp_get_wtick()
```

Fortran

Effect

The **omp_get_wtick** routine returns the precision of the timer used by **omp_get_wtime** as a value equal to the number of seconds between successive clock ticks. The return value of the **omp_get_wtick** routine is not guaranteed to be consistent across any set of threads.

Cross References

- **omp_get_wtime** Routine, see Section 30.3.1

30.4 omp_display_env Routine

Name: **omp_display_env** **Category:** subroutine	**Properties:** *default*

Arguments

Name	Type	Properties
verbose	logical	intent(in)

Prototypes

C / C++

```
void omp_display_env(int verbose);
```

C / C++

Fortran

```
subroutine omp_display_env(verbose)
  logical, intent(in) :: verbose
```

Fortran

Effect

Each time that the **omp_display_env** routine is invoked, the runtime system prints the OpenMP version number and the initial values of the ICVs associated with the environment variables described in Chapter 4. The displayed values are the values of the ICVs after they have been modified according to the environment variable settings and before the execution of any construct or routine.

The display begins with "OPENMP DISPLAY ENVIRONMENT BEGIN", followed by the **_OPENMP** version macro (or the **openmp_version** predefined identifier for Fortran) and ICV values, in the format *NAME* '=' *VALUE*. *NAME* corresponds to the macro or environment variable name, prepended with a bracketed *DEVICE*. *VALUE* corresponds to the value of the macro or ICV associated with this environment variable. Values are enclosed in single quotes. *DEVICE* corresponds to a comma-separated list of the devices on which the value of the ICV is applied. It is **host** if the device is the host device; **device** if the ICV applies to all non-host devices; **all** if the ICV has global scope or the value applies to the host device and all non-host devices; **dev**, a space, and the device number if it applies to a specific non-host devices. Instead of a single number a range can also be specified using the first and last device number separated by a hyphen. Whether

ICVs with the same value are combined or displayed in multiple lines is implementation defined. The display is terminated with "OPENMP DISPLAY ENVIRONMENT END".

If the *verbose* argument evaluates to *false*, the runtime displays the OpenMP version number defined by the **_OPENMP** version macro (or the **openmp_version** predefined identifier for Fortran) value and the initial ICV values for the environment variables listed in Chapter 4. If the *verbose* argument evaluates to *true*, the runtime may also display the values of vendor-specific ICVs that may be modified by vendor-specific environment variables.

Example output:

```
OPENMP DISPLAY ENVIRONMENT BEGIN
  _OPENMP='202411'
  [dev 1] OMP_SCHEDULE='GUIDED,4'
  [host] OMP_NUM_THREADS='4,3,2'
  [device] OMP_NUM_THREADS='2'
  [host, dev 2] OMP_DYNAMIC='TRUE'
  [dev 2-3, dev 5] OMP_DYNAMIC='FALSE'
  [all] OMP_WAIT_POLICY='ACTIVE'
  [host] OMP_PLACES='{0:4},{4:4},{8:4},{12:4}'
  ...
OPENMP DISPLAY ENVIRONMENT END
```

Restrictions

Restrictions to the **omp_display_env** routine are as follows:

- When called from within a **target** region the effect is unspecified.

Cross References

- Predefined Identifiers, see Section 20.1

31 Tool Support Routines

This chapter describes the OpenMP API routines that support the use of OpenMP tool interfaces.

31.1 omp_control_tool Routine

Name: `omp_control_tool` **Category:** function	**Properties:** *default*

Return Type and Arguments

Name	Type	Properties
<return type>	control_tool_result	*default*
command	control_tool	omp
modifier	integer	*default*
arg	void	C/C++ pointer

Prototypes

C / C++

```c
omp_control_tool_result_t omp_control_tool(
  omp_control_tool_t command, int modifier, void *arg);
```

C / C++

Fortran

```fortran
integer (kind=omp_control_tool_result_kind) function &
  omp_control_tool(command, modifier)
  integer (kind=omp_control_tool_kind) command
  integer modifier
```

Fortran

Effect

An OpenMP program may use the **omp_control_tool** routine to pass commands to a tool. An OpenMP program can use the routine to request: that a tool starts or restarts data collection when a code region of interest is encountered; that a tool pauses data collection when leaving the region of interest; that a tool flushes any data that it has collected so far; or that a tool ends data collection. Additionally, the **omp_control_tool** routine can be used to pass tool-specific commands to a particular tool.

Any values for *modifier* and *arg* are tool defined.

If the OMPT interface state is OMPT inactive, the OpenMP implementation returns **omp_control_tool_notool**. If the OMPT interface state is OMPT active, but no callback is registered for the *tool-control* event, the OpenMP implementation returns **omp_control_tool_nocallback**. An OpenMP implementation may return other implementation defined negative values strictly smaller than -64; an OpenMP program may assume that any negative return value indicates that a tool has not received the command. A return value of **omp_control_tool_success** indicates that the tool has performed the specified command. A return value of **omp_control_tool_ignored** indicates that the tool has ignored the specified command. A tool may return other positive values strictly greater than 64 that are tool defined.

Execution Model Events

The *tool-control* event occurs in the encountering thread inside the corresponding region.

Tool Callbacks

A thread dispatches a registered **control_tool** callback for each occurrence of a *tool-control* event. The callback executes in the context of the call that occurs in the user program. The callback may return any non-negative value, which will be returned to the OpenMP program by the OpenMP implementation as the return value of the **omp_control_tool** call that triggered the callback.

Arguments passed to the callback are those passed by the user to **omp_control_tool**. If the call is made in Fortran, the tool will be passed NULL as the third argument to the callback. If any of the standard commands is presented to a tool, the tool will ignore the *modifier* and *arg* argument values.

Restrictions

Restrictions on access to the state of an OpenMP first-party tool are as follows:

- An OpenMP program may access the tool state modified by an OMPT callback only by using **omp_control_tool**.

Cross References

- **control_tool** Callback, see Section 34.8
- OpenMP **control_tool** Type, see Section 20.12.1
- OpenMP **control_tool_result** Type, see Section 20.12.2
- OMPT Overview, see Chapter 32

Part IV

OMPT

32 OMPT Overview

This chapter provides an overview of OMPT, which is an interface for first-party tools. First-party tools are linked or loaded directly into the OpenMP program. OMPT defines mechanisms to initialize a tool, to examine thread state associated with a thread, to interpret the call stack of a thread, to receive notification about events, to trace activity on target devices, to assess implementation-dependent details of an OpenMP implementation (such as supported states and mutual exclusion implementations), and to control a tool from an OpenMP program.

32.1 OMPT Interfaces Definitions

C / C++

A compliant implementation must supply a set of definitions for the OMPT runtime entry points, OMPT callback signatures, and the OMPT types. These definitions, which are listed throughout this and the immediately following chapters, and their associated declarations shall be provided in a header file named `omp-tools.h`. In addition, the set of definitions may specify other implementation defined values.

The **`ompt_start_tool`** procedure is an external function with `C` linkage.

C / C++

32.2 Activating a First-Party Tool

To activate a tool, an OpenMP implementation first determines whether the tool should be initialized. If so, the OpenMP implementation invokes the OMPT-tool initializer of the tool, which enables the tool to prepare to monitor execution on the host device. The tool may then also arrange to monitor computation that executes on target devices. This section explains how the tool and an OpenMP implementation interact to accomplish these activities.

32.2.1 `ompt_start_tool` Procedure

Name: `ompt_start_tool` **Category:** function	**Properties:** C-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	start_tool_result	pointer, OMPT
omp_version	integer	unsigned
runtime_version	char	intent(in), pointer

Prototypes

C

```
ompt_start_tool_result_t  *ompt_start_tool(
  unsigned int omp_version, const char *runtime_version);
```

C

Semantics

For a tool to use the OMPT interface that an OpenMP implementation provides, the tool must define a globally-visible implementation of the **ompt_start_tool** procedure. The tool indicates that it will use the OMPT interface that an OpenMP implementation provides by returning a non-null pointer to a **start_tool_result** OMPT type structure from the **ompt_start_tool** implementation that it provides. The **start_tool_result** structure contains pointers to **initialize** and **finalize** callbacks as well as a tool data word that an OpenMP implementation must pass by reference to these callbacks. A tool may return NULL from **ompt_start_tool** to indicate that it will not use the OMPT interface in a particular execution.

A tool may use the *omp_version* argument to determine if it is compatible with the OMPT interface that the OpenMP implementation provides. The *omp_version* argument is the value of the **_OPENMP** version macro associated with the OpenMP implementation. This value identifies the version that an implementation supports, which specifies the version of the OMPT interface that it supports. The *runtime_version* argument is a version string that unambiguously identifies the OpenMP implementation.

If a tool returns a non-null pointer to a **start_tool_result** OMPT type structure, an OpenMP implementation will call the OMPT-tool initializer specified by the **initialize** field in this structure before beginning execution of any construct or completing execution of any routine; the OpenMP implementation will call the OMPT-tool finalizer specified by the **finalize** field in this structure when the OpenMP implementation shuts down.

Restrictions

Restrictions to **ompt_start_tool** procedures are as follows:

- The *runtime_version* argument must be an immutable string that is defined for the lifetime of a program execution.

Cross References

- **finalize** Callback, see Section 34.1.2
- **initialize** Callback, see Section 34.1.1
- OMPT **start_tool_result** Type, see Section 33.30

32.2.2 Determining Whether to Initialize a First-Party Tool

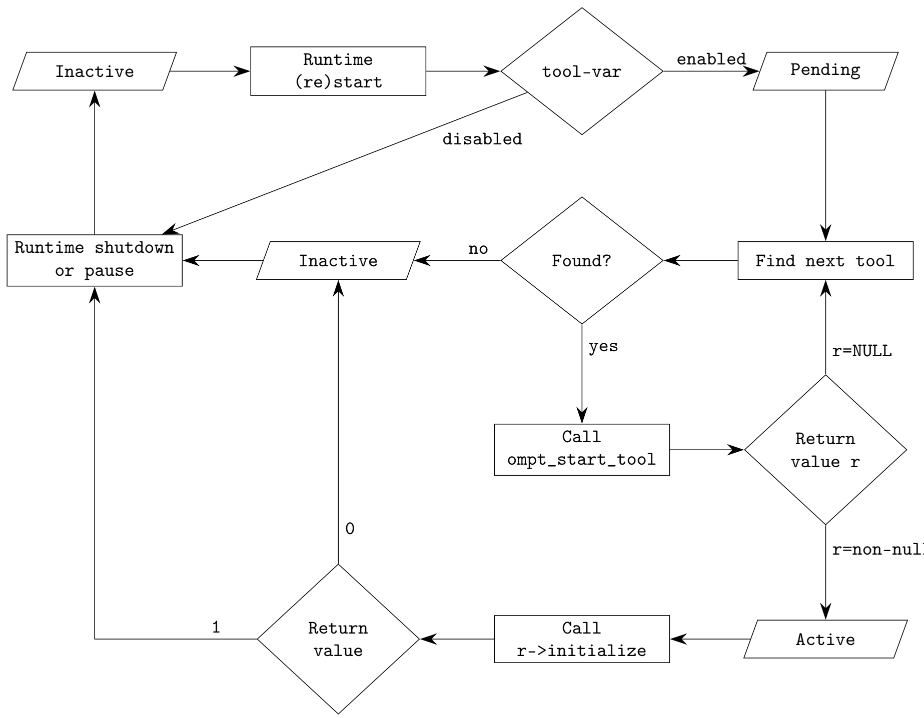

FIGURE 32.1: First-Party Tool Activation Flow Chart

An OpenMP implementation examines the *tool-var* ICV as one of its first initialization steps. If the value of *tool-var* is *disabled*, the initialization continues without a check for the presence of a tool and the functionality of the OMPT interface will be unavailable as the OpenMP program executes. In this case, the OMPT interface state remains OMPT inactive.

Otherwise, the OMPT interface state changes to OMPT pending and the OpenMP implementation activates any first-party tool that it finds. A tool can provide a definition of **ompt_start_tool** to an OpenMP implementation in three ways:

- By statically linking its definition of **ompt_start_tool** into an OpenMP program;
- By introducing a dynamically-linked library that includes its definition of **ompt_start_tool** into the address space of the program; or
- By providing, in the *tool-libraries-var* ICV, the name of a dynamically-linked library that is appropriate for the OpenMP architecture and operating system used by the OpenMP program and that includes a definition of **ompt_start_tool**.

If the value of *tool-var* is *enabled*, the OpenMP implementation must check if a tool has provided an implementation of **ompt_start_tool**. The OpenMP implementation first checks if a tool-provided implementation of **ompt_start_tool** is available in the address space, either statically-linked into the OpenMP program or in a dynamically-linked library loaded in the address space. If multiple implementations of **ompt_start_tool** are available, the implementation will use the first tool-provided implementation of **ompt_start_tool** that it finds.

If the implementation does not find a tool-provided implementation of **ompt_start_tool** in the address space, it consults the *tool-libraries-var* ICV, which contains a (possibly empty) list of dynamically-linked libraries. As described in detail in Section 4.5.2, the libraries in *tool-libraries-var* are then searched for the first usable implementation of **ompt_start_tool** that one of the libraries in the list provides.

If the implementation finds a tool-provided definition of **ompt_start_tool**, it invokes that procedure; if a NULL pointer is returned, the OMPT interface state remains OMPT pending and the implementation continues to look for implementations of **ompt_start_tool**; otherwise a non-null pointer to a **start_tool_result** OMPT type structure is returned, the OMPT interface state changes to OMPT active and the OpenMP implementation makes the OMPT interface available as the program executes. In this case, as the OpenMP implementation completes its initialization, it initializes the OMPT interface.

If no tool can be found, the OMPT interface state changes to OMPT inactive.

Cross References

- *tool-libraries-var* ICV, see Table 3.1
- *tool-var* ICV, see Table 3.1
- **ompt_start_tool** Procedure, see Section 32.2.1
- OMPT **start_tool_result** Type, see Section 33.30

32.2.3 Initializing a First-Party Tool

To initialize the OMPT interface, the OpenMP implementation invokes the OMPT-tool initializer that is specified in the **initialize** field of the **start_tool_result** structure that **ompt_start_tool** returns. This **initialize** callback is invoked prior to the occurrence of any OpenMP event.

An **initialize** callback uses the entry point specified in its *lookup* argument to look up pointers to OMPT entry points that the OpenMP implementation provides; this process is described in Section 32.2.3.1. Typically, an OMPT-tool initializer obtains a pointer to the **set_callback** entry point and then uses it to perform callback registration for events, as described in Section 32.2.4.

An OMPT-tool initializer may use the **enumerate_states** entry point to determine the thread states that an OpenMP implementation employs. Similarly, it may use the **enumerate_mutex_impls** entry point to determine the mutual exclusion implementations that the OpenMP implementation employs.

If an OMPT-tool initializer returns a non-zero value, the OMPT interface state remains OMPT active for the execution; otherwise, the OMPT interface state changes to OMPT inactive.

Cross References

- **enumerate_mutex_impls** Entry Point, see Section 36.3
- **enumerate_states** Entry Point, see Section 36.2
- Binding Entry Points, see Section 32.2.3.1
- **initialize** Callback, see Section 34.1.1
- **ompt_start_tool** Procedure, see Section 32.2.1
- **set_callback** Entry Point, see Section 36.4
- OMPT **start_tool_result** Type, see Section 33.30

32.2.3.1 Binding Entry Points

Routines that an OpenMP implementation provides to support OMPT are not defined as global symbols. Instead, they are defined as runtime entry points that a tool can only identify through the value returned in the *lookup* argument of the **initialize** callback. A tool can use this **function_lookup** entry point to obtain a pointer to each of the other entry points that an OpenMP implementation provides to support OMPT. Once a tool has obtained a **function_lookup** entry point, it may employ it at any point in the future.

For each OMPT entry point for the host device, Table 32.1 provides the string name by which it is known and its associated type signature. Implementations can provide additional implementation defined names and corresponding entry points.

During initialization, a tool should look up each entry point by name and assign the entry point to a pointer that it maintains so it can later invoke that entry point. The entry points described in Table 32.1 enable a tool to assess the thread states and mutual exclusion implementations that an implementation supports for callback registration, to inspect registered callbacks, to introspect OpenMP state associated with threads, and to use tracing to monitor computations that execute on target devices.

Cross References

- **enumerate_mutex_impls** Entry Point, see Section 36.3
- **enumerate_states** Entry Point, see Section 36.2
- **finalize_tool** Entry Point, see Section 36.20
- **function_lookup** Entry Point, see Section 36.1
- **get_callback** Entry Point, see Section 36.5
- **get_num_devices** Entry Point, see Section 36.18
- **get_num_places** Entry Point, see Section 36.8
- **get_num_procs** Entry Point, see Section 36.7

TABLE 32.1: OMPT Callback Interface Runtime Entry Point Names and Their Type Signatures

Entry Point String Name	OMPT Type
"ompt_enumerate_states"	enumerate_states
"ompt_enumerate_mutex_impls"	enumerate_mutex_impls
"ompt_set_callback"	set_callback
"ompt_get_callback"	get_callback
"ompt_get_thread_data"	get_thread_data
"ompt_get_num_places"	get_num_places
"ompt_get_place_proc_ids"	get_place_proc_ids
"ompt_get_place_num"	get_place_num
"ompt_get_partition_place_nums"	get_partition_place_nums
"ompt_get_proc_id"	get_proc_id
"ompt_get_state"	get_state
"ompt_get_parallel_info"	get_parallel_info
"ompt_get_task_info"	get_task_info
"ompt_get_task_memory"	get_task_memory
"ompt_get_num_devices"	get_num_devices
"ompt_get_num_procs"	get_num_procs
"ompt_get_target_info"	get_target_info
"ompt_get_unique_id"	get_unique_id
"ompt_finalize_tool"	finalize_tool

- **get_parallel_info** Entry Point, see Section 36.14
- **get_partition_place_nums** Entry Point, see Section 36.11
- **get_place_num** Entry Point, see Section 36.10
- **get_place_proc_ids** Entry Point, see Section 36.9
- **get_proc_id** Entry Point, see Section 36.12
- **get_state** Entry Point, see Section 36.13
- **get_target_info** Entry Point, see Section 36.17
- **get_task_info** Entry Point, see Section 36.15
- **get_task_memory** Entry Point, see Section 36.16
- **get_thread_data** Entry Point, see Section 36.6
- **get_unique_id** Entry Point, see Section 36.19
- **initialize** Callback, see Section 34.1.1
- **set_callback** Entry Point, see Section 36.4

TABLE 32.2: Callbacks for which **set_callback** Must Return **ompt_set_always**

Callback Name
thread_begin
thread_end
parallel_begin
parallel_end
task_create
task_schedule
implicit_task
target_data_op_emi
target_emi
target_submit_emi
control_tool
device_initialize
device_finalize
device_load
device_unload
error

32.2.4 Monitoring Activity on the Host with OMPT

To monitor the execution of an OpenMP program on the host device, an OMPT-tool initializer must register to receive notification of events that occur as an OpenMP program executes. A tool can use the **set_callback** entry point to perform callback registrations for events. The return codes for **set_callback** use the **set_result** OMPT type. If the **set_callback** entry point is called outside an **initialize** OMPT callback, callback registration may fail for supported callbacks with a return value of **ompt_set_error**. All registered callbacks and all callbacks returned by **get_callback** use the **callback** OMPT type as a dummy type signature.

For callbacks listed in Table 32.2, **ompt_set_always** is the only registration return code that is allowed. An OpenMP implementation must guarantee that the callback will be invoked every time that a runtime event that is associated with it occurs. Support for such callbacks is required in a minimal implementation of the OMPT interface.

For any other callbacks not listed in Table 32.2, the **set_callback** entry point may return any non-error code. Whether an OpenMP implementation invokes a registered callback never, sometimes, or always is implementation defined. If registration for a callback allows a return code of **ompt_set_never**, support for invoking such a callback may not be present in a minimal implementation of the OMPT interface. The return code from callback registration indicates the implementation defined level of support for the callback.

Two techniques reduce the size of the OMPT interface. First, in cases where events are naturally paired, for example the beginning and end of a region, and the arguments needed by the callback at each region endpoint are identical, a tool registers a single callback for the pair of events, with

ompt_scope_begin or **ompt_scope_end** provided as an argument to identify for which region endpoint the callback is invoked. Second, when a class of events is amenable to uniform treatment, OMPT provides a single callback for that class of events; for example, a **sync_region_wait** callback is used for multiple kinds of synchronization regions, such as barrier, **taskwait**, and **taskgroup** regions. Some events, for example, those that correspond to **sync_region_wait**, use both techniques.

Cross References

- **get_callback** Entry Point, see Section 36.5
- **initialize** Callback, see Section 34.1.1
- OMPT **scope_endpoint** Type, see Section 33.27
- **set_callback** Entry Point, see Section 36.4
- OMPT **set_result** Type, see Section 33.28

32.2.5 Tracing Activity on Target Devices

A target device may not initialize a full OpenMP runtime system. Without one, using a tool interface based on callbacks to monitor activity on a device may incur unacceptable overhead. Thus, OMPT defines a monitoring interface for tracing activity on target devices. This section details the use of that interface.

First, to prepare to trace device activity, a tool must register a **device_initialize** callback. A tool may also register a **device_load** callback to be notified when code is loaded onto a target device or a **device_unload** callback to be notified when code is unloaded from a target device. A tool may also optionally register a **device_finalize** callback.

When an OpenMP implementation initializes a target device, it dispatches the **device_initialize** callback (the device initializer) of the tool on the host device. If the OpenMP implementation or target device does not support tracing, the OpenMP implementation passes NULL to the device initializer of the tool for its *lookup* argument; otherwise, the OpenMP implementation passes a pointer to a device-specific **function_lookup** entry point to the **device_initialize** callback of the tool.

If the *lookup* argument of the **device_initialize** of the tool is a non-null pointer, the tool may use it to determine the entry points in the tracing interface that are available for the device and may bind the returned function pointers to tool variables. Table 32.3 lists the names of runtime entry points that may be available for a device; an implementation may provide additional implementation defined names and corresponding entry points. The driver for the device provides the entry points that enable a tool to control the trace collection interface of the device. The native trace format that the interface uses may be device-specific and the available kinds of trace records are implementation defined.

Some devices may allow a tool to collect trace records in a standard trace format known as OMPT trace records. Each OMPT trace record serves as a substitute for an OMPT callback that is not appropriate to be dispatched on the device. The fields in each trace record type are defined in the

TABLE 32.3: OMPT Tracing Interface Runtime Entry Point Names and Their Type Signatures

Entry Point String Name	OMPT Type
"ompt_get_device_num_procs"	get_device_num_procs
"ompt_get_device_time"	get_device_time
"ompt_translate_time"	translate_time
"ompt_set_trace_ompt"	set_trace_ompt
"ompt_set_trace_native"	set_trace_native
"ompt_get_buffer_limits"	get_buffer_limits
"ompt_start_trace"	start_trace
"ompt_pause_trace"	pause_trace
"ompt_flush_trace"	flush_trace
"ompt_stop_trace"	stop_trace
"ompt_advance_buffer_cursor"	advance_buffer_cursor
"ompt_get_record_type"	get_record_type
"ompt_get_record_ompt"	get_record_ompt
"ompt_get_record_native"	get_record_native
"ompt_get_record_abstract"	get_record_abstract

description of the callback that the record represents. If this type of record is provided then the **function_lookup** entry point returns values for the entry points **set_trace_ompt** and **get_record_ompt**, which support collecting and decoding OMPT traces. If the native trace format for a device is the OMPT format then tracing can be controlled using the entry points for native or OMPT tracing.

The tool uses the **set_trace_native** and/or the **set_trace_ompt** runtime entry point to specify what types of events or activities to monitor on the device. The return codes for **set_trace_ompt** and **set_trace_native** use the **set_result** OMPT type. If the **set_trace_native** or the **set_trace_ompt** entry point is called outside a device initializer, registration of supported callbacks may fail with a return code of **ompt_set_error**. After specifying the events or activities to monitor, the tool initiates tracing of device activity by invoking the **start_trace** entry point. Arguments to **start_trace** include two tool callbacks through which the OpenMP implementation can manage traces associated with the device. The **buffer_request** callback allocates a buffer in which trace records that correspond to device activity can be deposited. The **buffer_complete** callback processes a buffer of trace records from the device. If the OpenMP implementation requires a trace buffer for device activity, it invokes the tool-supplied callback on the host device to request a new buffer. The OpenMP implementation then monitors the execution of OpenMP constructs on the device and records a trace of events or activities into a trace buffer. If possible, device trace records are marked with a **host_op_id** — an identifier that associates device activities with the target device operation that the host device initiated to cause these activities.

To correlate activities on the host device with activities on a target device, a tool can register a **target_submit_emi** callback. Before and after the host device initiates creation of an initial task on a device associated with a structured block for a **target** construct, the OpenMP implementation dispatches the **target_submit_emi** callback on the host device in the thread that is executing the encountering task of the **target** construct. This callback provides the tool with a pair of identifiers: one that identifies the **target** region and a second that uniquely identifies the initial task associated with that region. These identifiers help the tool correlate activities on the target device with their **target** region.

When appropriate, for example, when a trace buffer fills or needs to be flushed, the OpenMP implementation invokes the tool-supplied **buffer_complete** callback to process a non-empty sequence of trace records in a trace buffer that is associated with the device. The **buffer_complete** callback may return immediately, ignoring records in the trace buffer, or it may iterate through them using the **advance_buffer_cursor** entry point to inspect each trace record.

A tool may use the **get_record_type** entry point to inspect the type of the trace record at the current cursor position. Three entry points (**get_record_ompt**, **get_record_native**, and **get_record_abstract**) allow tools to inspect the contents of some or all trace records in a trace buffer. The **get_record_native** entry point uses the native trace format of the device. The **get_record_abstract** entry point decodes the contents of a native trace record and summarizes them as a **record_abstract** OMPT type record. The **get_record_ompt** entry point can only be used to retrieve trace records in OMPT format.

Once device tracing has been started, a tool may pause or resume device tracing at any time by invoking **pause_trace** with an appropriate flag value as an argument. Further, a tool may invoke the **flush_trace** entry point for a device at any time between device initialization and finalization to cause the pending trace records for that device to be flushed.

At any time, a tool may use the **start_trace** entry point to start or the **stop_trace** entry point to stop device tracing. When device tracing is stopped, the OpenMP implementation eventually gathers all trace records already collected from device tracing and presents them to the tool using the buffer-completion callback.

An OpenMP implementation can be shut down while device tracing is in progress. When an OpenMP implementation is shut down, it finalizes each device. Device finalization occurs in three steps. First, the OpenMP implementation halts any tracing in progress for the device. Second, the OpenMP implementation flushes all trace records collected for the device and uses the **buffer_complete** callback associated with that device to present them to the tool. Finally, the OpenMP implementation dispatches any **device_finalize** callback registered for the device.

Cross References

- **advance_buffer_cursor** Entry Point, see Section 37.11
- **buffer_complete** Callback, see Section 35.6
- **buffer_request** Callback, see Section 35.5
- **device_finalize** Callback, see Section 35.2
- **device_initialize** Callback, see Section 35.1
- **device_load** Callback, see Section 35.3

- **device_unload** Callback, see Section 35.4
- **flush_trace** Entry Point, see Section 37.9
- **function_lookup** Entry Point, see Section 36.1
- **get_buffer_limits** Entry Point, see Section 37.6
- **get_device_num_procs** Entry Point, see Section 37.1
- **get_device_time** Entry Point, see Section 37.2
- **get_record_abstract** Entry Point, see Section 37.15
- **get_record_native** Entry Point, see Section 37.14
- **get_record_ompt** Entry Point, see Section 37.13
- **get_record_type** Entry Point, see Section 37.12
- **pause_trace** Entry Point, see Section 37.8
- OMPT **record_abstract** Type, see Section 33.24
- OMPT **set_result** Type, see Section 33.28
- **set_trace_native** Entry Point, see Section 37.5
- **set_trace_ompt** Entry Point, see Section 37.4
- **start_trace** Entry Point, see Section 37.7
- **stop_trace** Entry Point, see Section 37.10
- **translate_time** Entry Point, see Section 37.3

32.3 Finalizing a First-Party Tool

If the OMPT interface state is OMPT active, the OMPT-tool finalizer, which is a **finalize** callback and is specified by the **finalize** field in the **start_tool_result** OMPT type structure returned from the **ompt_start_tool** procedure, is called when the OpenMP implementation shuts down.

Cross References

- **finalize** Callback, see Section 34.1.2
- **ompt_start_tool** Procedure, see Section 32.2.1
- OMPT **start_tool_result** Type, see Section 33.30

33 OMPT Data Types

This chapter specifies OMPT types that the **omp-tools.h** C/C++ header file defines.

C / C++

33.1 OMPT Predefined Identifiers

Predefined Identifiers

Name	Value	Properties
ompt_addr_none	**~0**	*default*
ompt_mutex_impl_none	**0**	*default*

In addition to the predefined identifiers of OMPT type that are defined with their corresponding OMPT type, the OpenMP API includes the predefined identifiers shown above. The **ompt_addr_none void *** predefined identifier indicates that no address on the relevant device is available. The **ompt_mutex_impl_none** predefined identifier indicates an invalid mutex implementation.

C / C++

33.2 OMPT any_record_ompt Type

Name: **any_record_ompt** **Properties:** C/C++-only, OMPT	**Base Type:** union

Fields

Name	Type	Properties
thread_begin	thread_begin	C/C++-only
parallel_begin	parallel_begin	C/C++-only
parallel_end	parallel_end	C/C++-only
work	work	C/C++-only
dispatch	dispatch	C/C++-only
task_create	task_create	C/C++-only
dependences	dependences	C/C++-only
task_dependence	task_dependence	C/C++-only
task_schedule	task_schedule	C/C++-only
implicit_task	implicit_task	C/C++-only
masked	masked	C/C++-only
sync_region	sync_region	C/C++-only
mutex_acquire	mutex_acquire	C/C++-only
mutex	mutex	C/C++-only
nest_lock	nest_lock	C/C++-only
flush	flush	C/C++-only
cancel	cancel	C/C++-only
target_emi	target_emi	C/C++-only
target_data_op_emi	target_data_op_emi	C/C++-only
target_map_emi	target_map_emi	C/C++-only
target_submit_emi	target_submit_emi	C/C++-only
control_tool	control_tool	C/C++-only
error	error	C/C++-only

Type Definition

C / C++

```
typedef union ompt_any_record_ompt_t {
  ompt_record_thread_begin_t thread_begin;
  ompt_record_parallel_begin_t parallel_begin;
  ompt_record_parallel_end_t parallel_end;
  ompt_record_work_t work;
  ompt_record_dispatch_t dispatch;
  ompt_record_task_create_t task_create;
  ompt_record_dependences_t dependences;
  ompt_record_task_dependence_t task_dependence;
  ompt_record_task_schedule_t task_schedule;
  ompt_record_implicit_task_t implicit_task;
  ompt_record_masked_t masked;
  ompt_record_sync_region_t sync_region;
  ompt_record_mutex_acquire_t mutex_acquire;
  ompt_record_mutex_t mutex;
```

```
  ompt_record_nest_lock_t nest_lock;
  ompt_record_flush_t flush;
  ompt_record_cancel_t cancel;
  ompt_record_target_emi_t target_emi;
  ompt_record_target_data_op_emi_t target_data_op_emi;
  ompt_record_target_map_emi_t target_map_emi;
  ompt_record_target_submit_emi_t target_submit_emi;
  ompt_record_control_tool_t control_tool;
  ompt_record_error_t error;
} ompt_any_record_ompt_t;
```

C / C++

Additional information

The union also includes **target**, **taget_data_op**, **target_kernel**, and **target_map** fields with corresponding trace record OMPT types. These fields have been deprecated.

Semantics

The **any_record_ompt** OMPT type is a union of all standard trace format event-specific trace record OMPT types that is the type of the **record** field of the **record_ompt** OMPT type.

Cross References

- OMPT **record_ompt** Type, see Section 33.26

33.3 OMPT buffer Type

Name: buffer **Properties:** C/C++-only, OMPT, opaque	**Base Type: void**

Type Definition

C / C++

```
typedef void ompt_buffer_t;
```

C / C++

Semantics

The **buffer** OMPT type represents a handle for a device buffer.

33.4 OMPT buffer_cursor Type

Name: buffer_cursor **Properties:** C/C++-only, OMPT, opaque	**Base Type: c_uint64_t**

Type Definition

C / C++

```
typedef uint64_t ompt_buffer_cursor_t;
```

C / C++

Summary

The **buffer_cursor** OMPT type represents a handle for a position in a device buffer.

33.5 OMPT callback Type

Name: callback **Category:** subroutine pointer	**Properties:** C/C++-only, OMPT

Type Signature

C / C++

```
typedef void (*ompt_callback_t) (void);
```

C / C++

Semantics

Pointers to OMPT callbacks with different type signatures are passed to the **set_callback** entry point and returned by the **get_callback** entry point. For convenience, these entry points require all type signatures to be cast to the **callback** OMPT type.

33.6 OMPT callbacks Type

Name: callbacks **Properties:** C/C++-only, OMPT	**Base Type:** enumeration

Values

Name	Value	Properties
ompt_callback_thread_begin	1	C-only, OMPT
ompt_callback_thread_end	2	C-only, OMPT
ompt_callback_parallel_begin	3	C-only, OMPT
ompt_callback_parallel_end	4	C-only, OMPT
ompt_callback_task_create	5	C-only, OMPT
ompt_callback_task_schedule	6	C-only, OMPT
ompt_callback_implicit_task	7	C-only, OMPT
ompt_callback_control_tool	11	C-only, OMPT
ompt_callback_device_initialize	12	C-only, OMPT
ompt_callback_device_finalize	13	C-only, OMPT
ompt_callback_device_load	14	C-only, OMPT
ompt_callback_device_unload	15	C-only, OMPT
ompt_callback_sync_region_wait	16	C-only, OMPT
ompt_callback_mutex_released	17	C-only, OMPT
ompt_callback_dependences	18	C-only, OMPT
ompt_callback_task_dependence	19	C-only, OMPT
ompt_callback_work	20	C-only, OMPT
ompt_callback_masked	21	C-only, OMPT
ompt_callback_sync_region	23	C-only, OMPT
ompt_callback_lock_init	24	C-only, OMPT
ompt_callback_lock_destroy	25	C-only, OMPT
ompt_callback_mutex_acquire	26	C-only, OMPT
ompt_callback_mutex_acquired	27	C-only, OMPT
ompt_callback_nest_lock	28	C-only, OMPT
ompt_callback_flush	29	C-only, OMPT
ompt_callback_cancel	30	C-only, OMPT
ompt_callback_reduction	31	C-only, OMPT
ompt_callback_dispatch	32	C-only, OMPT
ompt_callback_target_emi	33	C-only, OMPT
ompt_callback_target_data_op_emi	34	C-only, OMPT
ompt_callback_target_submit_emi	35	C-only, OMPT
ompt_callback_target_map_emi	36	C-only, OMPT
ompt_callback_error	37	C-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_callbacks_t {
  ompt_callback_thread_begin        = 1,
  ompt_callback_thread_end          = 2,
  ompt_callback_parallel_begin      = 3,
  ompt_callback_parallel_end        = 4,
```

```
    ompt_callback_task_create          = 5,
    ompt_callback_task_schedule        = 6,
    ompt_callback_implicit_task        = 7,
    ompt_callback_control_tool         = 11,
    ompt_callback_device_initialize    = 12,
    ompt_callback_device_finalize      = 13,
    ompt_callback_device_load          = 14,
    ompt_callback_device_unload        = 15,
    ompt_callback_sync_region_wait     = 16,
    ompt_callback_mutex_released       = 17,
    ompt_callback_dependences          = 18,
    ompt_callback_task_dependence      = 19,
    ompt_callback_work                 = 20,
    ompt_callback_masked               = 21,
    ompt_callback_sync_region          = 23,
    ompt_callback_lock_init            = 24,
    ompt_callback_lock_destroy         = 25,
    ompt_callback_mutex_acquire        = 26,
    ompt_callback_mutex_acquired       = 27,
    ompt_callback_nest_lock            = 28,
    ompt_callback_flush                = 29,
    ompt_callback_cancel               = 30,
    ompt_callback_reduction            = 31,
    ompt_callback_dispatch             = 32,
    ompt_callback_target_emi           = 33,
    ompt_callback_target_data_op_emi   = 34,
    ompt_callback_target_submit_emi    = 35,
    ompt_callback_target_map_emi       = 36,
    ompt_callback_error                = 37
} ompt_callbacks_t;
```

C / C++

Additional information

The following instances and associated values of the **callbacks** OMPT type are also defined: **ompt_callback_target**, with value 8; **ompt_callback_target_data_op**, with value 9; **ompt_callback_target_submit**, with value 10; and **ompt_callback_target_map**, with value 22. These instances have been deprecated.

Semantics

The **callbacks** OMPT type provides codes that identify OMPT callbacks when registering or querying them.

33.7 OMPT `cancel_flag` Type

Name: `cancel_flag` **Properties:** C/C++-only, OMPT	**Base Type:** enumeration

Values

Name	Value	Properties
`ompt_cancel_parallel`	`0x01`	C/C++-only, OMPT
`ompt_cancel_sections`	`0x02`	C/C++-only, OMPT
`ompt_cancel_loop`	`0x04`	C/C++-only, OMPT
`ompt_cancel_taskgroup`	`0x08`	C/C++-only, OMPT
`ompt_cancel_activated`	`0x10`	C/C++-only, OMPT
`ompt_cancel_detected`	`0x20`	C/C++-only, OMPT
`ompt_cancel_discarded_task`	`0x40`	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_cancel_flag_t {
  ompt_cancel_parallel       = 0x01,
  ompt_cancel_sections       = 0x02,
  ompt_cancel_loop           = 0x04,
  ompt_cancel_taskgroup      = 0x08,
  ompt_cancel_activated      = 0x10,
  ompt_cancel_detected       = 0x20,
  ompt_cancel_discarded_task = 0x40
} ompt_cancel_flag_t;
```

C / C++

Semantics

The **`cancel_flag`** OMPT type defines cancel flag values.

33.8 OMPT `data` Type

Name: `data` **Properties:** C/C++-only, OMPT	**Base Type:** union

Fields

Name	Type	Properties
value	c_uint64_t	*default*
ptr	void	C/C++-only, pointer

Predefined Identifiers

Name	Value	Properties
`ompt_data_none`	0	C/C++-only, OMPT

Type Definition

C / C++

```
typedef union ompt_data_t {
  uint64_t value;
  void *ptr;
} ompt_data_t;
```

C / C++

Semantics

The **data** OMPT type represents data that is reserved for tool use. When an OpenMP implementation creates a thread or an instance of a parallel region, **teams** region, task region, or device region, it initializes the associated **data** object with the value **ompt_data_none**.

33.9 OMPT dependence Type

Name: dependence **Properties:** C/C++-only, OMPT	**Base Type:** structure

Fields

Name	Type	Properties
variable	data	C/C++-only
dependence_type	dependence_type	C/C++-only

Type Definition

C / C++

```
typedef struct ompt_dependence_t {
  ompt_data_t variable;
  ompt_dependence_type_t dependence_type;
} ompt_dependence_t;
```

C / C++

Semantics

The **dependence** OMPT type represents a dependence in a structure that holds information about a **depend** or **doacross** clause. For task dependences, the **ptr** field of its **variable** field points to the storage location of the dependence. For doacross dependences, the **value** field of the **variable** field contains the value of a vector element that describes the dependence. The **dependence_type** field indicates the type of the dependence. For task dependences with the reserved locator **omp_all_memory**, the value of the **variable** field is undefined and the **dependence_type** field contains a value that has the **_all_memory** suffix.

Cross References

- OMPT **data** Type, see Section 33.8
- OMPT **dependence_type** Type, see Section 33.10

33.10 OMPT `dependence_type` Type

Name: `dependence_type` **Properties:** C/C++-only, OMPT	**Base Type:** enumeration

Values

Name	Value	Properties
`ompt_dependence_type_in`	1	C/C++-only, OMPT
`ompt_dependence_type_out`	2	C/C++-only, OMPT
`ompt_dependence_type_inout`	3	C/C++-only, OMPT
`ompt_dependence_type_mutexinoutset`	4	C/C++-only, OMPT
`ompt_dependence_type_source`	5	C/C++-only, OMPT
`ompt_dependence_type_sink`	6	C/C++-only, OMPT
`ompt_dependence_type_inoutset`	7	C/C++-only, OMPT
`ompt_dependence_type_out_all_memory`	34	C/C++-only, OMPT
`ompt_dependence_type_inout_all_memory`	35	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_dependence_type_t {
  ompt_dependence_type_in                = 1,
  ompt_dependence_type_out               = 2,
  ompt_dependence_type_inout             = 3,
  ompt_dependence_type_mutexinoutset     = 4,
  ompt_dependence_type_source            = 5,
  ompt_dependence_type_sink              = 6,
  ompt_dependence_type_inoutset          = 7,
  ompt_dependence_type_out_all_memory    = 34,
  ompt_dependence_type_inout_all_memory = 35
} ompt_dependence_type_t;
```

C / C++

Semantics

The **`dependence_type`** OMPT type defines task dependence type values. The **`ompt_dependence_type_in`**, **`ompt_dependence_type_out`**, **`ompt_dependence_type_inout`**, **`ompt_dependence_type_mutexinoutset`**, **`ompt_dependence_type_inoutset`**, **`ompt_dependence_type_out_all_memory`**, and **`ompt_dependence_type_inout_all_memory`** values represent the task dependence type present in a **`depend`** clause while the **`ompt_dependence_type_source`** and **`ompt_dependence_type_sink`** values represent the *dependence-type* present in a **`doacross`** clause. The **`ompt_dependence_type_out_all_memory`** and **`ompt_dependence_type_inout_all_memory`** represent task dependences for which the **`omp_all_memory`** reserved locator is specified.

33.11 OMPT device Type

Name: device Properties: C/C++-only, OMPT, opaque	Base Type: void

Type Definition

C / C++

```
typedef void ompt_device_t;
```

C / C++

Semantics

The **device** OMPT type represents a device.

33.12 OMPT device_time Type

Name: device_time Properties: C/C++-only, OMPT, opaque	Base Type: c_uint64_t

Predefined Identifiers

Name	Value	Properties
ompt_time_none	0	C/C++-only, OMPT

Type Definition

C / C++

```
typedef uint64_t ompt_device_time_t;
```

C / C++

Semantics

The **device_time** OMPT type represents raw device time values; **ompt_time_none** represents an unknown or unspecified time.

33.13 OMPT dispatch Type

Name: dispatch Properties: C/C++-only, OMPT, overlapping-type-name	Base Type: enumeration

Values

Name	Value	Properties
`ompt_dispatch_iteration`	1	C/C++-only, OMPT
`ompt_dispatch_section`	2	C/C++-only, OMPT
`ompt_dispatch_ws_loop_chunk`	3	C/C++-only, OMPT
`ompt_dispatch_taskloop_chunk`	4	C/C++-only, OMPT
`ompt_dispatch_distribute_chunk`	5	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_dispatch_t {
  ompt_dispatch_iteration        = 1,
  ompt_dispatch_section          = 2,
  ompt_dispatch_ws_loop_chunk    = 3,
  ompt_dispatch_taskloop_chunk   = 4,
  ompt_dispatch_distribute_chunk = 5
} ompt_dispatch_t;
```

C / C++

Semantics

The **`dispatch`** OMPT type defines the valid dispatch values.

33.14 OMPT `dispatch_chunk` Type

Name: `dispatch_chunk` **Properties:** C/C++-only, OMPT	**Base Type:** structure

Fields

Name	Type	Properties
start	c_uint64_t	*default*
iterations	c_uint64_t	*default*

Type Definition

C / C++

```
typedef struct ompt_dispatch_chunk_t {
  uint64_t start;
  uint64_t iterations;
} ompt_dispatch_chunk_t;
```

C / C++

Semantics

The **dispatch_chunk** OMPT type represents chunk information for a dispatched chunk. The **start** field specifies the first logical iteration of the chunk and the **iterations** field specifies the number of logical iterations in the chunk. Whether the chunk of a **taskloop** region is contiguous is implementation defined.

33.15 OMPT frame Type

Name: **frame** **Properties:** C/C++-only, OMPT	**Base Type:** structure

Fields

Name	Type	Properties
exit_frame	data	C/C++-only, OMPT
enter_frame	data	C/C++-only, OMPT
exit_frame_flags	integer	*default*
enter_frame_flags	integer	*default*

Type Definition

C / C++

```
typedef struct ompt_frame_t {
  ompt_data_t exit_frame;
  ompt_data_t enter_frame;
  int exit_frame_flags;
  int enter_frame_flags;
} ompt_frame_t;
```

C / C++

Semantics

The **frame** OMPT type describes procedure frame information for a task. Each **frame** object is associated with the task to which the procedure frames belong. Every task that is not a merged task with one or more frames on the stack of a native thread, whether an initial task, an implicit task, an explicit task, or a target task, has an associated **frame** object.

The **exit_frame** field contains information to identify the first procedure frame executing the task region. The **exit_frame** for the **frame** object associated with the initial task that is not nested inside any OpenMP construct is **ompt_data_none**. The **enter_frame** field contains information to identify the latest still active procedure frame executing the task region before entering the OpenMP runtime implementation or before executing a different task. If a task with frames on the stack is not executing implementation code in the OpenMP runtime, the value of **enter_frame** for its associated **frame** object is **ompt_data_none**.

For the frame indicated by **exit_frame** (**enter_frame**), the **exit_frame_flags** (**enter_frame_flags**) field indicates that the provided frame information points to a runtime or an OpenMP program frame address. The same fields also specify the kind of information that is provided to identify the frame, These fields are a disjunction of values in the **frame_flag** OMPT type.

The lifetime of a **frame** object begins when a task is created and ends when the task is destroyed. Tools should not assume that a frame structure remains at a constant location in memory throughout the lifetime of the task. A pointer to a **frame** object is passed to some callbacks; a pointer to the **frame** object of a task can also be retrieved by a tool at any time, including in a signal handler, by invoking the **get_task_info** entry point. A pointer to a **frame** object that a tool retrieved is valid as long as the tool does not pass back control to the OpenMP implementation.

Note – A monitoring tool that uses asynchronous sampling can observe values of **exit_frame** and **enter_frame** at inconvenient times. Tools must be prepared to handle **frame** objects observed just prior to when their field values will be set or cleared.

Cross References

- OMPT **data** Type, see Section 33.8
- OMPT **frame_flag** Type, see Section 33.16
- **get_task_info** Entry Point, see Section 36.15

33.16 OMPT frame_flag Type

Name: **frame_flag**	Base Type: enumeration
Properties: C/C++-only, OMPT	

Values

Name	Value	Properties
ompt_frame_runtime	**0x00**	C/C++-only, OMPT
ompt_frame_application	**0x01**	C/C++-only, OMPT
ompt_frame_cfa	**0x10**	C/C++-only, OMPT
ompt_frame_framepointer	**0x20**	C/C++-only, OMPT
ompt_frame_stackaddress	**0x30**	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_frame_flag_t {
  ompt_frame_runtime       = 0x00,
  ompt_frame_application   = 0x01,
  ompt_frame_cfa           = 0x10,
  ompt_frame_framepointer = 0x20,
```

```
    ompt_frame_stackaddress = 0x30
} ompt_frame_flag_t;
```

C / C++

Semantics

The **frame_flag** OMPT type defines frame information flags. The **ompt_frame_runtime** value indicates that a frame address is a procedure frame in the OpenMP runtime implementation. The **ompt_frame_application** value indicates that a frame address is a procedure frame in the OpenMP program. Higher order bits indicate the specific information for a particular frame pointer. The **ompt_frame_cfa** value indicates that a frame address specifies a canonical frame address. The **ompt_frame_framepointer** value indicates that a frame address provides the value of the frame pointer register. The **ompt_frame_stackaddress** value indicates that a frame address specifies a pointer address that is contained in the current stack frame.

33.17 OMPT hwid Type

Name: hwid Properties: C/C++-only, OMPT	Base Type: c_uint64_t

Predefined Identifiers

Name	Value	Properties
ompt_hwid_none	0	C/C++-only, OMPT

Type Definition

C / C++

```
typedef uint64_t ompt_hwid_t;
```

C / C++

Semantics

The **hwid** OMPT type is a handle for a hardware identifier for a target device; **ompt_hwid_none** represents an unknown or unspecified hardware identifier. If no specific value for the **hwid** field is associated with an instance of the **record_abstract** OMPT type then the value of **hwid** is **ompt_hwid_none**.

Cross References

- OMPT **record_abstract** Type, see Section 33.24

33.18 OMPT id Type

Name: id Properties: C/C++-only, OMPT	Base Type: c_uint64_t

Predefined Identifiers

Name	Value	Properties
`ompt_id_none`	0	C/C++-only, OMPT

Type Definition

C / C++

```
typedef uint64_t ompt_id_t;
```

C / C++

Semantics

The **id** OMPT type is used to provide various identifiers to tools; **ompt_id_none** is used when the specific ID is unknown or unavailable. When tracing asynchronous activity on devices, identifiers enable tools to correlate device regions and operations that the host device initiates with associated activities on a target device. In addition, OMPT provides identifiers to refer to parallel regions and tasks that execute on a device.

Restrictions

Restrictions to the **id** OMPT type are as follows:

- Identifiers created on each device must be unique from the time an OpenMP implementation is initialized until it is shut down. Identifiers for each device region and target data operation instance that the host device initiates must be unique over time on the host device. Identifiers for instances of parallel regions and task regions that execute on a device must be unique over time within that device.

33.19 OMPT `interface_fn` Type

Name: `interface_fn` **Category:** subroutine pointer	**Properties:** C/C++-only, OMPT

Type Signature

C / C++

```
typedef void (*ompt_interface_fn_t) (void);
```

C / C++

Semantics

The **interface_fn** OMPT type serves as a generic function pointer that the **function_lookup** entry point returns to provide access to a tool to entry points by name.

33.20 OMPT `mutex` Type

Name: `mutex` **Properties:** C/C++-only, OMPT, overlapping-type-name	**Base Type:** enumeration

Values

Name	Value	Properties
`ompt_mutex_lock`	1	C/C++-only, OMPT
`ompt_mutex_test_lock`	2	C/C++-only, OMPT
`ompt_mutex_nest_lock`	3	C/C++-only, OMPT
`ompt_mutex_test_nest_lock`	4	C/C++-only, OMPT
`ompt_mutex_critical`	5	C/C++-only, OMPT
`ompt_mutex_atomic`	6	C/C++-only, OMPT
`ompt_mutex_ordered`	7	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_mutex_t {
  ompt_mutex_lock           = 1,
  ompt_mutex_test_lock      = 2,
  ompt_mutex_nest_lock      = 3,
  ompt_mutex_test_nest_lock = 4,
  ompt_mutex_critical       = 5,
  ompt_mutex_atomic         = 6,
  ompt_mutex_ordered        = 7
} ompt_mutex_t;
```

C / C++

Semantics

The **`mutex`** OMPT type defines the valid mutex values.

33.21 OMPT `native_mon_flag` Type

Name: `native_mon_flag` **Properties:** C/C++-only, OMPT	**Base Type:** enumeration

Values

Name	Value	Properties
`ompt_native_data_motion_explicit`	0x01	C/C++-only, OMPT
`ompt_native_data_motion_implicit`	0x02	C/C++-only, OMPT
`ompt_native_kernel_invocation`	0x04	C/C++-only, OMPT
`ompt_native_kernel_execution`	0x08	C/C++-only, OMPT
`ompt_native_driver`	0x10	C/C++-only, OMPT
`ompt_native_runtime`	0x20	C/C++-only, OMPT
`ompt_native_overhead`	0x40	C/C++-only, OMPT
`ompt_native_idleness`	0x80	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_native_mon_flag_t {
  ompt_native_data_motion_explicit = 0x01,
  ompt_native_data_motion_implicit = 0x02,
  ompt_native_kernel_invocation    = 0x04,
  ompt_native_kernel_execution     = 0x08,
  ompt_native_driver               = 0x10,
  ompt_native_runtime              = 0x20,
  ompt_native_overhead             = 0x40,
  ompt_native_idleness             = 0x80
} ompt_native_mon_flag_t;
```

C / C++

Semantics

The **`native_mon_flag`** OMPT type defines the valid native monitoring flag values.

33.22 OMPT `parallel_flag` Type

Name: `parallel_flag` **Properties:** C/C++-only, OMPT	**Base Type:** enumeration

Values

Name	Value	Properties
`ompt_parallel_invoker_program`	`0x00000001`	C/C++-only, OMPT
`ompt_parallel_invoker_runtime`	`0x00000002`	C/C++-only, OMPT
`ompt_parallel_league`	`0x40000000`	C/C++-only, OMPT
`ompt_parallel_team`	`0x80000000`	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_parallel_flag_t {
  ompt_parallel_invoker_program = 0x00000001,
  ompt_parallel_invoker_runtime = 0x00000002,
  ompt_parallel_league          = 0x40000000,
  ompt_parallel_team            = 0x80000000
} ompt_parallel_flag_t;
```

C / C++

Semantics

The **parallel_flag** OMPT type defines valid invoker values, which indicate how the code that implements the associated structured block of the region is invoked or encountered. The **ompt_parallel_invoker_program** value indicates that the encountering thread for a **parallel** or **teams** region executes code to implement its associated structured block as if directly invoked or encountered in application code. The **ompt_parallel_invoker_runtime** value indicates that the encountering thread for a **parallel** or **teams** region invokes the code that implements its associated structured block from the runtime. The **ompt_parallel_league** value indicates that the callback is invoked due to the creation of a league of teams by a **teams** construct. The **ompt_parallel_team** value indicates that the callback is invoked due to the creation of a team of threads by a **parallel** construct.

33.23 OMPT record Type

Name: **record** **Properties:** C/C++-only, OMPT	**Base Type:** enumeration

Values

Name	Value	Properties
ompt_record_ompt	**1**	C/C++-only, OMPT
ompt_record_native	**2**	C/C++-only, OMPT
ompt_record_invalid	**3**	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_record_t {
  ompt_record_ompt    = 1,
  ompt_record_native  = 2,
  ompt_record_invalid = 3
} ompt_record_t;
```

C / C++

Semantics

The **record** OMPT type indicates the integer codes that identify OMPT trace record formats.

33.24 OMPT record_abstract Type

Name: **record_abstract** **Properties:** C/C++-only, OMPT	**Base Type:** structure

Fields

Name	Type	Properties
rclass	record_native	C/C++-only, OMPT
type	char	common-field, intent(in), pointer
start_time	device_time	C/C++-only, OMPT
end_time	device_time	C/C++-only, OMPT
hwid	hwid	C/C++-only, OMPT

Type Definition

C / C++

```
typedef struct ompt_record_abstract_t {
  ompt_record_native_t rclass;
  const char *type;
  ompt_device_time_t start_time;
  ompt_device_time_t end_time;
  ompt_hwid_t hwid;
} ompt_record_abstract_t;
```

C / C++

Semantics

The **record_abstract** OMPT type is an abstract trace record format that summarizes native trace records. It contains information that a tool can use to process a native trace record that it may not fully understand. The **rclass** field indicates that the trace record is informational or that it represents an event; this information can help a tool determine how to present the trace record. The **type** field points to a statically-allocated, immutable character string that provides a meaningful name that a tool can use to describe the event. The **start_time** and **end_time** fields are used to place an event in time. The times are relative to the device clock. If an event does not have an associated **start_time** (**end_time**), the value of the **start_time** (**end_time**) field is **ompt_time_none**. The hardware identifier field, **hwid**, indicates the location on the device where the event occurred. A **hwid** may represent a hardware abstraction such as a core or a hardware thread identifier. The meaning of a **hwid** value for a device is implementation defined. If no hardware abstraction is associated with the trace record then the value of **hwid** is **ompt_hwid_none**.

Cross References

- OMPT **device_time** Type, see Section 33.12
- OMPT **hwid** Type, see Section 33.17
- OMPT **record_native** Type, see Section 33.25

33.25 OMPT `record_native` Type

Name: `record_native` **Properties:** C/C++-only, OMPT	**Base Type:** enumeration

Values

Name	Value	Properties
`ompt_record_native_info`	1	C/C++-only, OMPT
`ompt_record_native_event`	2	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_record_native_t {
  ompt_record_native_info  = 1,
  ompt_record_native_event = 2
} ompt_record_native_t;
```

C / C++

Semantics

The **`record_native`** OMPT type indicates the integer codes that identify OMPT native trace record contents.

33.26 OMPT `record_ompt` Type

Name: `record_ompt` **Properties:** C/C++-only, OMPT	**Base Type:** structure

Fields

Name	Type	Properties
type	callbacks	C/C++-only, common-field, OMPT
time	device_time	C/C++-only, OMPT
thread_id	id	C/C++-only, OMPT
target_id	id	C/C++-only, OMPT
record	any_record_ompt	C/C++-only, OMPT

Type Definition

C / C++

```
typedef struct ompt_record_ompt_t {
  ompt_callbacks_t type;
  ompt_device_time_t time;
  ompt_id_t thread_id;
  ompt_id_t target_id;
  ompt_any_record_ompt_t record;
} ompt_record_ompt_t;
```

C / C++

Semantics

The **record_ompt** OMPT type provides a complete trace record by specifying the common fields of the standard trace format along with a field that is an instance of the **any_record_ompt** OMPT type. The **type** field specifies the type of trace record that the structure provides. According to the type, event-specific information is stored in the matching **record** field.

Restrictions

Restrictions to the **record_ompt** OMPT type are as follows:

- If **type** is **ompt_callback_thread_end** then the value of **record** is undefined.

Cross References

- OMPT **any_record_ompt** Type, see Section 33.2
- OMPT **callbacks** Type, see Section 33.6
- OMPT **device_time** Type, see Section 33.12
- OMPT **id** Type, see Section 33.18

33.27 OMPT scope_endpoint Type

Name: **scope_endpoint** **Properties:** C/C++-only, OMPT	**Base Type:** enumeration

Values

Name	Value	Properties
ompt_scope_begin	1	C/C++-only, OMPT
ompt_scope_end	2	C/C++-only, OMPT
ompt_scope_beginend	3	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_scope_endpoint_t {
  ompt_scope_begin    = 1,
  ompt_scope_end      = 2,
  ompt_scope_beginend = 3
} ompt_scope_endpoint_t;
```

C / C++

Summary

The **scope_endpoint** OMPT type defines valid region endpoint values.

33.28 OMPT set_result Type

Name: set_result Properties: C/C++-only, OMPT	Base Type: enumeration

Values

Name	Value	Properties
ompt_set_error	0	C/C++-only, OMPT
ompt_set_never	1	C/C++-only, OMPT
ompt_set_impossible	2	C/C++-only, OMPT
ompt_set_sometimes	3	C/C++-only, OMPT
ompt_set_sometimes_paired	4	C/C++-only, OMPT
ompt_set_always	5	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_set_result_t {
  ompt_set_error            = 0,
  ompt_set_never            = 1,
  ompt_set_impossible       = 2,
  ompt_set_sometimes        = 3,
  ompt_set_sometimes_paired = 4,
  ompt_set_always           = 5
} ompt_set_result_t;
```

C / C++

Summary

The **set_result** OMPT type corresponds to values that the **set_callback**, **set_trace_ompt** and **set_trace_native** entry points return. Its values indicate several possible outcomes. The **ompt_set_error** value indicates that the associated call failed. Otherwise, the value indicates when an event may occur and, when appropriate, callback dispatch leads to the invocation of the callback. The **ompt_set_never** value indicates that the event will never occur or that the callback will never be invoked at runtime. The **ompt_set_impossible** value indicates that the event may occur but that tracing of it is not possible. The **ompt_set_sometimes** value indicates that the event may occur and, for an implementation defined subset of associated event occurrences, will be traced or the callback will be invoked at runtime. The **ompt_set_sometimes_paired** value indicates the same result as **ompt_set_sometimes** and, in addition, that a callback with an *endpoint* value of **ompt_scope_begin** will be invoked if and only if the same callback with an *endpoint* value of **ompt_scope_end** will also be invoked sometime in the future. The **ompt_set_always** value indicates that, whenever an associated event occurs, it will be traced or the callback will be invoked.

Cross References

- OMPT **scope_endpoint** Type, see Section 33.27
- **set_callback** Entry Point, see Section 36.4
- **set_trace_native** Entry Point, see Section 37.5
- **set_trace_ompt** Entry Point, see Section 37.4

33.29 OMPT severity Type

Name: severity **Properties:** C/C++-only, OMPT	**Base Type:** enumeration

Values

Name	Value	Properties
ompt_warning	**1**	C/C++-only, OMPT
ompt_fatal	**2**	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_severity_t {
  ompt_warning = 1,
  ompt_fatal   = 2
} ompt_severity_t;
```

C / C++

Semantics

The **severity** OMPT type defines severity values.

33.30 OMPT `start_tool_result` Type

Name: `start_tool_result` **Properties:** C/C++-only, OMPT	**Base Type:** structure

Fields

Name	Type	Properties
initialize	initialize	C/C++-only, OMPT
finalize	finalize	C/C++-only, OMPT
tool_data	data	C/C++-only, OMPT

Type Definition

C / C++

```
typedef struct ompt_start_tool_result_t {
  ompt_initialize_t initialize;
  ompt_finalize_t finalize;
  ompt_data_t tool_data;
} ompt_start_tool_result_t;
```

C / C++

Semantics

The **`ompt_start_tool`** procedure returns a pointer to a structure of the **`start_tool_result`** OMPT type, which provides pointers to the tool's **`initialize`** and **`finalize`** callbacks as well as a **`data`** object for use by the tool.

Restrictions

Restrictions to the **`start_tool_result`** OMPT type are as follows:

- The **`initialize`** and **`finalize`** callback pointer values in a **`start_tool_result`** structure that **`ompt_start_tool`** returns must be non-null values.

Cross References

- OMPT **`data`** Type, see Section 33.8
- **`finalize`** Callback, see Section 34.1.2
- **`initialize`** Callback, see Section 34.1.1
- **`ompt_start_tool`** Procedure, see Section 32.2.1

33.31 OMPT `state` Type

Name: `state` **Properties:** C/C++-only, OMPT	**Base Type:** enumeration

Values

Name	Value	Properties
ompt_state_work_serial	0x000	C/C++-only, OMPT
ompt_state_work_parallel	0x001	C/C++-only, OMPT
ompt_state_work_reduction	0x002	C/C++-only, OMPT
ompt_state_work_free_agent	0x003	C/C++-only, OMPT
ompt_state_work_induction	0x004	C/C++-only, OMPT
ompt_state_wait_barrier_implicit_parallel	0x011	C/C++-only, OMPT
ompt_state_wait_barrier_implicit_workshare	0x012	C/C++-only, OMPT
ompt_state_wait_barrier_explicit	0x014	C/C++-only, OMPT
ompt_state_wait_barrier_implementation	0x015	C/C++-only, OMPT
ompt_state_wait_barrier_teams	0x016	C/C++-only, OMPT
ompt_state_wait_taskwait	0x020	C/C++-only, OMPT
ompt_state_wait_taskgroup	0x021	C/C++-only, OMPT
ompt_state_wait_mutex	0x040	C/C++-only, OMPT
ompt_state_wait_lock	0x041	C/C++-only, OMPT
ompt_state_wait_critical	0x042	C/C++-only, OMPT
ompt_state_wait_atomic	0x043	C/C++-only, OMPT
ompt_state_wait_ordered	0x044	C/C++-only, OMPT
ompt_state_wait_target	0x080	C/C++-only, OMPT
ompt_state_wait_target_map	0x081	C/C++-only, OMPT
ompt_state_wait_target_update	0x082	C/C++-only, OMPT
ompt_state_idle	0x100	C/C++-only, OMPT
ompt_state_overhead	0x101	C/C++-only, OMPT
ompt_state_undefined	0x102	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_state_t {
  ompt_state_work_serial                      = 0x000,
  ompt_state_work_parallel                    = 0x001,
  ompt_state_work_reduction                   = 0x002,
  ompt_state_work_free_agent                  = 0x003,
  ompt_state_work_induction                   = 0x004,
  ompt_state_wait_barrier_implicit_parallel   = 0x011,
  ompt_state_wait_barrier_implicit_workshare = 0x012,
  ompt_state_wait_barrier_explicit            = 0x014,
  ompt_state_wait_barrier_implementation      = 0x015,
  ompt_state_wait_barrier_teams               = 0x016,
  ompt_state_wait_taskwait                    = 0x020,
  ompt_state_wait_taskgroup                   = 0x021,
  ompt_state_wait_mutex                       = 0x040,
  ompt_state_wait_lock                        = 0x041,
```

```
    ompt_state_wait_critical                    = 0x042,
    ompt_state_wait_atomic                      = 0x043,
    ompt_state_wait_ordered                     = 0x044,
    ompt_state_wait_target                      = 0x080,
    ompt_state_wait_target_map                  = 0x081,
    ompt_state_wait_target_update               = 0x082,
    ompt_state_idle                             = 0x100,
    ompt_state_overhead                         = 0x101,
    ompt_state_undefined                        = 0x102
} ompt_state_t;
```

C / C++

Semantics

The **state** OMPT type defines thread states that indicate the current activity of a thread. If the OMPT interface is in the *active* state then an OpenMP implementation must maintain thread state information for each thread. The thread state maintained is an approximation of the instantaneous state of a thread. A thread state must be one of the values of the **state** OMPT type or an implementation defined state value of 0x200 (512) or higher that extends the OMPT type.

A tool can query the OpenMP thread state at any time. If a tool queries the thread state of a native thread that is not associated with OpenMP then the implementation reports the state as **ompt_state_undefined**.

The **ompt_state_work_serial** value indicates that the thread is executing code outside all **parallel** regions. The **ompt_state_work_parallel** value indicates that the thread is executing code within the scope of a **parallel** region. The **ompt_state_work_reduction** value indicates that the thread is combining partial reduction results from threads in its team. An OpenMP implementation may never report a thread in this state; a thread that is combining partial reduction results may have its state reported as **ompt_state_work_parallel** or **ompt_state_overhead**. The **ompt_state_work_free_agent** value indicates that the thread is executing code within the scope of a task while not being assigned to the current team of that task. The **ompt_state_wait_barrier_implicit_parallel** value indicates that the thread is waiting at the implicit barrier at the end of a **parallel** region. The **ompt_state_wait_barrier_implicit_workshare** value indicates that the thread is waiting at an implicit barrier at the end of a worksharing construct. The **ompt_state_wait_barrier_explicit** value indicates that the thread is waiting in an explicit **barrier** region. The **ompt_state_wait_barrier_implementation** value indicates that the thread is waiting in a barrier that the OpenMP specification does not require but the implementation introduces. The **ompt_state_wait_barrier_teams** value indicates that the thread is waiting at a barrier at the end of a **teams** region. The value **ompt_state_wait_taskwait** indicates that the thread is waiting at a **taskwait** construct. The **ompt_state_wait_taskgroup** value indicates that the thread is waiting at the end of a **taskgroup** construct. The **ompt_state_wait_mutex** value indicates that the thread is waiting for a mutex of an unspecified type. The **ompt_state_wait_lock** value indicates that the thread is waiting for a lock or nestable lock. The **ompt_state_wait_critical** value indicates that the thread is waiting to enter a **critical** region. The **ompt_state_wait_atomic** value indicates that the thread is waiting to enter an **atomic** region. The **ompt_state_wait_ordered** value indicates that the thread is waiting to enter an **ordered** region. The **ompt_state_wait_target** value indicates that the thread is waiting for a **target** region to complete. The **ompt_state_wait_target_map** value indicates that the thread is waiting for a mapping operation to complete. An implementation may report **ompt_state_wait_target** for **target_data** constructs. The **ompt_state_wait_target_update** value indicates that the thread is waiting for a **target_update** operation to complete. An implementation may report **ompt_state_wait_target** for **target_update** constructs. The **ompt_state_idle** value indicates that the native thread is an idle thread, that is, it is an unassigned thread that is not a free-agent thread. The **ompt_state_overhead** value indicates that the thread is in the overhead state at any point while executing within the OpenMP runtime, except while waiting at a synchronization point. The **ompt_state_undefined** value indicates that the native thread is not created by the OpenMP implementation.

33.32 OMPT subvolume Type

Name: subvolume **Properties:** C/C++-only, OMPT	**Base Type:** structure

Fields

Name	Type	Properties
base	c_ptr	C/C++-only, intent(in), value
size	c_uint64_t	value
num_dims	c_uint64_t	value, positive
volume	c_uint64_t	C/C++-only, intent(in), pointer
offsets	c_uint64_t	C/C++-only, intent(in), pointer
dimensions	c_uint64_t	C/C++-only, intent(in), pointer

Type Definition

C / C++

```
typedef struct ompt_subvolume_t {
  const void *base;
  uint64_t size;
  uint64_t num_dims;
  const uint64_t *volume;
  const uint64_t *offsets;
```

```
  const uint64_t *dimensions;
} ompt_subvolume_t;
```

C / C++

Semantics

The **subvolume** OMPT type represents a rectangular subvolume used in a rectangular-memory-copying routine.

Cross References

- Memory Copying Routines, see Section 25.7

33.33 OMPT sync_region Type

Name: sync_region **Properties:** C/C++-only, OMPT, overlapping-type-name	**Base Type:** enumeration

Values

Name	Value	Properties
ompt_sync_region_barrier_explicit	3	C/C++-only, OMPT
ompt_sync_region_barrier_implementation	4	C/C++-only, OMPT
ompt_sync_region_taskwait	5	C/C++-only, OMPT
ompt_sync_region_taskgroup	6	C/C++-only, OMPT
ompt_sync_region_reduction	7	C/C++-only, OMPT
ompt_sync_region_barrier_implicit_workshare	8	C/C++-only, OMPT
ompt_sync_region_barrier_implicit_parallel	9	C/C++-only, OMPT
ompt_sync_region_barrier_teams	10	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_sync_region_t {
  ompt_sync_region_barrier_explicit           = 3,
  ompt_sync_region_barrier_implementation     = 4,
  ompt_sync_region_taskwait                   = 5,
  ompt_sync_region_taskgroup                  = 6,
  ompt_sync_region_reduction                  = 7,
  ompt_sync_region_barrier_implicit_workshare = 8,
  ompt_sync_region_barrier_implicit_parallel  = 9,
  ompt_sync_region_barrier_teams              = 10
} ompt_sync_region_t;
```

C / C++

Semantics

The **sync_region** OMPT type defines the valid synchronization region values.

33.34 OMPT target Type

Name: target **Properties:** C/C++-only, OMPT	**Base Type:** enumeration

Values

Name	Value	Properties
ompt_target	1	C/C++-only, OMPT
ompt_target_enter_data	2	C/C++-only, OMPT
ompt_target_exit_data	3	C/C++-only, OMPT
ompt_target_update	4	C/C++-only, OMPT
ompt_target_nowait	9	C/C++-only, OMPT
ompt_target_enter_data_nowait	10	C/C++-only, OMPT
ompt_target_exit_data_nowait	11	C/C++-only, OMPT
ompt_target_update_nowait	12	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_target_t {
  ompt_target                   = 1,
  ompt_target_enter_data        = 2,
  ompt_target_exit_data         = 3,
  ompt_target_update            = 4,
  ompt_target_nowait            = 9,
  ompt_target_enter_data_nowait = 10,
  ompt_target_exit_data_nowait  = 11,
  ompt_target_update_nowait     = 12
} ompt_target_t;
```

C / C++

Semantics

The **target** OMPT type defines valid values to identify device constructs.

33.35 OMPT target_data_op Type

Name: target_data_op **Properties:** C/C++-only, OMPT	**Base Type:** enumeration

Values

Name	Value	Properties
ompt_target_data_alloc	**1**	C/C++-only, OMPT
ompt_target_data_delete	**4**	C/C++-only, OMPT
ompt_target_data_associate	**5**	C/C++-only, OMPT
ompt_target_data_disassociate	**6**	C/C++-only, OMPT
ompt_target_data_transfer	**7**	C/C++-only, OMPT
ompt_target_data_memset	**8**	C/C++-only, OMPT
ompt_target_data_transfer_rect	**9**	C/C++-only, OMPT
ompt_target_data_alloc_async	**17**	C/C++-only, OMPT
ompt_target_data_delete_async	**20**	C/C++-only, OMPT
ompt_target_data_transfer_async	**23**	C/C++-only, OMPT
ompt_target_data_memset_async	**24**	C/C++-only, OMPT
ompt_target_data_transfer_rect_async	**25**	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_target_data_op_t {
  ompt_target_data_alloc                = 1,
  ompt_target_data_delete               = 4,
  ompt_target_data_associate            = 5,
  ompt_target_data_disassociate         = 6,
  ompt_target_data_transfer             = 7,
  ompt_target_data_memset               = 8,
  ompt_target_data_transfer_rect        = 9,
  ompt_target_data_alloc_async          = 17,
  ompt_target_data_delete_async         = 20,
  ompt_target_data_transfer_async       = 23,
  ompt_target_data_memset_async         = 24,
  ompt_target_data_transfer_rect_async = 25
} ompt_target_data_op_t;
```

C / C++

Additional information

The following instances and associated values of the **target_data_op** OMPT type are also defined: **ompt_target_data_transfer_to_device**, with value 2; **ompt_target_data_transfer_from_device**, with value 3; **ompt_target_data_transfer_to_device_async**, with value 18; and **ompt_target_data_transfer_from_device**, with value 19. These instances have been deprecated.

Semantics

The **target_data_op** OMPT type indicates the kind of target data operation for **target_data_op_emi** callbacks, which can be allocate (**ompt_target_data_alloc** and **ompt_target_data_alloc_async**); delete (**ompt_target_data_delete** and

ompt_target_data_delete_async); associate (**ompt_target_data_associate**); disassociate (**ompt_target_data_disassociate**); transfer (**ompt_target_data_transfer**, **ompt_target_data_transfer_async**, **ompt_target_data_transfer_rect**, and **ompt_target_data_transfer_rect_async**); or memset (**ompt_target_data_memset** and **ompt_target_data_memset_async**), where the values that end with **_async** correspond to asynchronous data operations.

33.36 OMPT target_map_flag Type

Name: **target_map_flag**	Base Type: enumeration
Properties: C/C++-only, OMPT	

Values

Name	Value	Properties
ompt_target_map_flag_to	**0x01**	C/C++-only, OMPT
ompt_target_map_flag_from	**0x02**	C/C++-only, OMPT
ompt_target_map_flag_alloc	**0x04**	C/C++-only, OMPT
ompt_target_map_flag_release	**0x08**	C/C++-only, OMPT
ompt_target_map_flag_delete	**0x10**	C/C++-only, OMPT
ompt_target_map_flag_implicit	**0x20**	C/C++-only, OMPT
ompt_target_map_flag_always	**0x40**	C/C++-only, OMPT
ompt_target_map_flag_present	**0x80**	C/C++-only, OMPT
ompt_target_map_flag_close	**0x100**	C/C++-only, OMPT
ompt_target_map_flag_shared	**0x200**	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_target_map_flag_t {
  ompt_target_map_flag_to       = 0x01,
  ompt_target_map_flag_from     = 0x02,
  ompt_target_map_flag_alloc    = 0x04,
  ompt_target_map_flag_release  = 0x08,
  ompt_target_map_flag_delete   = 0x10,
  ompt_target_map_flag_implicit = 0x20,
  ompt_target_map_flag_always   = 0x40,
  ompt_target_map_flag_present  = 0x80,
  ompt_target_map_flag_close    = 0x100,
  ompt_target_map_flag_shared   = 0x200
} ompt_target_map_flag_t;
```

C / C++

Semantics

The **target_map_flag** OMPT type defines the valid map flag values. The **ompt_target_map_flag_to**, **ompt_target_map_flag_from**,

ompt_target_map_flag_alloc, and **ompt_target_map_flag_release** values are set when the mapping operations have the corresponding *map-type*. If the *map-type* for the mapping operations is **tofrom**, both the **ompt_target_map_flag_to** and **ompt_target_map_flag_from** values are set. The **ompt_target_map_flag_implicit** value is set if the mapping operations correspond to implicitly determined data-mapping attributes. The **ompt_target_map_flag_delete**, **ompt_target_map_flag_always**, **ompt_target_map_flag_present**, and **ompt_target_map_flag_close**, values are set if the mapping operations are specified with the corresponding *map-type-modifier* modifiers. The **ompt_target_map_flag_shared** value is set if the original storage and corresponding storage are shared for the mapping operation.

33.37 OMPT task_flag Type

Name: task_flag Properties: C/C++-only, OMPT	Base Type: enumeration

Values

Name	Value	Properties
ompt_task_initial	0x00000001	C/C++-only, OMPT
ompt_task_implicit	0x00000002	C/C++-only, OMPT
ompt_task_explicit	0x00000004	C/C++-only, OMPT
ompt_task_target	0x00000008	C/C++-only, OMPT
ompt_task_taskwait	0x00000010	C/C++-only, OMPT
ompt_task_importing	0x02000000	C/C++-only, OMPT
ompt_task_exporting	0x04000000	C/C++-only, OMPT
ompt_task_undeferred	0x08000000	C/C++-only, OMPT
ompt_task_untied	0x10000000	C/C++-only, OMPT
ompt_task_final	0x20000000	C/C++-only, OMPT
ompt_task_mergeable	0x40000000	C/C++-only, OMPT
ompt_task_merged	0x80000000	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_task_flag_t {
  ompt_task_initial     = 0x00000001,
  ompt_task_implicit    = 0x00000002,
  ompt_task_explicit    = 0x00000004,
  ompt_task_target      = 0x00000008,
  ompt_task_taskwait    = 0x00000010,
  ompt_task_importing   = 0x02000000,
  ompt_task_exporting   = 0x04000000,
  ompt_task_undeferred = 0x08000000,
  ompt_task_untied      = 0x10000000,
  ompt_task_final       = 0x20000000,
  ompt_task_mergeable   = 0x40000000,
  ompt_task_merged      = 0x80000000
} ompt_task_flag_t;
```

C / C++

Semantics

The **task_flag** OMPT type defines valid task values. The least significant byte provides information about the general classification of the task. The other bits represent its properties.

33.38 OMPT task_status Type

Name: task_status **Properties:** C/C++-only, OMPT	**Base Type:** enumeration

Values

Name	Value	Properties
ompt_task_complete	1	C/C++-only, OMPT
ompt_task_yield	2	C/C++-only, OMPT
ompt_task_cancel	3	C/C++-only, OMPT
ompt_task_detach	4	C/C++-only, OMPT
ompt_task_early_fulfill	5	C/C++-only, OMPT
ompt_task_late_fulfill	6	C/C++-only, OMPT
ompt_task_switch	7	C/C++-only, OMPT
ompt_taskwait_complete	8	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_task_status_t {
  ompt_task_complete       = 1,
  ompt_task_yield          = 2,
  ompt_task_cancel         = 3,
  ompt_task_detach         = 4,
  ompt_task_early_fulfill = 5,
  ompt_task_late_fulfill  = 6,
  ompt_task_switch         = 7,
  ompt_taskwait_complete  = 8
} ompt_task_status_t;
```

C / C++

Semantics

The **task_status** OMPT type indicates the reason that a task was switched when it reached a task scheduling point. The **ompt_task_complete** value indicates that the task that encountered the task scheduling point completed execution of its associated structured block and any associated *allow-completion* event was fulfilled. The **ompt_task_yield** value indicates that the task encountered a **taskyield** construct. The **ompt_task_cancel** value indicates that the task was canceled when it encountered an active cancellation point. The **ompt_task_detach** value indicates that a task for which the **detach** clause was specified completed execution of the associated structured block and is waiting for an *allow-completion* event to be fulfilled. The **ompt_task_early_fulfill** value indicates that the *allow-completion* event of the task was fulfilled before the task completed execution of the associated structured block. The **ompt_task_late_fulfill** value indicates that the *allow-completion* event of the task was fulfilled after the task completed execution of the associated structured block. The **ompt_taskwait_complete** value indicates completion of the dependent task that results from a **taskwait** construct with one or more **depend** clauses. The **ompt_task_switch** value is used for all other cases that a task was switched.

33.39 OMPT thread Type

Name: thread **Properties:** C/C++-only, OMPT	**Base Type:** enumeration

Values

Name	Value	Properties
ompt_thread_initial	1	C/C++-only, OMPT
ompt_thread_worker	2	C/C++-only, OMPT
ompt_thread_other	3	C/C++-only, OMPT
ompt_thread_unknown	4	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_thread_t {
  ompt_thread_initial = 1,
  ompt_thread_worker  = 2,
  ompt_thread_other   = 3,
  ompt_thread_unknown = 4
} ompt_thread_t;
```

C / C++

Semantics

The **thread** OMPT type defines the valid thread type values. Any initial thread has thread type **ompt_thread_initial**. All threads that are thread-pool-worker threads have thread type **ompt_thread_worker**. A native thread that an OpenMP implementation uses but that does not execute user code has thread type **ompt_thread_other**. Any native thread that is created outside an OpenMP implementation and that is not an initial thread has thread type **ompt_thread_unknown**.

33.40 OMPT wait_id Type

Name: **wait_id** **Properties:** C/C++-only, OMPT	**Base Type: c_uint64_t**

Predefined Identifiers

Name	Value	Properties
ompt_wait_id_none	0	C/C++-only, OMPT

Type Definition

C / C++

```
typedef uint64_t ompt_wait_id_t;
```

C / C++

Semantics

The **wait_id** OMPT type describes wait identifiers for a thread; each thread maintains one of these wait identifiers. When a task that a thread executes is waiting for mutual exclusion, the wait identifier of the thread indicates the reason that the thread is waiting. A wait identifier may represent the *name* argument of a critical section, or a lock, or a variable accessed in an **atomic** region, or a synchronization object that is internal to an OpenMP implementation. When a thread is not in a wait state then the value of the wait identifier of the thread is undefined.

33.41 OMPT work Type

Name: **work** **Properties:** C/C++-only, OMPT, overlapping-type-name	**Base Type:** enumeration

Values

Name	Value	Properties
ompt_work_loop	1	C/C++-only, OMPT
ompt_work_sections	2	C/C++-only, OMPT
ompt_work_single_executor	3	C/C++-only, OMPT
ompt_work_single_other	4	C/C++-only, OMPT
ompt_work_workshare	5	C/C++-only, OMPT
ompt_work_distribute	6	C/C++-only, OMPT
ompt_work_taskloop	7	C/C++-only, OMPT
ompt_work_scope	8	C/C++-only, OMPT
ompt_work_workdistribute	9	C/C++-only, OMPT
ompt_work_loop_static	10	C/C++-only, OMPT
ompt_work_loop_dynamic	11	C/C++-only, OMPT
ompt_work_loop_guided	12	C/C++-only, OMPT
ompt_work_loop_other	13	C/C++-only, OMPT

Type Definition

C / C++

```
typedef enum ompt_work_t {
  ompt_work_loop              = 1,
  ompt_work_sections          = 2,
  ompt_work_single_executor = 3,
  ompt_work_single_other    = 4,
  ompt_work_workshare        = 5,
  ompt_work_distribute       = 6,
  ompt_work_taskloop         = 7,
  ompt_work_scope            = 8,
  ompt_work_workdistribute  = 9,
  ompt_work_loop_static     = 10,
  ompt_work_loop_dynamic    = 11,
  ompt_work_loop_guided     = 12,
  ompt_work_loop_other      = 13
} ompt_work_t;
```

C / C++

Semantics

The **work** OMPT type defines the valid work values.

34 General Callbacks and Trace Records

This chapter describes general OMPT callbacks that an OMPT tool may register and that are called during the runtime of an OpenMP program. The C/C++ header file (**omp-tools.h**) provides the types that this chapter defines. Tool implementations of callbacks are not required to be async signal safe.

Several OMPT callbacks include a *codeptr_ra* argument that relates the implementation of an OpenMP region to its source code. If a routine implements the region associated with a callback then *codeptr_ra* contains the return address of the call to that routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the callback invocation. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be NULL.

Several OMPT callbacks have a *flags* argument; the meaning and valid values for that argument is described with the callback. Some callbacks have an *encountering_task_frame* argument that points to the frame object that is associated with the encountering task. The behavior for accessing the frame object after the callback returns is unspecified. Some callbacks have a *tool_data* argument that is a pointer to the **tool_data** field in the **start_tool_result** structure that **ompt_start_tool** returned. Some callbacks have a *parallel_data* argument; the binding of these arguments is the **parallel** or **teams** region that is beginning or ending or the current parallel region for callbacks that are dispatched during the execution of one. Some callbacks have an *encountering_task_data* argument; the binding of these arguments is the encountering task. Some callbacks have an *endpoint* argument that indicates whether the callback signals that a region begins or ends. Some callbacks have a *wait_id* argument, which indicates the object being awaited.

Several OMPT callbacks have a *task_data* argument; unless otherwise specified, the binding of these arguments is the encountering task of the event for which the implementation dispatches the callback. For some of those callbacks, OpenMP semantics imply that this task to which the *task_data* argument binds is the implicit task that executes the structured block of the binding parallel region or **teams** region.

An implementation may also provide a trace of events per device. Along with the callbacks, this chapter also defines standard trace records. For these trace records, unless otherwise specified, tool data arguments are replaced by an ID, which must be initialized by the OpenMP implementation. Each of **parallel_id**, **task_id**, and **thread_id** must be unique per **target** region. If the **target_emi** callback is dispatched, the **target_id** used in any trace records associated with the device region is given by the **value** field of the *target_data* **data** object that is set in the callback.

Restrictions

Restrictions to OpenMP tool callbacks are as follows:

- Tool callbacks may not use directives or call any routines.
- Tool callbacks must exit by either returning to the caller or aborting.

34.1 Initialization and Finalization Callbacks

This section describes callbacks that are called to initialize and to finalize tools and when native threads are initialized and finalized.

34.1.1 `initialize` Callback

Name: `initialize` **Category:** function	**Properties:** C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
lookup	function_lookup	OMPT
initial_device_num	integer	*default*
tool_data	data	OMPT, pointer

Type Signature

C / C++

```
typedef int (*ompt_initialize_t) (ompt_function_lookup_t lookup,
  int initial_device_num, ompt_data_t *tool_data);
```

C / C++

Semantics

A tool provides an **`initialize`** callback, which has the **`initialize`** OMPT type, in the non-null pointer to a **`start_tool_result`** OMPT type structure that its implementation of **`ompt_start_tool`** returns. An OpenMP implementation must call this OMPT-tool initializer after fully initializing itself but before beginning execution of any construct or routine. An **`initialize`** callback returns a non-zero value if it succeeds; otherwise, the OMPT interface state changes to OMPT inactive as described in Section 32.2.3.

The *lookup* argument of an **`initialize`** callback is a pointer to a runtime entry point that a tool must use to obtain pointers to the other entry points in the OMPT interface. The *initial_device_num* argument provides the value that a call to **`omp_get_initial_device`** would return.

C / C++

A callback of **`initialize`** OMPT type is a callback of type **`ompt_initialize_t`**.

C / C++

Cross References

- OMPT **`data`** Type, see Section 33.8
- **`omp_get_initial_device`** Routine, see Section 24.10
- **`ompt_start_tool`** Procedure, see Section 32.2.1
- OMPT **`start_tool_result`** Type, see Section 33.30

34.1.2 `finalize` Callback

Name: `finalize` **Category:** subroutine	**Properties:** C/C++-only, OMPT

Arguments

Name	Type	Properties
tool_data	data	OMPT, pointer

Type Signature

C / C++

```
typedef void (*ompt_finalize_t) (ompt_data_t *tool_data);
```

C / C++

Semantics

A tool provides a **`finalize`** callback, which has the **`finalize`** OMPT type, in the non-null pointer to a **`start_tool_result`** OMPT type structure that its implementation of **`ompt_start_tool`** returns. An OpenMP implementation must call this OMPT-tool finalizer after the last OMPT event as the OpenMP implementation shuts down.

C / C++

A callback of **`finalize`** OMPT type is a callback of type **`ompt_finalize_t`**.

C / C++

Cross References

- OMPT **`data`** Type, see Section 33.8
- **`ompt_start_tool`** Procedure, see Section 32.2.1
- OMPT **`start_tool_result`** Type, see Section 33.30

34.1.3 `thread_begin` Callback

Name: `thread_begin` **Category:** subroutine	**Properties:** C/C++-only, OMPT

Arguments

Name	Type	Properties
thread_type	thread	OMPT
thread_data	data	OMPT, pointer, untraced-argument

Type Signature

C / C++

```
typedef void (*ompt_callback_thread_begin_t) (
  ompt_thread_t thread_type, ompt_data_t *thread_data);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_thread_begin_t {
  ompt_thread_t thread_type;
} ompt_record_thread_begin_t;
```

C / C++

Semantics

A tool provides a **thread_begin** callback, which has the **thread_begin** OMPT type, that the OpenMP implementation dispatches when native threads are created. The *thread_type* argument indicates the type of the new thread: initial, worker, other, or unknown. The binding of the *thread_data* argument is the new thread.

Cross References

- OMPT **data** Type, see Section 33.8
- OMPT **thread** Type, see Section 33.39

34.1.4 thread_end Callback

Name: thread_end Category: subroutine	Properties: C/C++-only, OMPT

Arguments

Name	Type	Properties
thread_data	data	OMPT, pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_thread_end_t) (
  ompt_data_t *thread_data);
```

C / C++

Semantics

A tool provides a **thread_end** callback, which has the **thread_end** OMPT type, that the OpenMP implementation dispatches when native threads are destroyed. The binding of the *thread_data* argument is the thread that will be destroyed.

Cross References

- OMPT **data** Type, see Section 33.8

34.2 `error` Callback

Name: `error` Category: subroutine	Properties: C/C++-only, OMPT

Arguments

Name	Type	Properties
severity	severity	OMPT
message	char	intent(in), pointer
length	size_t	*default*
codeptr_ra	void	intent(in), pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_error_t) (ompt_severity_t severity,
  const char *message, size_t length, const void *codeptr_ra);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_error_t {
  ompt_severity_t severity;
  const char *message;
  size_t length;
  const void *codeptr_ra;
} ompt_record_error_t;
```

C / C++

Semantics

A tool provides an **`error`** callback, which has the **`error`** OMPT type, that the OpenMP implementation dispatches when an **`error`** directive is encountered for which the *action-time* argument of the **`at`** clause is specified as **`execution`**. The *severity* argument passes the specified severity level. The *message* argument passes the C string from the **`message`** clause. The *length* argument provides the length of the C string.

Cross References

- **`error`** Directive, see Section 10.1
- OMPT **`severity`** Type, see Section 33.29

34.3 Parallelism Generation Callback Signatures

This section describes callbacks that are related to constructs for generating and controlling parallelism.

34.3.1 parallel_begin Callback

Name: **parallel_begin** **Category:** subroutine	**Properties:** C/C++-only, OMPT

Arguments

Name	Type	Properties
encountering_task_data	data	OMPT, pointer
encountering_task_frame	frame	intent(in), OMPT, pointer, untraced-argument
parallel_data	data	OMPT, pointer
requested_parallelism	integer	unsigned
flags	integer	*default*
codeptr_ra	void	intent(in), pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_parallel_begin_t) (
  ompt_data_t *encountering_task_data,
  const ompt_frame_t *encountering_task_frame,
  ompt_data_t *parallel_data, unsigned int requested_parallelism,
  int flags, const void *codeptr_ra);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_parallel_begin_t {
  ompt_id_t encountering_task_id;
  ompt_id_t parallel_id;
  unsigned int requested_parallelism;
  int flags;
  const void *codeptr_ra;
} ompt_record_parallel_begin_t;
```

C / C++

Semantics

A tool provides a **parallel_begin** callback, which has the **parallel_begin** OMPT type, that the OpenMP implementation dispatches when a **parallel** or **teams** region starts. The *requested_parallelism* argument indicates the number of threads or teams that the user requested. The *flags* argument indicates whether the code for the region is inlined into the application or invoked by the runtime and also whether the region is a **parallel** or **teams** region. Valid values for *flags* are a disjunction of elements in the **parallel_flag** OMPT type.

Cross References

- OMPT **data** Type, see Section 33.8
- OMPT **frame** Type, see Section 33.15
- OMPT **id** Type, see Section 33.18
- **parallel** Construct, see Section 12.1
- OMPT **parallel_flag** Type, see Section 33.22
- **teams** Construct, see Section 12.2

34.3.2 parallel_end Callback

Name: **parallel_end** Category: subroutine	**Properties:** C/C++-only, OMPT

Arguments

Name	Type	Properties
parallel_data	data	OMPT, pointer
encountering_task_data	data	OMPT, pointer
flags	integer	*default*
codeptr_ra	void	intent(in), pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_parallel_end_t) (
  ompt_data_t *parallel_data, ompt_data_t *encountering_task_data,
  int flags, const void *codeptr_ra);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_parallel_end_t {
  ompt_id_t parallel_id;
  ompt_id_t encountering_task_id;
  int flags;
  const void *codeptr_ra;
} ompt_record_parallel_end_t;
```

C / C++

Semantics

A tool provides a **parallel_end** callback, which has the **parallel_end** OMPT type, that the OpenMP implementation dispatches when a **parallel** or **teams** region ends. The *flags* argument indicates whether the code for the region is inlined into the application or invoked by the runtime and also whether the region is a **parallel** or **teams** region. Valid values for *flags* are a disjunction of elements in the **parallel_flag** OMPT type.

Cross References

- OMPT **data** Type, see Section 33.8
- OMPT **id** Type, see Section 33.18
- **parallel** Construct, see Section 12.1
- OMPT **parallel_flag** Type, see Section 33.22
- **teams** Construct, see Section 12.2

34.3.3 masked Callback

Name: **masked** Category: subroutine	Properties: C/C++-only, OMPT

Arguments

Name	Type	Properties
endpoint	scope_endpoint	OMPT
parallel_data	data	OMPT, pointer
task_data	data	OMPT, pointer
codeptr_ra	void	intent(in), pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_masked_t) (
  ompt_scope_endpoint_t endpoint, ompt_data_t *parallel_data,
  ompt_data_t *task_data, const void *codeptr_ra);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_masked_t {
  ompt_scope_endpoint_t endpoint;
  ompt_id_t parallel_id;
  ompt_id_t task_id;
  const void *codeptr_ra;
} ompt_record_masked_t;
```

C / C++

Semantics

A tool provides a **masked** callback, which has the **masked** OMPT type, that the OpenMP implementation dispatches for **masked** regions.

Cross References

- OMPT **data** Type, see Section 33.8
- **masked** Construct, see Section 12.5
- OMPT **id** Type, see Section 33.18
- OMPT **scope_endpoint** Type, see Section 33.27

34.4 Work Distribution Callback Signatures

This section describes callbacks that are related to work-distribution constructs.

34.4.1 work Callback

Name: work **Category:** subroutine	**Properties:** C/C++-only, OMPT, overlapping-type-name

Arguments

Name	Type	Properties
work_type	work	OMPT, overlapping-type-name
endpoint	scope_endpoint	OMPT
parallel_data	data	OMPT, pointer
task_data	data	OMPT, pointer
count	c_uint64_t	*default*
codeptr_ra	void	intent(in), pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_work_t) (ompt_work_t work_type,
  ompt_scope_endpoint_t endpoint, ompt_data_t *parallel_data,
  ompt_data_t *task_data, uint64_t count, const void *codeptr_ra);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_work_t {
  ompt_work_t work_type;
  ompt_scope_endpoint_t endpoint;
  ompt_id_t parallel_id;
  ompt_id_t task_id;
  uint64_t count;
  const void *codeptr_ra;
} ompt_record_work_t;
```

C / C++

Semantics

A tool provides a **work** callback, which has the **work** OMPT type, that the OpenMP implementation dispatches for worksharing regions and **taskloop** regions. The *work_type* argument indicates the kind of region. The *count* argument is a measure of the quantity of work involved in the construct. For a worksharing-loop construct or **taskloop** construct, *count* represents the number of collapsed iterations. For a **sections** construct, *count* represents the number of sections. For a **workshare** or **workdistribute** construct, *count* represents the units of work, as defined by the **workshare** or **workdistribute** construct. For a **single** or **scope** construct, *count* is always 1. When the *endpoint* argument signals the end of a region, a *count* value of 0 indicates that the actual *count* value is not available.

Cross References

- OMPT **data** Type, see Section 33.8
- Work-Distribution Constructs, see Chapter 13
- OMPT **id** Type, see Section 33.18
- OMPT **scope_endpoint** Type, see Section 33.27
- **taskloop** Construct, see Section 14.2
- OMPT **work** Type, see Section 33.41

34.4.2 `dispatch` Callback

Name: dispatch **Category:** subroutine	**Properties:** C/C++-only, OMPT, overlapping-type-name

Arguments

Name	Type	Properties
parallel_data	data	OMPT, pointer
task_data	data	OMPT, pointer
kind	dispatch	OMPT, overlapping-type-name
instance	data	OMPT

Type Signature

C / C++

```
typedef void (*ompt_callback_dispatch_t) (
  ompt_data_t *parallel_data, ompt_data_t *task_data,
  ompt_dispatch_t kind, ompt_data_t instance);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_dispatch_t {
  ompt_id_t parallel_id;
  ompt_id_t task_id;
  ompt_dispatch_t kind;
  ompt_id_t instance;
} ompt_record_dispatch_t;
```

C / C++

Semantics

A tool provides a **dispatch** callback, which has the **dispatch** OMPT type (which has an overlapping type name with the **dispatch** OMPT type that applies to the *kind* argument of the callback), that the OpenMP implementation dispatches when a thread begins to execute a section or a collapsed iteration. The *kind* argument indicates whether a collapsed iteration or a section is being dispatched. If the *kind* argument is **ompt_dispatch_iteration**, the **value** field of the *instance* argument contains the logical iteration number. If the *kind* argument is **ompt_dispatch_section**, the **ptr** field of the *instance* argument contains a code address that identifies the structured block. In cases where a routine implements the structured block associated with this callback, the **ptr** field of the *instance* argument contains the return address of the call to the routine. In cases where the implementation of the structured block is inlined, the **ptr** field of the *instance* argument contains the return address of the invocation of this callback. If the *kind* argument is **ompt_dispatch_ws_loop_chunk**, **ompt_dispatch_taskloop_chunk** or **ompt_dispatch_distribute_chunk**, the **ptr** field of the *instance* argument points to a structure of type **dispatch_chunk** that contains the information for the chunk.

Cross References

- OMPT **data** Type, see Section 33.8
- OMPT **dispatch** Type, see Section 33.13
- OMPT **dispatch_chunk** Type, see Section 33.14
- Worksharing-Loop Constructs, see Section 13.6
- OMPT **id** Type, see Section 33.18
- **sections** Construct, see Section 13.3
- **taskloop** Construct, see Section 14.2

34.5 Tasking Callback Signatures

This section describes callbacks that are related to tasks.

34.5.1 task_create Callback

Name: task_create Category: subroutine	Properties: C/C++-only, OMPT

Arguments

Name	Type	Properties
encountering_task_data	data	OMPT, pointer
encountering_task_frame	frame	intent(in), OMPT, pointer, untraced-argument
new_task_data	data	OMPT, pointer
flags	integer	*default*
has_dependences	integer	*default*
codeptr_ra	void	intent(in), pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_task_create_t) (
  ompt_data_t *encountering_task_data,
  const ompt_frame_t *encountering_task_frame,
  ompt_data_t *new_task_data, int flags, int has_dependences,
  const void *codeptr_ra);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_task_create_t {
  ompt_id_t encountering_task_id;
  ompt_id_t new_task_id;
  int flags;
  int has_dependences;
  const void *codeptr_ra;
} ompt_record_task_create_t;
```

C / C++

Semantics

A tool provides a **task_create** callback, which has the **task_create** OMPT type, that the OpenMP implementation dispatches when task regions are generated. The binding of the *new_task_data* argument is the generated task. The *flags* argument indicates the kind of task (explicit task or target task) that is generated. Values for *flags* are a disjunction of elements in the **task_flag** OMPT type. The *has_dependences* argument is *true* if the generated task has dependences and *false* otherwise.

Cross References

- OMPT **data** Type, see Section 33.8
- OMPT **frame** Type, see Section 33.15
- Initial Task, see Section 14.13
- OMPT **id** Type, see Section 33.18
- **task** Construct, see Section 14.1
- OMPT **task_flag** Type, see Section 33.37

34.5.2 task_schedule Callback

Name: **task_schedule** **Category:** subroutine	**Properties:** C/C++-only, OMPT

Arguments

Name	Type	Properties
prior_task_data	data	OMPT, pointer
prior_task_status	task_status	OMPT
next_task_data	data	OMPT, pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_task_schedule_t) (
  ompt_data_t *prior_task_data,
  ompt_task_status_t prior_task_status,
  ompt_data_t *next_task_data);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_task_schedule_t {
  ompt_id_t prior_task_id;
  ompt_task_status_t prior_task_status;
  ompt_id_t next_task_id;
} ompt_record_task_schedule_t;
```

C / C++

Semantics

A tool provides a **task_schedule** callback, which has the **task_schedule** OMPT type, that the OpenMP implementation dispatches when task scheduling decisions are made. The binding of the *prior_task_data* argument is the task that arrived at the task scheduling point. This argument can be NULL if no task was active when the next task is scheduled. The *prior_task_status* argument indicates the status of that prior task. The binding of the *next_task_data* argument is the task that is resumed at the task scheduling point. This argument is NULL if the callback is dispatched for a *task-fulfill* event or if the callback signals completion of a **taskwait** construct. This argument can be NULL if no task was active when the prior task was scheduled.

Cross References

- OMPT **data** Type, see Section 33.8
- Task Scheduling, see Section 14.14
- OMPT **id** Type, see Section 33.18
- OMPT **task_status** Type, see Section 33.38

34.5.3 implicit_task Callback

Name: implicit_task **Category:** subroutine	**Properties:** C/C++-only, OMPT

Arguments

Name	Type	Properties
endpoint	scope_endpoint	OMPT
parallel_data	data	OMPT, pointer
task_data	data	OMPT, pointer
actual_parallelism	integer	unsigned
index	integer	unsigned
flags	integer	*default*

Type Signature

C / C++

```
typedef void (*ompt_callback_implicit_task_t) (
  ompt_scope_endpoint_t endpoint, ompt_data_t *parallel_data,
  ompt_data_t *task_data, unsigned int actual_parallelism,
  unsigned int index, int flags);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_implicit_task_t {
  ompt_scope_endpoint_t endpoint;
  ompt_id_t parallel_id;
  ompt_id_t task_id;
  unsigned int actual_parallelism;
  unsigned int index;
  int flags;
} ompt_record_implicit_task_t;
```

C / C++

Semantics

A tool provides an **implicit_task** callback, which has the **implicit_task** OMPT type, that the OpenMP implementation dispatches when initial tasks and implicit tasks are generated and completed. The *flags* argument, which has the **task_flag** OMPT type, indicates the kind of task (initial task or implicit task). For the *implicit-task-end* and the *initial-task-end* events, the *parallel_data* argument is NULL.

The *actual_parallelism* argument indicates the number of threads in the **parallel** region or the number of teams in the **teams** region. For initial tasks that are not closely nested in a **teams** construct, this argument is **1**. For the *implicit-task-end* and the *initial-task-end* events, this argument is **0**.

The *index* argument indicates the thread number or team number of the calling thread, within the team or league that is executing the parallel region or **teams** region to which the implicit task region binds. For initial tasks that are not created by a **teams** construct, this argument is **1**.

Cross References

- OMPT **data** Type, see Section 33.8
- OMPT **id** Type, see Section 33.18
- **parallel** Construct, see Section 12.1
- OMPT **scope_endpoint** Type, see Section 33.27
- OMPT **task_flag** Type, see Section 33.37
- **teams** Construct, see Section 12.2

34.6 cancel Callback

Name: **cancel** Category: subroutine	Properties: C/C++-only, OMPT

Arguments

Name	Type	Properties
task_data	data	OMPT, pointer
flags	integer	*default*
codeptr_ra	void	intent(in), pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_cancel_t) (ompt_data_t *task_data,
  int flags, const void *codeptr_ra);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_cancel_t {
  ompt_id_t task_id;
  int flags;
  const void *codeptr_ra;
} ompt_record_cancel_t;
```

C / C++

Semantics

A tool provides a **cancel** callback, which has the **cancel** OMPT type, that the OpenMP implementation dispatches when *cancellation*, *cancel* and *discarded-task* events occur. The *flags* argument, which is defined by the **cancel_flag** OMPT type, indicates whether cancellation is activated by the encountering task or detected as being activated by another task. The construct that is being canceled is also described in the *flags* argument. When several constructs are detected as being concurrently canceled, each corresponding bit in the argument will be set.

Cross References

- OMPT **cancel_flag** Type, see Section 33.7
- OMPT **data** Type, see Section 33.8
- OMPT **id** Type, see Section 33.18

34.7 Synchronization Callback Signatures

This section describes callbacks that are related to synchronization constructs and clauses.

34.7.1 dependences Callback

Name: **dependences** Category: subroutine	**Properties:** C/C++-only, OMPT

Arguments

Name	Type	Properties
task_data	data	OMPT, pointer
deps	dependence	intent(in), pointer
ndeps	integer	*default*

Type Signature

C / C++

```
typedef void (*ompt_callback_dependences_t) (
  ompt_data_t *task_data, const ompt_dependence_t *deps, int ndeps);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_dependences_t {
  ompt_id_t task_id;
  ompt_dependence_t dep;
  int ndeps;
} ompt_record_dependences_t;
```

C / C++

Semantics

A tool provides a **dependences** callback, which has the **dependences** OMPT type, that the OpenMP implementation dispatches when tasks are generated and when **ordered** constructs are encountered. The binding of the *task_data* argument is the generated task for a **depend** clause on a **task** construct, the target task for a **depend** clause on a device construct, the depend object in an asynchronous routine, or the encountering task for a **doacross** clause of the **ordered** construct. The *deps* argument points to an array of structures of **dependence** OMPT type that represent dependences of the generated task or the *iteration-specifier* of the **doacross** clause. Dependences denoted with depend objects are described in terms of their dependence semantics. The *ndeps* argument specifies the length of the list passed by the *deps* argument. The memory for *deps* is owned by the caller; the tool cannot rely on the data after the callback returns.

When the implementation logs **dependences** trace records for a given event, the **ndeps** field determines the number of trace records that are logged, one for each dependence. The **dep** field in a given trace record denotes a structure of **dependence** OMPT type that represents the dependence.

Cross References

- OMPT **data** Type, see Section 33.8
- **depend** Clause, see Section 17.9.5
- OMPT **dependence** Type, see Section 33.9
- OMPT **id** Type, see Section 33.18
- Stand-alone **ordered** Construct, see Section 17.10.1

34.7.2 task_dependence Callback

Name: task_dependence **Category:** subroutine	**Properties:** C/C++-only, OMPT

Arguments

Name	Type	Properties
src_task_data	data	OMPT, pointer
sink_task_data	data	OMPT, pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_task_dependence_t) (
  ompt_data_t *src_task_data, ompt_data_t *sink_task_data);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_task_dependence_t {
  ompt_id_t src_task_id;
  ompt_id_t sink_task_id;
} ompt_record_task_dependence_t;
```

C / C++

Semantics

A tool provides a **task_dependence** callback, which has the **task_dependence** OMPT type, that the OpenMP implementation dispatches when it encounters an unfulfilled task dependence. The binding of the *src_task_data* argument is an uncompleted antecedent task. The binding of the *sink_task_data* argument is a corresponding dependent task.

Cross References

- OMPT **data** Type, see Section 33.8
- **depend** Clause, see Section 17.9.5
- OMPT **id** Type, see Section 33.18

34.7.3 OMPT sync_region Type

Name: **sync_region** **Category:** subroutine pointer	**Properties:** C/C++-only, OMPT, overlapping-type-name

Arguments

Name	Type	Properties
kind	sync_region	OMPT
endpoint	scope_endpoint	OMPT
parallel_data	data	OMPT, pointer
task_data	data	OMPT, pointer
codeptr_ra	void	intent(in), pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_sync_region_t) (
  ompt_sync_region_t kind, ompt_scope_endpoint_t endpoint,
  ompt_data_t *parallel_data, ompt_data_t *task_data,
  const void *codeptr_ra);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_sync_region_t {
  ompt_sync_region_t kind;
  ompt_scope_endpoint_t endpoint;
  ompt_id_t parallel_id;
  ompt_id_t task_id;
  const void *codeptr_ra;
} ompt_record_sync_region_t;
```

C / C++

Semantics

Callbacks that have the **sync_region** OMPT type are synchronizing-region callbacks, which each have the synchronizing-region property. A tool provides these callbacks to mark the beginning and end of regions that have synchronizing semantics. The *kind* argument, which has the **sync_region** OMPT type, indicates the kind of synchronization.

Cross References

- OMPT **data** Type, see Section 33.8
- OMPT **id** Type, see Section 33.18
- OMPT **scope_endpoint** Type, see Section 33.27
- OMPT **sync_region** Type, see Section 33.33

34.7.4 sync_region Callback

Name: sync_region Category: subroutine	Properties: C/C++-only, common-type-callback, synchronizing-region, OMPT

Type Signature

sync_region

Semantics

A tool provides a **sync_region** callback, which has the **sync_region** OMPT type, that the OpenMP implementation dispatches when barrier regions, **taskwait** regions, and **taskgroup** regions begin and end. For the *implicit-barrier-end* event at the end of a parallel region, *parallel_data* argument is NULL.

Cross References

- **barrier** Construct, see Section 17.3.1
- Implicit Barriers, see Section 17.3.2
- OMPT **sync_region** Type, see Section 34.7.3
- **taskgroup** Construct, see Section 17.4
- **taskwait** Construct, see Section 17.5

34.7.5 sync_region_wait Callback

Name: sync_region_wait Category: subroutine	Properties: C/C++-only, common-type-callback, synchronizing-region, OMPT

Type Signature

sync_region

Semantics

A tool provides a **sync_region_wait** callback, which has the **sync_region** OMPT type, that the OpenMP implementation dispatches when waiting begins and ends for barrier regions, **taskwait** regions, and **taskgroup** regions. For the *implicit-barrier-wait-begin* and *implicit-barrier-wait-end* events at the end of a parallel region, whether *parallel_data* is NULL or is the current parallel region is implementation defined.

Cross References

- **barrier** Construct, see Section 17.3.1
- Implicit Barriers, see Section 17.3.2
- OMPT **sync_region** Type, see Section 34.7.3
- **taskgroup** Construct, see Section 17.4
- **taskwait** Construct, see Section 17.5

34.7.6 reduction Callback

Name: **reduction** **Category:** subroutine	**Properties:** C/C++-only, common-type-callback, synchronizing-region, OMPT

Type Signature

sync_region

Semantics

A tool provides a **reduction** callback, which is a synchronizing-region callback, that the OpenMP implementation dispatches when it performs reductions.

Cross References

- Properties Common to All Reduction Clauses, see Section 7.6.6
- OMPT **sync_region** Type, see Section 34.7.3

34.7.7 OMPT mutex_acquire Type

Name: **mutex_acquire** **Category:** subroutine pointer	**Properties:** C/C++-only, OMPT

Arguments

Name	Type	Properties
kind	mutex	OMPT, overlapping-type-name
hint	integer	unsigned
impl	integer	unsigned
wait_id	wait_id	OMPT
codeptr_ra	void	intent(in), pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_mutex_acquire_t) (ompt_mutex_t kind,
  unsigned int hint, unsigned int impl, ompt_wait_id_t wait_id,
  const void *codeptr_ra);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_mutex_acquire_t {
  ompt_mutex_t kind;
  unsigned int hint;
  unsigned int impl;
  ompt_wait_id_t wait_id;
  const void *codeptr_ra;
} ompt_record_mutex_acquire_t;
```

C / C++

Semantics

Callbacks that have the **mutex_acquire** OMPT type are mutex-acquiring callbacks, which each have the mutex-acquiring property. A tool provides these callbacks to monitor the beginning of regions associated with mutual-exclusion constructs, lock-initializing routines and lock-acquiring routines. The *kind* argument, which has the **mutex** OMPT type, indicates the kind of mutual exclusion event. The *hint* argument indicates the hint that was provided when initializing an implementation of mutual exclusion. If no hint is available when a thread initiates acquisition of mutual exclusion, the runtime may supply **omp_sync_hint_none** as the value for *hint*. The *impl* argument indicates the mechanism chosen by the runtime to implement the mutual exclusion.

Cross References

- OMPT **mutex** Type, see Section 33.20
- OMPT **wait_id** Type, see Section 33.40

34.7.8 `mutex_acquire` Callback

Name: `mutex_acquire` **Category:** subroutine	**Properties:** C/C++-only, common-type-callback, mutex-acquiring, OMPT

Type Signature

`mutex_acquire`

Semantics

A tool provides a `mutex_acquire` callback, which has the `mutex_acquire` OMPT type, that the OpenMP implementation dispatches when regions associated with mutual-exclusion constructs, lock-acquiring routines and lock-testing routines are begun.

Cross References

- OMPT `mutex_acquire` Type, see Section 34.7.7

34.7.9 `lock_init` Callback

Name: `lock_init` **Category:** subroutine	**Properties:** C/C++-only, common-type-callback, mutex-acquiring, OMPT

Type Signature

`mutex_acquire`

Semantics

A tool provides a `lock_init` callback, which has the `mutex_acquire` OMPT type, that the OpenMP implementation dispatches when lock-initializing routines are executed.

Cross References

- OMPT `mutex_acquire` Type, see Section 34.7.7

34.7.10 OMPT `mutex` Type

Name: `mutex` **Category:** subroutine pointer	**Properties:** C/C++-only, OMPT, overlapping-type-name

Arguments

Name	Type	Properties
kind	mutex	OMPT, overlapping-type-name
wait_id	wait_id	OMPT
codeptr_ra	void	intent(in), pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_mutex_t) (ompt_mutex_t kind,
  ompt_wait_id_t wait_id, const void *codeptr_ra);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_mutex_t {
  ompt_mutex_t kind;
  ompt_wait_id_t wait_id;
  const void *codeptr_ra;
} ompt_record_mutex_t;
```

C / C++

Semantics

Callbacks that have the **mutex** OMPT type are mutex-execution callbacks, which each have the mutex-execution property. A tool provides these callbacks to monitor the execution of a lock-destroying routine or the beginning or completion of execution of either the structured block associated with a mutual-exclusion construct, or the region guarded by a lock-acquiring routine or lock-testing routine paired with a lock-releasing routine. The *kind* argument, which has the **mutex** OMPT type, indicates the kind of mutual exclusion event.

Cross References

- Lock Acquiring Routines, see Section 28.3
- Lock Destroying Routines, see Section 28.2
- Lock Releasing Routines, see Section 28.4
- Lock Testing Routines, see Section 28.5
- OMPT **mutex** Type, see Section 33.20
- OMPT **wait_id** Type, see Section 33.40

34.7.11 lock_destroy Callback

Name: lock_destroy Category: subroutine	Properties: C/C++-only, common-type-callback, mutex-execution, OMPT

Type Signature

mutex

Semantics

A tool provides a **lock_destroy** callback, which has the **mutex** OMPT type, that the OpenMP implementation dispatches when it executes a lock-destroying routine.

Cross References

- Lock Destroying Routines, see Section 28.2
- OMPT **mutex** Type, see Section 34.7.10

34.7.12 mutex_acquired Callback

Name: mutex_acquired Category: subroutine	Properties: C/C++-only, common-type-callback, mutex-execution, OMPT

Type Signature

mutex

Semantics

A tool provides a **mutex_acquired** callback, which has the **mutex** OMPT type, that the OpenMP implementation dispatches when the structured block associated with a mutual-exclusion construct begins execution or when a region guarded by a lock-acquiring routine or lock-testing routine begins execution.

Cross References

- Lock Acquiring Routines, see Section 28.3
- Lock Testing Routines, see Section 28.5
- OMPT **mutex** Type, see Section 34.7.10

34.7.13 mutex_released Callback

Name: mutex_released Category: subroutine	Properties: C/C++-only, common-type-callback, mutex-execution, OMPT

Type Signature

mutex

Semantics

A tool provides a **mutex_released** callback, which has the **mutex** OMPT type, that the OpenMP implementation dispatches when the structured block associated with a mutual-exclusion construct completes execution or, similarly, when a region that a lock-releasing routine guards completes execution.

Cross References

- Lock Releasing Routines, see Section 28.4
- OMPT **mutex** Type, see Section 34.7.10

34.7.14 `nest_lock` Callback

Name: `nest_lock` **Category:** subroutine	**Properties:** C/C++-only, OMPT

Arguments

Name	Type	Properties
endpoint	scope_endpoint	OMPT
wait_id	wait_id	OMPT
codeptr_ra	void	intent(in), pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_nest_lock_t) (
  ompt_scope_endpoint_t endpoint, ompt_wait_id_t wait_id,
  const void *codeptr_ra);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_nest_lock_t {
  ompt_scope_endpoint_t endpoint;
  ompt_wait_id_t wait_id;
  const void *codeptr_ra;
} ompt_record_nest_lock_t;
```

C / C++

Semantics

A tool provides a **`nest_lock`** callback, which has the **`nest_lock`** OMPT type, that the OpenMP implementation dispatches when a thread that owns a nestable lock invokes a routine that alters the nesting count of the lock but does not relinquish its ownership.

Cross References

- OMPT **`scope_endpoint`** Type, see Section 33.27
- OMPT **`wait_id`** Type, see Section 33.40

34.7.15 `flush` Callback

Name: `flush` **Category:** subroutine	**Properties:** C/C++-only, OMPT

Arguments

Name	Type	Properties
thread_data	data	OMPT, pointer, untraced-argument
codeptr_ra	void	intent(in), pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_flush_t) (ompt_data_t *thread_data,
  const void *codeptr_ra);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_flush_t {
  const void *codeptr_ra;
} ompt_record_flush_t;
```

C / C++

Semantics

A tool provides a **flush** callback, which has the **flush** OMPT type, that the OpenMP implementation dispatches when it encounters a **flush** construct. The binding of the *thread_data* argument is the encountering thread.

Cross References

- OMPT **data** Type, see Section 33.8
- **flush** Construct, see Section 17.8.6

34.8 control_tool Callback

Name: control_tool Category: function	Properties: C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
command	c_uint64_t	*default*
modifier	c_uint64_t	*default*
arg	c_ptr	iso_c, value, untraced-argument
codeptr_ra	void	intent(in), pointer

Type Signature

C / C++

```
typedef int (*ompt_callback_control_tool_t) (uint64_t command,
  uint64_t modifier, void *arg, const void *codeptr_ra);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_control_tool_t {
  uint64_t command;
  uint64_t modifier;
  const void *codeptr_ra;
} ompt_record_control_tool_t;
```

C / C++

Semantics

A tool provides a **control_tool** callback, which has the **control_tool** OMPT type, that the OpenMP implementation uses to dispatch *tool-control* events. This callback may return any non-negative value, which will be returned to the OpenMP program as the return value of the **omp_control_tool** call that triggered the callback.

The *command* argument passes a command from an OpenMP program to a tool. Standard values for *command* are defined by the **control_tool** OpenMP type. The *modifier* argument passes a command modifier from an OpenMP program to a tool. The *command* and *modifier* arguments may have tool-defined values. Tools must ignore *command* values that they are not designed to handle. The *arg* argument is a void pointer that enables a tool and an OpenMP program to exchange arbitrary state. The *arg* argument may be NULL.

Restrictions

Restrictions on **control_tool** callbacks are as follows:

- Tool-defined values for *command* must be greater than or equal to 64 and less than or equal to 2147483647 (**INT32_MAX**).
- Tool-defined values for *modifier* must be non-negative and less than or equal to 2147483647 (**INT32_MAX**).

Cross References

- OpenMP **control_tool** Type, see Section 20.12.1
- **omp_control_tool** Routine, see Section 31.1

35 Device Callbacks and Tracing

This chapter describes device-tracing callbacks, which have the device-tracing property. An OMPT tool may register these callbacks to monitor and to trace events that involve device execution. The C/C++ header file (**omp-tools.h**) also provides the types that this chapter defines.

35.1 device_initialize Callback

Name: **device_initialize** **Category:** subroutine	**Properties:** C/C++-only, device-tracing, OMPT

Arguments

Name	Type	Properties
device_num	integer	*default*
type	char	intent(in), pointer
device	device	OMPT, opaque, pointer
lookup	function_lookup	OMPT
documentation	char	intent(in), pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_device_initialize_t) (
  int device_num, const char *type, ompt_device_t *device,
  ompt_function_lookup_t lookup, const char *documentation);
```

C / C++

Semantics

A tool provides **device_initialize** callbacks, which have the **device_initialize** OMPT type, that the OpenMP implementation can use to initialize asynchronous collection of traces for devices. The OpenMP implementation dispatches this callback after OpenMP is initialized for the device but before execution of any construct is started on the device.

A **device_initialize** callback must fulfill several duties. First, the *type* argument should be used to determine if any special knowledge about the hardware or software of a device is employed. Second, the *lookup* argument should be used to look up pointers to device-tracing entry points for the device. Finally, these entry points should be used to set up tracing for the device. Initialization of tracing for a target device is described in Section 32.2.5.

The *device_num* argument indicates the device number of the device that is being initialized. The *type* argument is a C string that indicates the type of the device. A device type string is a semicolon-separated character string that includes, at a minimum, the vendor and model name of the device. These names may be followed by a semicolon-separated sequence of characteristics of the hardware or software of the device.

The *device* argument is a pointer to an OpenMP object that represents the target device instance. Device-tracing entry points use this pointer to identify the device that is being addressed. The *lookup* argument points to a **function_lookup** entry point that a tool must use to obtain pointers to other device-tracing entry points. If a device does not support tracing then *lookup* is NULL. The *documentation* argument is a C string that describes how to use these entry points. This documentation string may be a pointer to external documentation, or it may be inline descriptions that include names and type signatures for any device-specific entry points that are available through the **function_lookup** entry point along with descriptions of how to use them to control monitoring and analysis of device traces.

The *type* and *documentation* arguments are immutable strings that are defined for the lifetime of program execution.

Cross References

- OMPT **device** Type, see Section 33.11
- **function_lookup** Entry Point, see Section 36.1

35.2 device_finalize Callback

Name: **device_finalize** **Category:** subroutine	**Properties:** C/C++-only, device-tracing, OMPT

Arguments

Name	Type	Properties
device_num	integer	*default*

Type Signature

C / C++

```
typedef void (*ompt_callback_device_finalize_t) (int device_num);
```

C / C++

Semantics

A tool provides **device_finalize** callbacks, which have the **device_finalize** OMPT type, that the OpenMP implementation can use to finalize asynchronous collection of traces for devices. The OpenMP implementation dispatches this callback immediately prior to finalizing the device that the *device_num* argument identifies. Prior to dispatching a **device_finalize** callback for a device on which tracing is active, the OpenMP implementation stops tracing on the device and synchronously flushes all trace records for the device that have not yet been reported. These trace records are flushed through one or more **buffer_complete** callbacks as needed prior to the dispatch of the **device_finalize** callback.

Cross References

- **buffer_complete** Callback, see Section 35.6

35.3 device_load Callback

Name: **device_load** Category: subroutine	Properties: C/C++-only, device-tracing, OMPT

Arguments

Name	Type	Properties
device_num	integer	*default*
filename	char	intent(in), pointer
offset_in_file	c_int64_t	iso_c, value
vma_in_file	c_ptr	iso_c, value
bytes	c_size_t	iso_c, value
host_addr	c_ptr	iso_c, value
device_addr	c_ptr	iso_c, value
module_id	c_uint64_t	*default*

Type Signature

C / C++

```
typedef void (*ompt_callback_device_load_t) (int device_num,
  const char *filename, int64_t offset_in_file, void *vma_in_file,
  size_t bytes, void *host_addr, void *device_addr,
  uint64_t module_id);
```

C / C++

Semantics

A tool provides a **device_load** callback, which has the **device_load** OMPT type, that the OpenMP implementation can use to indicate that it has just loaded code onto the specified device. The *device_num* argument indicates the device number of the device that is being loaded. The *filename* argument indicates the name of a file in which the device code can be found. A NULL *filename* indicates that the code is not available in a file in the file system. The *offset_in_file* argument indicates an offset into *filename* at which the code can be found. A value of -1 indicates that no offset is provided. The *vma_in_file* argument indicates a virtual address in *filename* at which the code can be found. If no virtual address in the file is available then **ompt_addr_none** is used. The *bytes* argument indicates the size of the device code object in bytes.

The *host_addr* argument indicates the address at which a copy of the device code is available in host memory. The *device_addr* argument indicates the address at which the device code has been loaded in device memory. Both *host_addr* and *device_addr* will be **ompt_addr_none** when no code address is available for the relevant device. The *module_id* argument is an identifier that is associated with the device code object.

35.4 device_unload Callback

Name: device_unload **Category:** subroutine	**Properties:** C/C++-only, device-tracing, OMPT

Arguments

Name	Type	Properties
device_num	integer	*default*
module_id	c_uint64_t	*default*

Type Signature

C / C++

```
typedef void (*ompt_callback_device_unload_t) (int device_num,
  uint64_t module_id);
```

C / C++

Semantics

A tool provides a **device_unload** callback, which has the **device_unload** OMPT type, that the OpenMP implementation can use to indicate that it is about to unload code from the specified device. The *device_num* argument indicates the device number of the device that is being unloaded. The *module_id* argument is an identifier that is associated with the device code object.

35.5 buffer_request Callback

Name: buffer_request **Category:** subroutine	**Properties:** C/C++-only, device-tracing, OMPT

Arguments

Name	Type	Properties
device_num	integer	*default*
buffer	buffer	pointer-to-pointer
bytes	size_t	pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_buffer_request_t) (int device_num,
  ompt_buffer_t **buffer, size_t *bytes);
```

C / C++

Semantics

A tool provides a **buffer_request** callback, which has the **buffer_request** OMPT type, that the OpenMP implementation dispatches to request a buffer in which to store trace records for the device specified by the *device* argument. The callback sets the location to which the *buffer* argument points to point to the location of the provided buffer. On entry to the callback, the location to which the *bytes* argument points holds the minimum size of the buffer in bytes that the implementation requests; the implementation must ensure that this size does not exceed the recommended buffer size returned by the **get_buffer_limits** entry point for that device. A buffer request callback may set the location to which *bytes* points to 0 if it does not provide a buffer. If a callback sets that location to a value less than the minimum requested buffer size, further recording of events for the device may be disabled until the next invocation of the **start_trace** entry point. This action causes the implementation to drop any trace records for the device until recording is restarted.

Cross References

- OMPT **buffer** Type, see Section 33.3
- **get_buffer_limits** Entry Point, see Section 37.6

35.6 buffer_complete Callback

Name: **buffer_complete** Category: subroutine	Properties: C/C++-only, device-tracing, OMPT

Arguments

Name	Type	Properties
device_num	integer	*default*
buffer	buffer	pointer
bytes	size_t	*default*
begin	buffer_cursor	OMPT, opaque
buffer_owned	integer	*default*

Type Signature

C / C++

```
typedef void (*ompt_callback_buffer_complete_t) (int device_num,
  ompt_buffer_t *buffer, size_t bytes, ompt_buffer_cursor_t begin,
  int buffer_owned);
```

C / C++

Semantics

A tool provides a **buffer_complete** callback, which has the **buffer_complete** OMPT type, that the OpenMP implementation dispatches to indicate that it will not record any more trace records in the buffer at the location to which the *buffer* argument points. The implementation guarantees that all trace records in the buffer, which was previously allocated by a **buffer_request** callback, are valid. The *device* argument specifies the device for which the trace records were gathered. The *bytes* argument indicates the full size of the buffer. The *begin* argument is a OpenMP object that indicates the position of the beginning of the first trace record in the buffer. The *buffer_owned* argument is 1 if the data to which *buffer* points can be deleted by the callback and 0 otherwise. If multiple devices accumulate events into a single buffer, this callback may be invoked with a pointer to one or more trace records in a shared buffer with *buffer_owned* equal to zero.

Typically, a tool will iterate through the trace records in the buffer and process them. The OpenMP implementation makes these callbacks on a native thread that is not an OpenMP thread so these **buffer_complete** callbacks are not required to be async signal safe.

Restrictions

Restrictions on **buffer_complete** callbacks are as follows:

- The callback must not delete the buffer if *buffer_owned* is zero.

Cross References

- OMPT **buffer** Type, see Section 33.3
- OMPT **buffer_cursor** Type, see Section 33.4

35.7 target_data_op_emi Callback

Name: **target_data_op_emi** **Category:** subroutine	**Properties:** C/C++-only, device-tracing, OMPT

Arguments

Name	Type	Properties
endpoint	scope_endpoint	OMPT, untraced-argument
target_task_data	data	OMPT, pointer, untraced-argument
target_data	data	OMPT, pointer, untraced-argument
host_op_id	id	OMPT, pointer
optype	target_data_op	OMPT
dev1_addr	c_ptr	iso_c, value
dev1_device_num	integer	*default*
dev2_addr	c_ptr	iso_c, value
dev2_device_num	integer	*default*
bytes	size_t	*default*
codeptr_ra	void	intent(in), pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_target_data_op_emi_t) (
  ompt_scope_endpoint_t endpoint, ompt_data_t *target_task_data,
  ompt_data_t *target_data, ompt_id_t *host_op_id,
  ompt_target_data_op_t optype, void *dev1_addr,
  int dev1_device_num, void *dev2_addr, int dev2_device_num,
  size_t bytes, const void *codeptr_ra);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_target_data_op_emi_t {
  ompt_id_t host_op_id;
  ompt_target_data_op_t optype;
  void *dev1_addr;
  int dev1_device_num;
  void *dev2_addr;
  int dev2_device_num;
  size_t bytes;
  ompt_device_time_t end_time;
  const void *codeptr_ra;
} ompt_record_target_data_op_emi_t;
```

C / C++

Additional information

The **target_data_op** callback may also be used. This callback has identical arguments to the **target_data_op_emi** callback except that the *endpoint* and *target_task_data* arguments are omitted and the *target_data* argument is replaced by the *target_id* argument, which has the **id** OMPT type, and the *host_op_id* argument is not a pointer and is provided by the implementation. If this callback is registered, it is dispatched for the *target_data_op_end*, *target-data-allocation-end*, *target-data-free-begin*, *target-data-associate*, *target-global-data-op*, and *target-data-disassociate* events. This callback has been deprecated. In addition to the standard trace record OMPT type name, the **target_data_op** name may be used to specify a trace record OMPT type with identical fields. This OMPT type name has been deprecated.

Semantics

A tool provides a **target_data_op_emi** callback, which has the **target_data_op_emi** OMPT type, that the OpenMP implementation dispatches when a device memory is allocated or freed, as well as when data is copied to or from a device.

Note – An OpenMP implementation may aggregate variables and data operations upon them. For instance, an implementation may synthesize a composite to represent multiple scalar variables and then allocate, free, or copy this composite as a whole rather than performing data operations on each one individually. Thus, the implementation may not dispatch callbacks for separate data operations on each variable.

The binding of the *target_task_data* argument is the target task region. The binding of the *target_data* argument is the device region. The *host_op_id* argument points to a tool-controlled integer value that identifies a data operation for a target device. The *optype* argument indicates the kind of data operation.

TABLE 35.1: Association of dev1 and dev2 arguments for target data operations

Data op	dev1	dev2
allocate	host/none	device
transfer	*from* device	*to* device
delete	host/none	device
associate	host	device
disassociate	host	device
memset	none	device

The *dev1_addr* argument indicates the data address on the device given by Table 35.1 or NULL if the table indicates none for device memory routines that solely operate on device memory. For rectangular-memory-copying routines this argument points to a structure of **subvolume** OMPT type that describes a rectangular subvolume of a multi-dimensional array *src*, in the device data environment of device *dev1_device_num*. The address *src* of the array is referenced as *base* in the **subvolume** OMPT type. The *dev1_device_num* argument indicates the device number on the device given by Table 35.1. The *dev2_addr* argument indicates the data address on the device given by Table 35.1. For rectangular-memory-copying routines this argument points to a structure of **subvolume** OMPT type that describes a rectangular subvolume of a multi-dimensional array *dst*, in the device data environment of device *dev2_device_num*. The address *dst* of the array is referenced as *base* in the **subvolume** OMPT type. The *dev2_device_num* argument indicates the device number on the device given by Table 35.1. Whether in some operations *dev1_addr* or *dev2_addr* may point to an intermediate buffer is implementation defined. The *bytes* argument indicates the size of the data in bytes.

If **set_trace_ompt** has configured the implementation to trace data operations to device memory then the implementation will log a **target_data_op_emi** trace record in a trace. The fields in the record are as follows:

- The **host_op_id** field contains an identifier of a data operation for a target device; if the corresponding **target_data_op_emi** callback was dispatched, this identifier is the tool-controlled integer value to which the *host_op_id* argument of the callback points so that a tool may correlate the trace record with the callback, and otherwise the **host_op_id** field contains an implementation-controlled identifier;
- The **optype**, **dev1_addr**, **dev1_device_num**, **dev2_addr**, **dev2_device_num**, **bytes**, and **codeptr_ra** fields contain the same values as the callback;
- The time when the data operation began execution for the device is recorded in the **time** field of an enclosing trace record of **record_ompt** OMPT type; and
- The time when the data operation completed execution for the device is recorded in the **end_time** field.

Restrictions

Restrictions to **target_data_op_emi** callbacks are as follows:

- The deprecated **target_data_op** callback must not be registered if a **target_data_op_emi** callbacks is registered.

Cross References

- OMPT **data** Type, see Section 33.8
- OMPT **device_time** Type, see Section 33.12
- OMPT **id** Type, see Section 33.18
- **map** Clause, see Section 7.9.6
- OMPT **scope_endpoint** Type, see Section 33.27
- OMPT **target_data_op** Type, see Section 33.35

35.8 target_emi Callback

Name: **target_emi** Category: subroutine	**Properties:** C/C++-only, device-tracing, OMPT

Arguments

Name	Type	Properties
kind	target	OMPT
endpoint	scope_endpoint	OMPT
device_num	integer	*default*
task_data	data	OMPT, pointer
target_task_data	data	OMPT, pointer, untraced-argument
target_data	data	OMPT, pointer
codeptr_ra	void	intent(in), pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_target_emi_t) (ompt_target_t kind,
  ompt_scope_endpoint_t endpoint, int device_num,
  ompt_data_t *task_data, ompt_data_t *target_task_data,
  ompt_data_t *target_data, const void *codeptr_ra);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_target_emi_t {
  ompt_target_t kind;
  ompt_scope_endpoint_t endpoint;
  int device_num;
  ompt_id_t task_id;
  ompt_id_t target_id;
  const void *codeptr_ra;
} ompt_record_target_emi_t;
```

C / C++

Additional information

The **target** callback may also be used. This callback has identical arguments to the **target_emi** callback except that the *target_task_data* argument is omitted and the *target_data* argument is replaced by the *target_id* argument, which has the **id** OMPT type. If this callback is registered, it is dispatched for the *target-begin*, *target-end*, *target-enter-data-begin*, *target-enter-data-end*, *target-exit-data-begin*, *target-exit-data-end*, *target-update-begin*, and *target-update-end* events. This callback has been deprecated. In addition to the standard trace record OMPT type name, the **target** name may be used to specify a trace record OMPT type with identical fields. This OMPT type name has been deprecated.

Semantics

A tool provides a **target_emi** callback, which has the **target_emi** OMPT type, that the OpenMP implementation dispatches when a thread begins to execute a device construct. The *kind* argument indicates the kind of device region. The *device_num* argument specifies the device number of the target device associated with the region. The binding of the *task_data* argument is the encountering task. The binding of the *target_task_data* argument is the target task. If a device region does not have a target task or if the target task is a merged task, this argument is NULL. The binding of the *target_data* argument is the device region.

Restrictions

Restrictions to **target_emi** callbacks are as follows:

- The deprecated **target** callback must not be registered if a **target_emi** callback is registered.

Cross References

- OMPT **data** Type, see Section 33.8
- OMPT **id** Type, see Section 33.18
- OMPT **scope_endpoint** Type, see Section 33.27
- **target** Construct, see Section 15.8

- OMPT **target** Type, see Section 33.34
- **target_data** Construct, see Section 15.7
- **target_enter_data** Construct, see Section 15.5
- **target_exit_data** Construct, see Section 15.6
- **target_update** Construct, see Section 15.9

35.9 target_map_emi Callback

Name: **target_map_emi** **Category:** subroutine	**Properties:** C/C++-only, device-tracing, OMPT

Arguments

Name	Type	Properties
target_data	data	OMPT, pointer
nitems	integer	unsigned
host_addr	void	pointer-to-pointer
device_addr	void	pointer-to-pointer
bytes	size_t	pointer
mapping_flags	integer	unsigned, pointer
codeptr_ra	void	intent(in), pointer

Type Signature

C / C++

```
typedef void (*ompt_callback_target_map_emi_t) (
  ompt_data_t *target_data, unsigned int nitems, void **host_addr,
  void **device_addr, size_t *bytes, unsigned int *mapping_flags,
  const void *codeptr_ra);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_target_map_emi_t {
  ompt_id_t target_id;
  unsigned int nitems;
  void **host_addr;
  void **device_addr;
  size_t *bytes;
  unsigned int *mapping_flags;
  const void *codeptr_ra;
} ompt_record_target_map_emi_t;
```

C / C++

Additional information

The **target_map** callback may also be used. This callback has identical arguments to the **target_map_emi** callback except that the *target_data* argument is replaced by the *target_id* argument, which has the **id** OMPT type. If this callback is registered, it is dispatched for any *target-map* events. This callback has been deprecated. In addition to the standard trace record OMPT type name, the **target_map** name may be used to specify a trace record OMPT type with identical fields. This OMPT type name has been deprecated.

Semantics

A tool provides a **target_map_emi** callback, which has the **target_map_emi** OMPT type, that the OpenMP implementation dispatches to indicate data mapping relationships. The implementation may report mappings associated with multiple **map** clauses that appear on the same construct with a single callback to report the effect of all mappings or multiple callbacks with each reporting a subset of the mappings. Further, the implementation may omit mappings that it determines are unnecessary. If the implementation issues multiple **target_map_emi** callbacks, these callbacks may be interleaved with **target_data_op_emi** callbacks that report data operations associated with the mappings.

The binding of the *target_data* argument is the device region. The *nitems* argument indicates the number of data mappings that the callback reports. The *host_addr* argument indicates an array of host addresses. The *device_addr* argument indicates an array of device addresses. The *bytes* argument indicates an array of sizes of data. The *mapping_flags* argument indicates the kind of mapping operations, which may result from explicit **map** clauses or the implicit data-mapping rules (see Section 7.9). Flags for the mapping operations include one or more values specified by the **target_map_flag** type.

Restrictions

Restrictions to **target_map_emi** callbacks are as follows:

- The deprecated **target_map** callback must not be registered if a **target_map_emi** callback is registered.

Cross References

- OMPT **data** Type, see Section 33.8
- OMPT **id** Type, see Section 33.18
- **map** Clause, see Section 7.9.6
- **target_data_op_emi** Callback, see Section 35.7
- OMPT **target_map_flag** Type, see Section 33.36

35.10 `target_submit_emi` Callback

Name: `target_submit_emi` **Category:** subroutine	**Properties:** C/C++-only, device-tracing, OMPT

Arguments

Name	Type	Properties
endpoint	scope_endpoint	OMPT, untraced-argument
target_data	data	OMPT, pointer, untraced-argument
host_op_id	id	OMPT, pointer
requested_num_teams	integer	unsigned

Type Signature

C / C++

```
typedef void (*ompt_callback_target_submit_emi_t) (
  ompt_scope_endpoint_t endpoint, ompt_data_t *target_data,
  ompt_id_t *host_op_id, unsigned int requested_num_teams);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_target_submit_emi_t {
  ompt_id_t host_op_id;
  unsigned int requested_num_teams;
  unsigned int granted_num_teams;
  ompt_device_time_t end_time;
} ompt_record_target_submit_emi_t;
```

C / C++

Additional information

The **`target_submit`** callback may also be used. This callback has identical arguments to the **`target_submit_emi`** callback except that the *endpoint* argument is omitted and the *target_data* argument is replaced by the *target_id* argument, which has the **`id`** OMPT type, and the *host_op_id* argument is not a pointer and is provided by the implementation. If this callback is registered, it is dispatched for any *target-submit-begin* events. This callback has been deprecated. In addition to the standard trace record OMPT type name, the **`target_kernel`** name may be used to specify a trace record OMPT type with identical fields. This OMPT type name has been deprecated.

Semantics

A tool provides a **target_submit_emi** callback, which has the **target_submit_emi** OMPT type, that the OpenMP implementation dispatches before and after a target task initiates creation of an initial task on a device. The binding of the *target_data* argument is the device region. The *host_op_id* argument points to a tool-controlled integer value that identifies an initial task on a target device. The *requested_num_teams* argument is the number of teams that the device construct requested to execute the region. The actual number of teams that execute the region may be smaller and generally will not be known until the region begins to execute on the device.

If **set_trace_ompt** has configured the implementation to trace device region execution for a device then the implementation will log a **target_submit_emi** trace record. The fields in the record are as follows:

- The **host_op_id** field contains an identifier that identifies the initial task on the device; if the corresponding **target_submit_emi** callback was dispatched, this identifier is the tool-controlled integer value to which the *host_op_id* argument of the callback points so that a tool may correlate the trace record with the callback, and otherwise the **host_op_id** field contains an implementation-controlled identifier;
- The **requested_num_teams** field contains the number of teams that the device construct requested to execute the device region;
- The **granted_num_teams** field contains the number of teams that the device actually used to execute the device region;
- The time when the initial task began execution on the device is recorded in the **time** field of an enclosing trace record of **record_ompt** OMPT type; and
- The time when the initial task completed execution on the device is recorded in the **end_time** field.

Restrictions

Restrictions to **target_submit_emi** callbacks are as follows:

- The deprecated **target_submit** callback must not be registered if a **target_submit_emi** callback is registered.

Cross References

- OMPT **data** Type, see Section 33.8
- OMPT **device_time** Type, see Section 33.12
- OMPT **id** Type, see Section 33.18
- OMPT **scope_endpoint** Type, see Section 33.27
- **target** Construct, see Section 15.8

36 General Entry Points

OMPT supports two principal sets of runtime entry points for tools. For both sets, entry points should not be global symbols since tools cannot rely on the visibility of such symbols. This chapter defines the first set, which enables a tool to register callbacks for events and to inspect the state of threads while executing in a callback or a signal handler. The **omp-tools.h** C/C++ header file provides the definitions of the types that are specified throughout this chapter.

OMPT also supports entry points for two classes of lookup entry points. The first class of lookup entry points contains a single member that is provided through the **initialize** callback: a **function_lookup** entry point that returns pointers to the set of entry points that are defined in this chapter. The second class of lookup entry points includes a unique lookup entry point for each kind of device that can return pointers to entry points in a device's OMPT tracing interface.

The binding thread set for each OMPT entry point is the encountering thread unless otherwise specified. The binding task set is the task executing on the encountering thread.

Several entry points are async-signal-safe entry points, which means they each have the async-signal-safe property, which implies that they are async signal safe.

Restrictions

Restrictions on OMPT runtime entry points are as follows:

- Entry points must not be called from a signal handler on a native thread before a *native-thread-begin* or after a *native-thread-end* event.
- Device entry points must not be called after a *device-finalize* event for that device.

36.1 function_lookup Entry Point

Name: **function_lookup** **Category:** function	**Properties:** C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	interface_fn	*default*
interface_function_name	char	intent(in), pointer

Type Signature

C / C++

```
typedef ompt_interface_fn_t (*ompt_function_lookup_t) (
  const char *interface_function_name);
```

C / C++

Semantics

The **function_lookup** entry point, which has the **function_lookup** OMPT type, enables tools to look up pointers to OMPT entry points by name. When an OpenMP implementation invokes the **initialize** callback to configure the OMPT callback interface, it provides an entry point that provides pointers to other entry points that implement routines that are part of the OMPT callback interface. Alternatively, when it invokes a **device_initialize** callback to configure the OMPT tracing interface for a device, it provides an entry point that provides pointers to entry points that implement tracing control routines appropriate for that device.

For these entry points, the *interface_function_name* argument is a C string that represents the name of the entry point to look up. If the name is unknown to the implementation, the entry point returns NULL. In a compliant implementation, the entry point that is provided by the **initialize** callback returns a valid function pointer for any entry point name listed in Table 32.1. Similarly, in a compliant implementation, the entry point that is provided by the **device_initialize** callback returns non-NULL function pointers for any entry point name listed in Table 32.3, except for **set_trace_ompt** and **get_record_ompt**, as described in Section 32.2.5.

Cross References

- **device_initialize** Callback, see Section 35.1
- Binding Entry Points, see Section 32.2.3.1
- Tracing Activity on Target Devices, see Section 32.2.5
- **initialize** Callback, see Section 34.1.1
- OMPT **interface_fn** Type, see Section 33.19

36.2 enumerate_states Entry Point

Name: enumerate_states **Category:** function	**Properties:** C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
current_state	integer	*default*
next_state	integer	pointer
next_state_name	const char	intent(out), pointer-to-pointer

Type Signature

C / C++

```
typedef int (*ompt_enumerate_states_t) (int current_state,
  int *next_state, const char **next_state_name);
```

C / C++

Semantics

An OpenMP implementation may support only a subset of the thread states that the **state** OMPT type defines. An OpenMP implementation may also support implementation defined states. The **enumerate_states** entry point, which has the **enumerate_states** OMPT type, is the entry point that enables a tool to enumerate the supported thread states.

When a supported thread state is passed as *current_state*, the entry point assigns the next thread state in the enumeration to the variable passed by reference in *next_state* and assigns the name associated with that state to the character pointer passed by reference in *next_state_name*; the returned string is immutable and defined for the lifetime of program execution. Whenever one or more states are left in the enumeration, the **enumerate_states** entry point returns 1. When the last state in the enumeration is passed as *current_state*, **enumerate_states** returns 0, which indicates that the enumeration is complete.

To begin enumerating the supported states, a tool should pass **ompt_state_undefined** as *current_state*. Subsequent invocations of **enumerate_states** should pass the value assigned to the variable that was passed by reference in *next_state* to the previous call. The **ompt_state_undefined** value is returned to indicate an invalid thread state.

Cross References

- OMPT **state** Type, see Section 33.31

36.3 enumerate_mutex_impls Entry Point

Name: **enumerate_mutex_impls** Category: function	**Properties:** C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
current_impl	integer	*default*
next_impl	integer	pointer
next_impl_name	const char	intent(out), pointer-to-pointer

Type Signature

C / C++

```
typedef int (*ompt_enumerate_mutex_impls_t) (int current_impl,
  int *next_impl, const char **next_impl_name);
```

C / C++

Semantics

Mutual exclusion for locks, **critical** regions, and **atomic** regions may be implemented in several ways. The **enumerate_mutex_impls** entry point, which has the **enumerate_mutex_impls** OMPT type, enables a tool to enumerate the supported mutual exclusion implementations.

When a supported mutex implementation is passed as *current_impl*, the entry point assigns the next mutex implementation in the enumeration to the variable passed by reference in *next_impl* and assigns the name associated with that mutex implementation to the character pointer passed by reference in *next_impl_name*; the returned string is immutable and defined for the lifetime of program execution. Whenever one or more mutex implementations are left in the enumeration, the **enumerate_mutex_impls** entry point returns 1. When the last mutex implementation in the enumeration is passed as *current_impl*, the entry point returns 0, which indicates that the enumeration is complete.

To begin enumerating the supported mutex implementations, a tool should pass **ompt_mutex_impl_none** as *current_impl*. Subsequent invocations of **enumerate_mutex_impls** should pass the value assigned to the variable that was passed by reference in *next_impl* to the previous call. The value **ompt_mutex_impl_none** is returned to indicate an invalid mutex implementation.

36.4 set_callback Entry Point

Name: set_callback **Category:** function	**Properties:** C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	set_result	*default*
event	callbacks	OMPT
callback	callback	OMPT

Type Signature

C / C++

```
typedef ompt_set_result_t (*ompt_set_callback_t) (
  ompt_callbacks_t event, ompt_callback_t callback);
```

C / C++

Semantics

OpenMP implementations can use callbacks to indicate the occurrence of events during the execution of an OpenMP program. The **set_callback** entry point, which has the **set_callback** OMPT type, enables a tool to register the callback indicated by the *callback* argument for the event indicated by the *event* argument on the current device. The return value of **set_callback** indicates the outcome of registering the callback and may be any value in the **set_result** OMPT type except **ompt_set_impossible**. If *callback* is NULL then callbacks associated with *event* are disabled. If callbacks are successfully disabled then **ompt_set_always** is returned.

Restrictions

Restrictions on the **set_callback** entry point are as follows:

- The type signature for *callback* must match the type signature appropriate for the event.

Cross References

- OMPT **callback** Type, see Section 33.5
- OMPT **callbacks** Type, see Section 33.6
- Monitoring Activity on the Host with OMPT, see Section 32.2.4
- OMPT **set_result** Type, see Section 33.28

36.5 get_callback Entry Point

Name: get_callback **Category:** function	**Properties:** C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
event	callbacks	OMPT
callback	callback	OMPT, pointer

Type Signature

C / C++

```
typedef int (*ompt_get_callback_t) (ompt_callbacks_t event,
  ompt_callback_t *callback);
```

C / C++

Semantics

The **get_callback** entry point, which has the **get_callback** OMPT type, enables a tool to retrieve a pointer to a registered callback (if any) that an OpenMP implementation invokes when a host event occurs. If the callback that is registered for the event that is specified by the *event* argument is not NULL, the pointer to the callback is assigned to the variable passed by reference in *callback* and **get_callback** returns 1; otherwise, it returns 0. If **get_callback** returns 0, the value of the variable passed by reference as *callback* is undefined.

Restrictions

Restrictions on the **get_callback** entry point are as follows:

- The *callback* argument must not be NULL and must point to valid storage.

Cross References

- OMPT **callback** Type, see Section 33.5
- OMPT **callbacks** Type, see Section 33.6
- **set_callback** Entry Point, see Section 36.4

36.6 `get_thread_data` Entry Point

Name: `get_thread_data` **Category:** function	**Properties:** async-signal-safe, C/C++-only, OMPT

Return Type

Name	Type	Properties
<return type>	data	pointer

Type Signature

C / C++

```
typedef ompt_data_t *(*ompt_get_thread_data_t) (void);
```

C / C++

Semantics

Each thread can have an associated thread data object of **`data`** OMPT type. The **`get_thread_data`** entry point, which has the **`get_thread_data`** OMPT type, enables a tool to retrieve a pointer to the thread data object, if any, that is associated with the encountering thread. A tool may use a pointer to a thread's data object that **`get_thread_data`** retrieves to inspect or to modify the value of the data object. When a thread is created, its data object is initialized with the value **`ompt_data_none`**.

Cross References

- OMPT **`data`** Type, see Section 33.8

36.7 `get_num_procs` Entry Point

Name: `get_num_procs` **Category:** function	**Properties:** all-device-threads-binding, async-signal-safe, C/C++-only, OMPT

Return Type

Name	Type	Properties
<return type>	integer	*default*

Type Signature

C / C++

```
typedef int (*ompt_get_num_procs_t) (void);
```

C / C++

Semantics

The **`get_num_procs`** entry point, which has the **`get_num_procs`** OMPT type, enables a tool to retrieve the number of processors that are available on the host device at the time the entry point is called. This value may change between the time that it is determined and the time that it is read in the calling context due to system actions outside the control of the OpenMP implementation. The binding thread set of this entry point is all threads on the host device.

36.8 `get_num_places` Entry Point

Name: `get_num_places` **Category:** function	**Properties:** all-device-threads-binding, async-signal-safe, C/C++-only, OMPT

Return Type

Name	Type	Properties
<return type>	integer	*default*

Type Signature

C / C++

```
typedef int (*ompt_get_num_places_t) (void);
```

C / C++

Semantics

The **`get_num_places`** entry point, which has the **`get_num_places`** OMPT type, enables a tool to retrieve the number of places in the place list. This value is equal to the number of places in the *place-partition-var* ICV in the execution environment of the initial task. The binding thread set of this entry point is all threads on the host device.

Cross References

- **`OMP_PLACES`**, see Section 4.1.6
- *place-partition-var* ICV, see Table 3.1

36.9 `get_place_proc_ids` Entry Point

Name: `get_place_proc_ids` **Category:** function	**Properties:** all-device-threads-binding, C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
place_num	integer	*default*
ids_size	integer	*default*
ids	integer	pointer

Type Signature

C / C++

```
typedef int (*ompt_get_place_proc_ids_t) (int place_num,
  int ids_size, int *ids);
```

C / C++

Semantics

The **`get_place_proc_ids`** entry point, which has the **`get_place_proc_ids`** OMPT type, enables a tool to retrieve the numerical identifiers of each processor that is associated with the place specified by the *place_num* argument. The *ids* argument is an array in which the entry point can return a vector of processor identifiers in the specified place; these identifiers are non-negative, and their meaning is implementation defined. The *ids_size* argument indicates the size of the result array that is specified by *ids*. The binding thread set of this entry point is all threads on the device.

If the *ids* array of size *ids_size* is large enough to contain all identifiers then they are returned in *ids* and their order in the array is implementation defined. Otherwise, if the *ids* array is too small, the values in *ids* when the entry point returns are undefined. The entry point always returns the number of numerical identifiers of the processors that are available to the execution environment in the specified place.

36.10 `get_place_num` Entry Point

Name: `get_place_num` **Category:** function	**Properties:** async-signal-safe, C/C++-only, OMPT

Return Type

Name	Type	Properties
<return type>	integer	*default*

Type Signature

C / C++

```
typedef int (*ompt_get_place_num_t) (void);
```

C / C++

Semantics

When the encountering thread is bound to a place, the **`get_place_num`** entry point, which has the **`get_place_num`** OMPT type, enables a tool to retrieve the place number associated with the thread. The returned value is between zero and one less than the value returned by **`get_num_places`**, inclusive. When the encountering thread is not bound to a place, the entry point returns -1.

36.11 `get_partition_place_nums` Entry Point

Name: `get_partition_place_nums` **Category:** function	**Properties:** async-signal-safe, C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
place_nums_size	integer	*default*
place_nums	integer	pointer

Type Signature

C / C++

```
typedef int (*ompt_get_partition_place_nums_t) (
  int place_nums_size, int *place_nums);
```

C / C++

Semantics

The **get_partition_place_nums** entry point, which has the **get_partition_place_nums** OMPT type, enables a tool to retrieve a list of place numbers that correspond to the places in the *place-partition-var* ICV of the innermost implicit task. The *place_nums* argument is an array in which the entry point can return a vector of place identifiers. The *place_nums_size* argument indicates the size of that array.

If the *place_nums* array of size *place_nums_size* is large enough to contain all identifiers then they are returned in *place_nums* and their order in the array is implementation defined. Otherwise, if the *place_nums* array is too small, the values in *place_nums* when the entry point returns are undefined. The entry point always returns the number of places in the *place-partition-var* ICV of the innermost implicit task.

Cross References

- **OMP_PLACES**, see Section 4.1.6
- *place-partition-var* ICV, see Table 3.1

36.12 get_proc_id Entry Point

Name: get_proc_id **Category:** function	**Properties:** async-signal-safe, C/C++-only, OMPT

Return Type

Name	Type	Properties
<*return type*>	integer	*default*

Type Signature

C / C++

```
typedef int (*ompt_get_proc_id_t) (void);
```

C / C++

The **get_proc_id** entry point, which has the **get_proc_id** OMPT type, enables a tool to retrieve the numerical identifier of the processor of the encountering thread. A defined numerical identifier is non-negative, and its meaning is implementation defined. A negative number indicates a failure to retrieve the numerical identifier.

36.13 get_state Entry Point

Name: get_state **Category:** function	**Properties:** async-signal-safe, C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
wait_id	wait_id	OMPT, pointer

Type Signature

C / C++

```
typedef int (*ompt_get_state_t) (ompt_wait_id_t *wait_id);
```

C / C++

Semantics

Each thread has an associated state and a wait identifier. If the thread state indicates that the thread is waiting for mutual exclusion then its wait identifier contains a handle that indicates the data object upon which the thread is waiting. The **get_state** entry point, which has the **get_state** OMPT type, enables a tool to retrieve the state and the wait identifier of the encountering thread. The returned value may be any one of the states predefined by the **state** OMPT type or a value that represents an implementation defined state. The tool may obtain a string representation for each state with the **enumerate_states** entry point. If the returned state indicates that the thread is waiting for a lock, nestable lock, **critical** region, **atomic** region, or **ordered** region and the wait identifier passed as the *wait_id* argument is not NULL then the value of the wait identifier is assigned to that argument, which is a pointer to a handle. If the returned state is not one of the specified wait states then the value of that handle is undefined after the call.

Restrictions

Restrictions on the **get_state** entry point are as follows:

- The *wait_id* argument must be a reference to a variable of the **wait_id** OMPT type or NULL.

Cross References

- **enumerate_states** Entry Point, see Section 36.2
- OMPT **state** Type, see Section 33.31
- OMPT **wait_id** Type, see Section 33.40

36.14 get_parallel_info Entry Point

Name: get_parallel_info **Category:** function	**Properties:** async-signal-safe, C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
ancestor_level	integer	*default*
parallel_data	data	OMPT, pointer-to-pointer
team_size	integer	pointer

Type Signature

C / C++

```
typedef int (*ompt_get_parallel_info_t) (int ancestor_level,
  ompt_data_t **parallel_data, int *team_size);
```

C / C++

Semantics

During execution, an OpenMP program may employ nested parallel regions. The **get_parallel_info** entry point, which has the **get_parallel_info** OMPT type, enables a tool to retrieve information about the current parallel region and any enclosing parallel regions for the current execution context.

The *ancestor_level* argument specifies the parallel region of interest by its ancestor level. Ancestor level 0 refers to the innermost parallel region; information about enclosing parallel regions may be obtained using larger values for *ancestor_level*. Information about a parallel region may not be available if the ancestor level is 0; otherwise it must be available if a parallel region exists at the specified ancestor level. The entry point returns 2 if a parallel region exists at the specified ancestor level and the information is available, 1 if a parallel region exists at the specified ancestor level but the information is currently unavailable, and 0 otherwise. The *parallel_data* argument returns the parallel data if the argument is not NULL. The *team_size* argument returns the team size if the argument is not NULL. If no parallel region exists at the specified ancestor level or the information is unavailable then the values of variables passed by reference to the entry point are undefined when **get_parallel_info** returns.

A tool may use the pointer to the data object of a parallel region that it obtains from this entry point to inspect or to modify the value of the data object. When a parallel region is created, its data object will be initialized with the value **ompt_data_none**. Between a *parallel-begin* event and an *implicit-task-begin* event, a call to **get_parallel_info** with an *ancestor_level* value of 0 may return information about the outer team or the new team. If a thread is in the **ompt_state_wait_barrier_implicit_parallel** state then a call to **get_parallel_info** may return a pointer to a copy of the specified parallel region's *parallel_data* rather than a pointer to the data word for the region itself. This convention enables the primary thread for a parallel region to free storage for the region immediately after the region ends, yet avoid having some other thread in the team that is executing the region potentially reference the *parallel_data* object for the region after it has been freed.

If **get_parallel_info** returns two then the entry point has the following effects:

- If a non-null value was passed for *parallel_data*, the value returned in *parallel_data* is a pointer to a data word that is associated with the parallel region at the specified level; and
- If a non-null value was passed for *team_size*, the value returned in the integer to which *team_size* points is the number of threads in the team that is associated with the parallel region.

Restrictions

Restrictions on the **get_parallel_info** entry point are as follows:

- While the *ancestor_level* argument is passed by value, all other arguments must be valid pointers to variables of the specified types or NULL.

Cross References

- OMPT **data** Type, see Section 33.8
- OMPT **state** Type, see Section 33.31

36.15 get_task_info Entry Point

Name: get_task_info **Category:** function	**Properties:** async-signal-safe, C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
ancestor_level	integer	*default*
flags	integer	pointer
task_data	data	OMPT, pointer-to-pointer
task_frame	frame	OMPT, pointer-to-pointer
parallel_data	data	OMPT, pointer-to-pointer
thread_num	integer	pointer

Type Signature

C / C++

```
typedef int (*ompt_get_task_info_t) (int ancestor_level,
  int *flags, ompt_data_t **task_data, ompt_frame_t **task_frame,
  ompt_data_t **parallel_data, int *thread_num);
```

C / C++

Semantics

During execution, a thread may be executing a task. Additionally, the stack of the thread may contain procedure frames that are associated with suspended tasks or routines. The **get_task_info** entry point, which has the **get_task_info** OMPT type, enables a tool to retrieve information about any task on the stack of the encountering thread.

The *ancestor_level* argument specifies the task region of interest by its ancestor level. Ancestor level 0 refers to the encountering task; information about other tasks with associated frames present on the stack in the current execution context may be queried at higher ancestor levels. Information about a task region may not be available if the ancestor level is 0; otherwise it must be available if a task region exists at the specified ancestor level. The entry point returns 2 if a task region exists at the specified ancestor level and the information is available, 1 if a task region exists at the specified ancestor level but the information is currently unavailable, and 0 otherwise.

If a task exists at the specified ancestor level and the information is available then information is returned in the variables passed by reference to the entry point. The *flags* argument returns the task type if the argument is not NULL. The *task_data* argument returns the task data if the argument is not NULL. The *task_frame* argument returns the task frame pointer if the argument is not NULL. The *parallel_data* argument returns the parallel data if the argument is not NULL. The *thread_num* argument returns the thread number if the argument is not NULL. If no task region exists at the specified ancestor level or the information is unavailable then the values of variables passed by reference to the entry point are undefined when **get_task_info** returns.

A tool may use a pointer to a data object for a task or parallel region that it obtains from **get_task_info** to inspect or to modify the value of the data object. When either a parallel region or a task region is created, its data object will be initialized with the value **ompt_data_none**.

If **get_task_info** returns 2 then the entry point has the following effects:

- If a non-null value was passed for *flags* then the value returned in the integer to which *flags* points represents the type of the task at the specified level; possible task types include initial task, implicit task, explicit task, and target task;
- If a non-null value was passed for *task_data* then the value that is returned in the object to which it points is a pointer to a data word that is associated with the task at the specified level;
- If a non-null value was passed for *task_frame* then the value that is returned in the object to which *task_frame* points is a pointer to the **frame** OMPT type structure that is associated with the task at the specified level;
- If a non-null value was passed for *parallel_data* then the value that is returned in the object to which *parallel_data* points is a pointer to a data word that is associated with the parallel region that contains the task at the specified level or, if the task at the specified level is an initial task, NULL; and
- If a non-null value was passed for *thread_num*, then the value that is returned in the object to which *thread_num* points indicates the number of the thread in the parallel region that is executing the task at the specified level.

Restrictions

Restrictions on the **get_task_info** entry point are as follows:

- While the *ancestor_level* argument is passed by value, all other arguments must be valid pointers to variables of the specified types or NULL.

Cross References

- OMPT **data** Type, see Section 33.8
- OMPT **frame** Type, see Section 33.15
- OMPT **task_flag** Type, see Section 33.37

36.16 get_task_memory Entry Point

Name: **get_task_memory** **Category:** function	**Properties:** async-signal-safe, C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
addr	void	pointer-to-pointer
size	size_t	pointer
block	integer	*default*

Type Signature

C / C++

```
typedef int (*ompt_get_task_memory_t) (void **addr, size_t *size,
  int block);
```

C / C++

Semantics

During execution, a thread may be executing a task. The OpenMP implementation must preserve the data environment from the generation of the task for its execution. The **get_task_memory** entry point, which has the **get_task_memory** OMPT type, enables a tool to retrieve information about memory ranges that store the data environment for the encountering task. Multiple memory ranges may be used to store these data. The *addr* argument is a pointer to a void pointer return value to provide the start address of a memory range. The *size* argument is a pointer to a size type return value to provide the size of the memory range. The *block* argument, which is an integer value to specify the memory block of interest, supports iteration over the memory ranges. The **get_task_memory** entry point returns one if more memory ranges are available, and zero otherwise. If no memory is used for a task, *size* is set to zero. In this case, the value to which *addr* points is undefined.

36.17 `get_target_info` Entry Point

Name: **`get_target_info`** **Category:** function	**Properties:** async-signal-safe, C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
device_num	c_uint64_t	pointer
target_id	id	OMPT, pointer
host_op_id	id	OMPT, pointer-to-pointer

Type Signature

C / C++

```
typedef int (*ompt_get_target_info_t) (uint64_t *device_num,
  ompt_id_t *target_id, ompt_id_t **host_op_id);
```

C / C++

Semantics

The **`get_target_info`** entry point, which has the **`get_target_info`** OMPT type, enables a tool to retrieve identifiers that specify the current **`target`** region and target operation ID of the encountering thread, if any. This entry point returns one if the encountering thread is in a **`target`** region and zero otherwise. If the entry point returns zero then the values of the variables passed by reference as its arguments are undefined. If the encountering thread is in a **`target`** region then **`get_target_info`** returns information about the current device, active **`target`** region, and active host operation, if any. In this case, the *device_num* argument returns the device number of the **`target`** region and the *target_id* argument returns the **`target`** region identifier. If the encountering thread is in the process of initiating an operation on a target device (for example, copying data to or from a device) then *host_op_id* returns the identifier for the operation; otherwise, *host_op_id* returns **`ompt_id_none`**.

Restrictions

Restrictions on the **`get_target_info`** entry point are as follows:

- All arguments must be valid pointers to variables of the specified types.

Cross References

- OMPT **`id`** Type, see Section 33.18

36.18 `get_num_devices` Entry Point

Name: **`get_num_devices`** **Category:** function	**Properties:** async-signal-safe, C/C++-only, OMPT

Return Type

Name	Type	Properties
<return type>	integer	*default*

Type Signature

C / C++

```
typedef int (*ompt_get_num_devices_t) (void);
```

C / C++

Semantics

The **get_num_devices** entry point, which has the **get_num_devices** OMPT type, is the entry point that enables a tool to retrieve the number of devices available to an OpenMP program.

36.19 get_unique_id Entry Point

Name: `get_unique_id` **Category:** function	**Properties:** async-signal-safe, C/C++-only, OMPT

Return Type

Name	Type	Properties
<return type>	c_uint64_t	*default*

Type Signature

C / C++

```
typedef uint64_t (*ompt_get_unique_id_t) (void);
```

C / C++

Semantics

The **get_unique_id** entry point, which has the **get_unique_id** OMPT type, enables a tool to retrieve a number that is unique for the duration of an OpenMP program. Successive invocations may not result in consecutive or even increasing numbers.

36.20 finalize_tool Entry Point

Name: `finalize_tool` **Category:** subroutine	**Properties:** C/C++-only, OMPT

Type Signature

C / C++

```
typedef void (*ompt_finalize_tool_t) (void);
```

C / C++

Semantics

A tool may detect that the execution of an OpenMP program is ending before the OpenMP implementation does. To facilitate clean termination of the tool, the tool may invoke the **finalize_tool** entry point, which has the **finalize_tool** OMPT type. Upon completion of **finalize_tool**, no OMPT callbacks are dispatched. The entry point detaches the tool from the runtime, unregisters all callbacks and invalidates all OMPT entry points passed to the tool by **function_lookup**. Upon completion of **finalize_tool**, no further callbacks will be issued on any thread. Before the callbacks are unregistered, the OpenMP runtime will dispatch all callbacks as if the program were exiting.

Restrictions

Restrictions on the **finalize_tool** entry point are as follows:

- The entry point must not be called from inside an explicit region.
- As **finalize_tool** should only be called when a tool detects that the execution of an OpenMP program is ending, a thread encountering an explicit region after the entry point has completed results in unspecified behavior.

37 Device Tracing Entry Points

The second set of OMPT entry points enables a tool to trace activities on a device. When directed by the tracing interface, an OpenMP implementation will trace activities on a device, collect buffers of trace records, and invoke callbacks on the host device to process these trace records. This chapter defines that set of entry points.

Several OMPT entry points have a *device* argument. This argument is a pointer to an OpenMP object that represents the target device. Callbacks in the device tracing interface use a pointer to this device object to identify the device being addressed.

37.1 `get_device_num_procs` Entry Point

Name: `get_device_num_procs` **Category:** function	**Properties:** C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
device	device	OMPT, pointer

Type Signature

C / C++

```
typedef int (*ompt_get_device_num_procs_t) (
  ompt_device_t *device);
```

C / C++

Semantics

The **`get_device_num_procs`** entry point, which has the **`get_device_num_procs`** OMPT type, enables a tool to retrieve the number of processors that are available on the device at the time the entry point is called. This value may change between the time that it is determined and the time that it is read in the calling context due to system actions outside the control of the OpenMP implementation.

Cross References

- OMPT **`device`** Type, see Section 33.11

37.2 get_device_time Entry Point

Name: **get_device_time** Category: function	Properties: C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	device_time	*default*
device	device	OMPT, pointer

Type Signature

C / C++

```
typedef ompt_device_time_t (*ompt_get_device_time_t) (
  ompt_device_t *device);
```

C / C++

Semantics

Host devices and target devices are typically distinct and run independently. If the host device and any target devices are different hardware components, they may use different clock generators. For this reason, a common time base for ordering host-side and device-side events may not be available. The **get_device_time** entry point, which has the **get_device_time** OMPT type, enables a tool to retrieve the current time on the device specified by the *device* argument. A tool can use the information retrieved by **get_device_time** to align time stamps from different devices.

Cross References

- OMPT **device** Type, see Section 33.11
- OMPT **device_time** Type, see Section 33.12

37.3 translate_time Entry Point

Name: **translate_time** Category: function	Properties: C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	double	*default*
device	device	OMPT, pointer
time	device_time	OMPT

Type Signature

C / C++

```
typedef double (*ompt_translate_time_t) (ompt_device_t *device,
  ompt_device_time_t time);
```

C / C++

Semantics

The **translate_time** entry point, which has the **translate_time** OMPT type, enables a tool to translate a time value, specified by the *time* argument, obtained from the device specified by the *device* argument to a corresponding time value on the host device. The returned value for the host time has the same meaning as the value returned from **omp_get_wtime**.

Cross References

- OMPT **device** Type, see Section 33.11
- OMPT **device_time** Type, see Section 33.12
- **omp_get_wtime** Routine, see Section 30.3.1

37.4 set_trace_ompt Entry Point

Name: set_trace_ompt **Category:** function	**Properties:** C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	set_result	*default*
device	device	OMPT, pointer
enable	integer	OMPT, unsigned
etype	integer	OMPT, unsigned

Type Signature

C / C++

```
typedef ompt_set_result_t (*ompt_set_trace_ompt_t) (
  ompt_device_t *device, unsigned int enable, unsigned int etype);
```

C / C++

Semantics

A tool uses the **set_trace_ompt** entry point, which has the **set_trace_ompt** OMPT type, to enable or to disable the recording of standard trace records for one or more types of events that the *etype* argument indicates. If the value of *etype* is zero then the invocation applies to all events. If *etype* is positive then it applies to the event in the **callbacks** OMPT type that matches that value. The *enable* argument indicates whether tracing should be enabled or disabled for the events that *etype* specifies; a positive value indicates that recording should be enabled while a value of zero indicates that recording should be disabled. If *etype* specifies any of the events that correspond to the **target_data_op_emi** or **target_submit_emi** callbacks then tracing, if supported, is enabled or disabled for those events when they occur on the host device. If *etype* specifies any other events then tracing, if supported, is enabled or disabled for those events when they occur on the specified target device. The return value of **set_trace_ompt** indicates the outcome of enabling or disabling the recording of the trace records and can be any value in the **set_result** OMPT type except **ompt_set_sometimes_paired**.

Cross References

- OMPT **callbacks** Type, see Section 33.6
- OMPT **device** Type, see Section 33.11
- Tracing Activity on Target Devices, see Section 32.2.5
- OMPT **set_result** Type, see Section 33.28

37.5 set_trace_native Entry Point

Name: **set_trace_native** **Category:** function	**Properties:** C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	set_result	*default*
device	device	OMPT, pointer
enable	integer	*default*
flags	integer	*default*

Type Signature

C / C++

```
typedef ompt_set_result_t (*ompt_set_trace_native_t) (
  ompt_device_t *device, int enable, int flags);
```

C / C++

Semantics

A tool uses the **set_trace_native** entry point, which has the **set_trace_native** OMPT type, to enable or to disable the recording of native trace records. The *enable* argument indicates whether this invocation should enable or disable recording of events. The *flags* argument specifies the kinds of native device monitoring to enable or to disable. Each kind of monitoring is specified by a flag bit. Flags can be composed by using logical **or** to combine enumeration values of the **native_mon_flag** OMPT type. The return value of **set_trace_native** indicates the outcome of enabling or disabling the recording of the trace records and can be any value in the **set_result** OMPT type except **ompt_set_sometimes_paired**.

This interface is designed for use by a tool that cannot directly use native control procedures for the device. If a tool can directly use the native control procedures then it can invoke them directly using pointers that the **function_lookup** entry point associated with the device provides and that are described in the *documentation* string that is provided to its **device_initialize** callback.

Cross References

- OMPT **device** Type, see Section 33.11
- Tracing Activity on Target Devices, see Section 32.2.5
- OMPT **native_mon_flag** Type, see Section 33.21
- OMPT **set_result** Type, see Section 33.28

37.6 get_buffer_limits Entry Point

Name: get_buffer_limits Category: subroutine	Properties: C/C++-only, OMPT

Arguments

Name	Type	Properties
device	device	OMPT, pointer
max_concurrent_allocs	integer	pointer
recommended_bytes	size_t	pointer

Type Signature

C / C++

```
typedef void (*ompt_get_buffer_limits_t) (ompt_device_t *device,
  int *max_concurrent_allocs, size_t *recommended_bytes);
```

C / C++

Semantics

The **get_buffer_limits** entry point, which has the **get_buffer_limits** OMPT type, enables a tool to retrieve the maximum number of concurrent buffer allocations and the recommended size of any buffer allocation that will be requested of the tool for a specified device. The *max_concurrent_allocs* points to a location in which the entry point returns the maximum number of buffer allocations that the implementation may request for tracing activity on the target device without the implementation performing callback dispatch of **buffer_complete** callbacks with its *buffer_owned* argument set to a non-zero value for any of the buffers. The *recommended_bytes* argument points to a location in which the entry point returns the recommended buffer size of the buffer to be returned by the tool when the implementation dispatches a **buffer_request** callback for the target device.

A tool may use this entry point prior to a call to the **start_trace** entry point to determine the total size of the buffers that the implementation would need for tracing activity on the device at any given time. The limits that this entry point returns remain the same on each successive invocation unless the **stop_trace** entry point is called for the same target device between the successive invocations.

Cross References

- **buffer_complete** Callback, see Section 35.6
- **buffer_request** Callback, see Section 35.5
- OMPT **device** Type, see Section 33.11
- **start_trace** Entry Point, see Section 37.7
- **stop_trace** Entry Point, see Section 37.10

37.7 start_trace Entry Point

Name: start_trace **Category:** function	**Properties:** C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
device	device	OMPT, pointer
request	buffer_request	OMPT, procedure
complete	buffer_complete	OMPT, procedure

Type Signature

C / C++

```
typedef int (*ompt_start_trace_t) (ompt_device_t *device,
  ompt_callback_buffer_request_t request,
  ompt_callback_buffer_complete_t complete);
```

C / C++

Semantics

The **start_trace** entry point, which has the **start_trace** OMPT type, enables a tool to start tracing of activity on a specified device. The *request* argument specifies a callback that supplies a buffer in which a device can deposit events. The *complete* argument specifies a callback that the OpenMP implementation invokes to empty a buffer that contains trace records.

Under normal operating conditions, every event buffer that a tool callback provides for a device is returned to the tool before the OpenMP runtime shuts down. If an exceptional condition terminates execution of an OpenMP program, the runtime may not return buffers provided for the device. An invocation of **start_trace** returns one if the entry point succeeds and zero otherwise.

Cross References

- **buffer_complete** Callback, see Section 35.6
- **buffer_request** Callback, see Section 35.5
- OMPT **device** Type, see Section 33.11

37.8 pause_trace Entry Point

Name: pause_trace **Category:** function	**Properties:** C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
device	device	OMPT, pointer
begin_pause	integer	*default*

Type Signature

C / C++

```
typedef int (*ompt_pause_trace_t) (ompt_device_t *device,
  int begin_pause);
```

C / C++

Semantics

The **pause_trace** entry point, which has the **pause_trace** OMPT type, enables a tool to pause or to resume tracing on a device. The *begin_pause* argument indicates whether to pause or to resume tracing. To resume tracing, zero should be supplied for *begin_pause*; to pause tracing, any other value should be supplied. An invocation of **pause_trace** returns one if it succeeds and zero otherwise. Redundant pause or resume commands are idempotent and will return the same value as the prior command.

Cross References

- OMPT **device** Type, see Section 33.11

37.9 flush_trace Entry Point

Name: flush_trace **Category:** function	**Properties:** C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
device	device	OMPT, pointer

Type Signature

C / C++

```
typedef int (*ompt_flush_trace_t) (ompt_device_t *device);
```

C / C++

Semantics

The **flush_trace** entry point, which has the **flush_trace** OMPT type, enables a tool to cause the OpenMP implementation to issue a sequence of zero or more **buffer_complete** callbacks to deliver all trace records that have been collected prior to the flush for the specified device. An invocation of **flush_trace** returns one if the entry point succeeds and zero otherwise.

Cross References

- OMPT **device** Type, see Section 33.11

37.10 stop_trace Entry Point

Name: stop_trace Category: function	Properties: C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
device	device	OMPT, pointer

Type Signature

C / C++

```
typedef int (*ompt_stop_trace_t) (ompt_device_t *device);
```

C / C++

Semantics

The **stop_trace** entry point, which has the **stop_trace** OMPT type, provides a superset of the functionality of the **flush_trace** entry point. Specifically, the **stop_trace** entry point stops tracing for the specified device in addition to flushing pending trace records. An invocation of **stop_trace** returns one if the entry point succeeds and zero otherwise.

Cross References

- OMPT **device** Type, see Section 33.11
- **flush_trace** Entry Point, see Section 37.9

37.11 advance_buffer_cursor Entry Point

Name: advance_buffer_cursor Category: function	Properties: C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	integer	*default*
device	device	OMPT, pointer
buffer	buffer	OMPT, pointer
size	size_t	*default*
current	buffer_cursor	OMPT, opaque
next	buffer_cursor	OMPT, opaque, pointer

Type Signature

C / C++

```
typedef int (*ompt_advance_buffer_cursor_t) (
  ompt_device_t *device, ompt_buffer_t *buffer, size_t size,
  ompt_buffer_cursor_t current, ompt_buffer_cursor_t *next);
```

C / C++

Semantics

The **advance_buffer_cursor** entry point, which has the **advance_buffer_cursor** OMPT type, enables a tool to advance the trace buffer pointer for the buffer that the *buffer* argument indicates to the next trace record. The *size* argument indicates the size of *buffer* in bytes. The *current* argument is an OpenMP object that indicates the current position, while the *next* argument returns an OpenMP object with the next value. An invocation of **advance_buffer_cursor** returns *true* if the advance is successful and the next position in the buffer is valid. Otherwise, it returns *false*.

Cross References

- OMPT **buffer** Type, see Section 33.3
- OMPT **buffer_cursor** Type, see Section 33.4
- OMPT **device** Type, see Section 33.11

37.12 get_record_type Entry Point

Name: **get_record_type** **Category:** function	**Properties:** C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	record	*default*
buffer	buffer	OMPT, pointer
current	buffer_cursor	OMPT

Type Signature

C / C++

```
typedef ompt_record_t (*ompt_get_record_type_t) (
  ompt_buffer_t *buffer, ompt_buffer_cursor_t current);
```

C / C++

Semantics

Trace records for a device may be in one of two forms: native trace format, which may be device-specific, or standard trace format, in which each trace record corresponds to an OpenMP *event* and most fields in the trace record structure are the arguments that would be passed to the callback for the event. For the buffer specified by the *buffer* argument, the **get_record_type** entry point, which has the **get_record_type** OMPT type, enables a tool to inspect the type of a trace record at the position that the *current* argument specifies and to determine whether the trace record is an OMPT trace record, a native trace record, or is an invalid record, which is returned if the cursor is out of bounds.

Cross References

- OMPT **buffer** Type, see Section 33.3
- OMPT **buffer_cursor** Type, see Section 33.4
- OMPT **record** Type, see Section 33.23

37.13 get_record_ompt Entry Point

Name: get_record_ompt **Category:** function	**Properties:** C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	record_ompt	pointer
buffer	buffer	OMPT, pointer
current	buffer_cursor	OMPT, opaque

Type Signature

C / C++

```
typedef ompt_record_ompt_t *(*ompt_get_record_ompt_t) (
  ompt_buffer_t *buffer, ompt_buffer_cursor_t current);
```

C / C++

Semantics

The **get_record_ompt** entry point, which has the **get_record_ompt** OMPT type, enables a tool to obtain a pointer to an OMPT trace record from a trace buffer associated with a device. The pointer may point to storage in the buffer indicated by the *buffer* argument or it may point to a trace record in thread-local storage in which the information extracted from a trace record was assembled. The information available for an event depends upon its type. The *current* argument is an OpenMP object that indicates the position from which to extract the trace record. The return value of the **record_ompt** OMPT type includes a field of the **any_record_ompt** OMPT type, which is a union that can represent information for any OMPT trace record type. Another call to the entry point may overwrite the contents of the fields in a trace record returned by a prior invocation.

Cross References

- OMPT **any_record_ompt** Type, see Section 33.2
- OMPT **buffer** Type, see Section 33.3
- OMPT **buffer_cursor** Type, see Section 33.4
- OMPT **device** Type, see Section 33.11
- OMPT **record_ompt** Type, see Section 33.26

37.14 get_record_native Entry Point

Name: get_record_native **Category:** function	**Properties:** C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	c_ptr	*default*
buffer	buffer	OMPT, pointer
current	buffer_cursor	OMPT, opaque
host_op_id	id	OMPT, pointer

Type Signature

C / C++

```
typedef void *(*ompt_get_record_native_t) (
  ompt_buffer_t *buffer, ompt_buffer_cursor_t current,
  ompt_id_t *host_op_id);
```

C / C++

Semantics

The **get_record_native** entry point, which has the **get_record_native** OMPT type, enables a tool to obtain a pointer to a native trace record from a trace buffer associated with a device. The pointer may point to storage in the buffer indicated by the *buffer* argument or it may point to a trace record in thread-local storage in which the information extracted from a trace record was assembled. The information available for a native event depends upon its type. The *current* argument is an OpenMP object that indicates the position from which to extract the trace record. If the entry point returns a non-null pointer result, it will also set the object to which the *host_op_id* argument points to a host-side identifier for the operation that is associated with the trace record on the target device and was created when the operation was initiated by the host device. Another call to the entry point may overwrite the contents of the fields in a trace record returned by a prior invocation.

Cross References

- OMPT **buffer** Type, see Section 33.3
- OMPT **buffer_cursor** Type, see Section 33.4
- OMPT **id** Type, see Section 33.18

37.15 get_record_abstract Entry Point

Name: get_record_abstract Category: function	Properties: C/C++-only, OMPT

Return Type and Arguments

Name	Type	Properties
<return type>	record_abstract	pointer
native_record	void	pointer

Type Signature

C / C++

```
typedef ompt_record_abstract_t  *
  (*ompt_get_record_abstract_t) (void *native_record);
```

C / C++

Semantics

An OpenMP implementation may execute on a device that logs trace records in a native trace format that a tool cannot interpret directly. The **get_record_abstract** entry point, which has the **get_record_abstract** OMPT type, enables a tool to translate a native trace record to which the *native_record* argument points into a standard form.

Cross References

- OMPT **record_abstract** Type, see Section 33.24

Part V

OMPD

38 OMPD Overview

This chapter provides an overview of OMPD, which is an interface for third-party tools, such as a debugger. Third-party tools exist in separate processes from the OpenMP program. To provide OMPD support, an OpenMP implementation must provide an OMPD library that the third-party tool can load. An OpenMP implementation does not need to maintain any extra information to support OMPD inquiries from third-party tools unless it is explicitly instructed to do so.

OMPD allows third-party tools to inspect the OpenMP state of a live OpenMP program or core file in an implementation-agnostic manner. Thus, a third-party tool that uses OMPD should work with any compliant implementation. An OpenMP implementation provides a library for OMPD that a third-party tool can dynamically load. The third-party tool can use the interface exported by the OMPD library to inspect the OpenMP state of an OpenMP program. In order to satisfy requests from the third-party tool, the OMPD library may need to read data from the OpenMP program, or to find the addresses of symbols in it. The OMPD library provides this functionality through a callback interface that the third-party tool must instantiate for the OMPD library.

To use OMPD, the third-party tool loads the OMPD library, which exports the OMPD API and which the third-party tool uses to determine OpenMP information about the OpenMP program. The OMPD library must look up symbols and read data out of the program. It does not perform these operations directly but instead directs the third-party tool to perform them by using the callback interface that the third-party tool exports.

The OMPD design insulates third-party tools from the internal structure of the OpenMP runtime, while the OMPD library is insulated from the details of how to access the OpenMP program. This decoupled design allows for flexibility in how the OpenMP program and third-party tool are deployed, so that, for example, the third-party tool and the OpenMP program are not required to execute on the same machine.

Generally, the third-party tool does not interact directly with the OpenMP runtime but instead interacts with the runtime through the OMPD library. However, a few cases require the third-party tool to access the OpenMP runtime directly. These cases fall into two broad categories. The first is during initialization where the third-party tool must look up symbols and read variables in the OpenMP runtime in order to identify the OMPD library that it should use, which is discussed in Section 38.3.2 and Section 38.3.3. The second category relates to arranging for the third-party tool to be notified when certain events occur during the execution of the OpenMP program. For this purpose, the OpenMP implementation must define certain symbols in the runtime code, as is discussed in Chapter 42. Each of these symbols corresponds to an event type. The OpenMP runtime must ensure that control passes through the appropriate named location when events occur. If the third-party tool requires notification of an event, it can plant a breakpoint at the matching location. The location can, but may not, be a function. It can, for example, simply be a label. However, the names of the locations must have external **C** linkage.

38.1 OMPD Interfaces Definitions

C / C++

A compliant implementation must supply a set of definitions for the OMPD third-party tool callback signatures, third-party tool interface routines and the special data types of their parameters and return values. These definitions, which are listed throughout the OMPD chapters, and their associated declarations shall be provided in a header file named **omp-tools.h**. In addition, the set of definitions may specify other implementation defined values. The **ompd_dll_locations** variable and all OMPD third-party tool interface routines are external symbols with **C** linkage.

C / C++

38.2 Thread and Signal Safety

The OMPD library does not need to be reentrant. The third-party tool must ensure that only one native thread enters the OMPD library at a time. The OMPD library must not install signal handlers or otherwise interfere with the signal configuration of the third-party tool.

38.3 Activating a Third-Party Tool

The third-party tool and the OpenMP program exist as separate processes. Thus, OMPD requires coordination between the OpenMP runtime and the third-party tool.

38.3.1 Enabling Runtime Support for OMPD

In order to support third-party tools, the OpenMP runtime may need to collect and to store information that it may not otherwise maintain. The OpenMP runtime collects whatever information is necessary to support OMPD if the *debug-var* ICV is set to **enabled**.

Cross References

- *debug-var* ICV, see Table 3.1

38.3.2 ompd_dll_locations

Format

C

```
extern const char **ompd_dll_locations;
```

C

Semantics

An OpenMP runtime may have more than one OMPD library. The third-party tool must be able to locate the right library to use for the program that it is examining. The **ompd_dll_locations** global variable points to the locations of OMPD libraries that are compatible with the OpenMP implementation. The OpenMP runtime system must provide this public variable, which is an **argv**-style vector of pathname string pointers that provide the names of the compatible OMPD libraries. This variable must have **C** linkage. The third-party tool uses the name of the variable verbatim and, in particular, does not apply any name mangling before performing the look up.

The architecture on which the third-party tool and, thus, the OMPD library execute does not have to match the architecture on which the OpenMP program that is being examined executes. The third-party tool must interpret the contents of **ompd_dll_locations** to find a suitable OMPD library that matches its own architectural characteristics. On platforms that support different architectures (for example, 32-bit vs 64-bit), OpenMP implementations should provide an OMPD library for each supported architecture that can handle OpenMP programs that run on any supported architecture. Thus, for example, a 32-bit debugger that uses OMPD should be able to debug a 64-bit OpenMP program by loading a 32-bit OMPD implementation that can manage a 64-bit OpenMP runtime.

The **ompd_dll_locations** variable points to a NULL-terminated vector of zero or more null-terminated pathname strings that do not have any filename conventions. This vector must be fully initialized *before* **ompd_dll_locations** is set to a non-null value. Thus, if a third-party tool stops execution of the OpenMP program at any point at which **ompd_dll_locations** is a non-null value, the vector of strings to which it points shall be valid and complete.

38.3.3 ompd_dll_locations_valid Breakpoint

Format

C

```
void ompd_dll_locations_valid(void);
```

C

Semantics

Since **ompd_dll_locations** may not be a static variable, it may require runtime initialization. The OpenMP runtime notifies third-party tools that **ompd_dll_locations** is valid by having execution pass through a location that the symbol **ompd_dll_locations_valid** identifies. If **ompd_dll_locations** is NULL, a third-party tool can place a breakpoint at **ompd_dll_locations_valid** to be notified that **ompd_dll_locations** is initialized. In practice, the symbol **ompd_dll_locations_valid** may not be a function; instead, it may be a labeled machine instruction through which execution passes once the vector is valid.

39 OMPD Data Types

This chapter defines OMPD types, which support interactions with the OMPD library and provide information about the device architecture.

39.1 OMPD addr Type

Name: `addr` **Properties:** C/C++-only, OMPD	**Base Type:** `c_uint64_t`

Type Definition

C / C++

```
typedef uint64_t ompd_addr_t;
```

C / C++

Semantics

The **`addr`** OMPD type represents an address in an OpenMP process as an unsigned integer.

39.2 OMPD address Type

Name: `address` **Properties:** C/C++-only, OMPD	**Base Type:** structure

Fields

Name	Type	Properties
segment	seg	C/C++-only, OMPD
address	addr	C/C++-only, OMPD

Type Definition

C / C++

```
typedef struct ompd_address_t {
  ompd_seg_t segment;
  ompd_addr_t address;
} ompd_address_t;
```

C / C++

Semantics

The **address** type is a structure that OMPD uses to specify addresses, which may or may not be segmented. For non-segmented architectures, **ompd_segment_none** is used in the **segment** field of the **address** OMPD type.

Cross References

- OMPD **addr** Type, see Section 39.1
- OMPD **seg** Type, see Section 39.10

39.3 OMPD address_space_context Type

Name: address_space_context Properties: C/C++-only, handle, OMPD	Base Type: aspace_cont

Type Definition

C / C++

```
typedef struct _ompd_aspace_cont ompd_address_space_context_t;
```

C / C++

Semantics

A third-party tool uses the **address_space_context** OMPD type, which represents handles that are opaque to the OMPD library and that define an address space context uniquely, to identify the address space of the OpenMP process that it is monitoring.

39.4 OMPD callbacks Type

Name: callbacks Properties: C/C++-only, OMPD	Base Type: structure

Fields

Name	Type	Properties
alloc_memory	memory_alloc	C-only, OMPD
free_memory	memory_free	C-only, OMPD
print_string	print_string	C-only, OMPD
sizeof_type	sizeof	C-only, OMPD
symbol_addr_lookup	symbol_addr	C-only, OMPD
read_memory	memory_read	C-only, OMPD
write_memory	memory_write	C-only, OMPD
read_string	memory_read	C-only, OMPD
device_to_host	device_host	C-only, OMPD
host_to_device	device_host	C-only, OMPD
get_thread_context_for_thread_id	get_thread_context_for_thread_id	C-only, OMPD

Type Definition

C / C++

```
typedef struct ompd_callbacks_t {
  ompd_callback_memory_alloc_fn_t alloc_memory;
  ompd_callback_memory_free_fn_t free_memory;
  ompd_callback_print_string_fn_t print_string;
  ompd_callback_sizeof_fn_t sizeof_type;
  ompd_callback_symbol_addr_fn_t symbol_addr_lookup;
  ompd_callback_memory_read_fn_t read_memory;
  ompd_callback_memory_write_fn_t write_memory;
  ompd_callback_memory_read_fn_t read_string;
  ompd_callback_device_host_fn_t device_to_host;
  ompd_callback_device_host_fn_t host_to_device;
  ompd_callback_get_thread_context_for_thread_id_fn_t
    get_thread_context_for_thread_id;
} ompd_callbacks_t;
```

C / C++

Semantics

All OMPD library interactions with the OpenMP program must be through a set of callbacks that the third-party tool provides. These callbacks must also be used for allocating or releasing resources, such as memory, that the OMPD library needs. The set of callbacks that the OMPD library must use is collected in an instance of the **callbacks** OMPD type that is passed to the OMPD library as an argument to **ompd_initialize**. Each field points to a procedure that the OMPD library must use to interact with the OpenMP program or for memory operations.

The **alloc_memory** and **free_memory** fields are pointers to **alloc_memory** and **free_memory** callbacks, which the OMPD library uses to allocate and to release dynamic memory. The **print_string** field points to a **print_string** callback that prints a string.

The architecture on which the OMPD library and third-party tool execute may be different from the architecture on which the OpenMP program that is being examined executes. The **sizeof_type** field points to a **sizeof_type** callback that allows the OMPD library to determine the sizes of the basic integer and pointer types that the OpenMP program uses. Because of the potential differences in the targeted architectures, the conventions for representing data in the OMPD library and the OpenMP program may be different. The **device_to_host** field points to a **device_to_host** callback that translates data from the conventions that the OpenMP program uses to those that the third-party tool and OMPD library use. The reverse operation is performed by the **host_to_device** callback to which the **host_to_device** field points.

The **symbol_addr_lookup** field points to a **symbol_addr_lookup** callback, which the OMPD library can use to find the address of a global or thread local storage symbol. The **read_memory**, **read_string** and **write_memory** fields are pointers to **read_memory**, **read_string** and **write_memory** callbacks for reading from and writing to global memory or thread local storage in the OpenMP program.

The **get_thread_context_for_thread_id** field is a pointer to a **get_thread_context_for_thread_id** callback that the OMPD library can use to obtain a native thread context that corresponds to a native thread identifier.

Cross References

- **alloc_memory** Callback, see Section 40.1.1
- **device_to_host** Callback, see Section 40.4.2
- **free_memory** Callback, see Section 40.1.2
- **get_thread_context_for_thread_id** Callback, see Section 40.3.1
- **host_to_device** Callback, see Section 40.4.3
- **ompd_initialize** Routine, see Section 41.1.1
- **print_string** Callback, see Section 40.5
- **read_memory** Callback, see Section 40.2.2.1
- **read_string** Callback, see Section 40.2.2.2
- **sizeof_type** Callback, see Section 40.3.2
- **symbol_addr_lookup** Callback, see Section 40.2.1
- **write_memory** Callback, see Section 40.2.3

39.5 OMPD device Type

Name: device **Properties:** C/C++-only, OMPD	**Base Type:** c_uint64_t

Type Definition

C / C++

```
typedef uint64_t ompd_device_t;
```

C / C++

Semantics

The **device** OMPD type provides information about OpenMP devices. OpenMP runtimes may utilize different underlying devices, each represented by a device identifier. The device identifiers can vary in size and format and, thus, are not explicitly represented in OMPD. Instead, a device identifier is passed across the interface via its **device** kind, its size in bytes and a pointer to where it is stored. The OMPD library and the third-party tool use the **device** kind to interpret the format of the device identifier that is referenced by the pointer argument. Each different device identifier kind is represented by a unique unsigned 64-bit integer value. Recommended values of **device** kinds are defined in the **ompd-types.h** header file, which is contained in the *Supplementary Source Code* package available via https://www.openmp.org/specifications/.

39.6 OMPD device_type_sizes Type

Name: `device_type_sizes` **Properties:** C/C++-only, OMPD	**Base Type:** structure

Fields

Name	Type	Properties
sizeof_char	c_uint8_t	C/C++-only, OMPD
sizeof_short	c_uint8_t	C/C++-only, OMPD
sizeof_int	c_uint8_t	C/C++-only, OMPD
sizeof_long	c_uint8_t	C/C++-only, OMPD
sizeof_long_long	c_uint8_t	C/C++-only, OMPD
sizeof_pointer	c_uint8_t	C/C++-only, OMPD

Type Definition

C / C++

```
typedef struct ompd_device_type_sizes_t {
  uint8_t sizeof_char;
  uint8_t sizeof_short;
  uint8_t sizeof_int;
  uint8_t sizeof_long;
  uint8_t sizeof_long_long;
  uint8_t sizeof_pointer;
} ompd_device_type_sizes_t;
```

C / C++

Semantics

The **`device_type_sizes`** OMPD type is used in OMPD callbacks through which the OMPD library can interrogate a third-party tool about the size of primitive types for the target architecture of the OpenMP runtime, as returned by the **`sizeof`** operator. The fields of **`device_type_sizes`** give the sizes of the eponymous basic types used by the OpenMP runtime. As the third-party tool and the OMPD library, by definition, execute on the same architecture, the size of the fields can be given as **`uint8_t`**.

Cross References

- **`sizeof_type`** Callback, see Section 40.3.2

39.7 OMPD frame_info Type

Name: `frame_info` **Properties:** C/C++-only, OMPD	**Base Type:** structure

Fields

Name	Type	Properties
frame_address	address	C/C++-only, OMPD
frame_flag	word	C/C++-only, OMPD

Type Definition

C / C++

```
typedef struct ompd_frame_info_t {
  ompd_address_t frame_address;
  ompd_word_t frame_flag;
} ompd_frame_info_t;
```

C / C++

Semantics

The **frame_info** OMPD type is a structure type that OMPD uses to specify frame information. The **frame_address** field of **frame_info** identifies a frame. The **frame_flag** field of **frame_info** indicates what type of information is provided in **frame_address**. The values and meaning are the same as are defined for the **frame_flag** OMPT type.

Cross References

- OMPD **address** Type, see Section 39.2
- OMPT **frame_flag** Type, see Section 33.16
- OMPD **word** Type, see Section 39.17

39.8 OMPD icv_id Type

Name: **icv_id**	Base Type: **c_uint64_t**
Properties: C/C++-only, OMPD	

Predefined Identifiers

Name	Value	Properties
ompd_icv_undefined	0	C/C++-only, OMPD

Type Definition

C / C++

```
typedef uint64_t ompd_icv_id_t;
```

C / C++

Semantics

The **icv_id** OMPD type identifies ICVs.

39.9 OMPD `rc` Type

Name: `rc` **Properties:** C/C++-only, OMPD	**Base Type:** enumeration

Values

Name	Value	Properties
`ompd_rc_ok`	0	C-only, OMPD
`ompd_rc_unavailable`	1	C-only, OMPD
`ompd_rc_stale_handle`	2	C-only, OMPD
`ompd_rc_bad_input`	3	C-only, OMPD
`ompd_rc_error`	4	C-only, OMPD
`ompd_rc_unsupported`	5	C-only, OMPD
`ompd_rc_needs_state_tracking`	6	C-only, OMPD
`ompd_rc_incompatible`	7	C-only, OMPD
`ompd_rc_device_read_error`	8	C-only, OMPD
`ompd_rc_device_write_error`	9	C-only, OMPD
`ompd_rc_nomem`	10	C-only, OMPD
`ompd_rc_incomplete`	11	C-only, OMPD
`ompd_rc_callback_error`	12	C-only, OMPD
`ompd_rc_incompatible_handle`	13	C-only, OMPD

Type Definition

C / C++

```
typedef enum ompd_rc_t {
  ompd_rc_ok                   = 0,
  ompd_rc_unavailable          = 1,
  ompd_rc_stale_handle         = 2,
  ompd_rc_bad_input            = 3,
  ompd_rc_error                = 4,
  ompd_rc_unsupported          = 5,
  ompd_rc_needs_state_tracking = 6,
  ompd_rc_incompatible         = 7,
  ompd_rc_device_read_error    = 8,
  ompd_rc_device_write_error   = 9,
  ompd_rc_nomem                = 10,
  ompd_rc_incomplete           = 11,
  ompd_rc_callback_error       = 12,
  ompd_rc_incompatible_handle  = 13
} ompd_rc_t;
```

C / C++

The **rc** OMPD type is the return code type of OMPD routines and OMPD callbacks. The values of the **rc** OMPD type and their semantics are defined as follows:

- **ompd_rc_ok**: The routine or callback procedure was successful;
- **ompd_rc_unavailable**: Information was not available for the specified context;
- **ompd_rc_stale_handle**: The specified handle was not valid;
- **ompd_rc_bad_input**: The arguments (other than handles) are invalid;
- **ompd_rc_error**: A fatal error occurred;
- **ompd_rc_unsupported**: The requested routine or callback is not supported for the specified arguments;
- **ompd_rc_needs_state_tracking**: The state tracking operation failed because state tracking was not enabled;
- **ompd_rc_incompatible**: The selected OMPD library was incompatible with the OpenMP program or was incapable of handling it;
- **ompd_rc_device_read_error**: A read operation failed on the device;
- **ompd_rc_device_write_error**: A write operation failed on the device;
- **ompd_rc_nomem**: A memory allocation failed;
- **ompd_rc_incomplete**: The information provided on return was incomplete, while the arguments were set to valid values;
- **ompd_rc_callback_error**: The callback interface or one of the required callback procedures provided by the third-party tool was invalid; and
- **ompd_rc_incompatible_handle**: The specified handle was incompatible with the routine or callback.

39.10 OMPD seg Type

Name: **seg** **Properties:** C/C++-only, OMPD	**Base Type:** **c_uint64_t**

Predefined Identifiers

Name	Value	Properties
ompd_segment_none	0	C/C++-only, OMPD

Type Definition

C / C++

```
typedef uint64_t ompd_seg_t;
```

C / C++

Semantics

The **seg** OMPD type represents a segment value as an unsigned integer.

39.11 OMPD scope Type

Name: scope Properties: C/C++-only, OMPD	Base Type: enumeration

Values

Name	Value	Properties
ompd_scope_global	1	C-only, OMPD
ompd_scope_address_space	2	C-only, OMPD
ompd_scope_thread	3	C-only, OMPD
ompd_scope_parallel	4	C-only, OMPD
ompd_scope_implicit_task	5	C-only, OMPD
ompd_scope_task	6	C-only, OMPD
ompd_scope_teams	7	C-only, OMPD
ompd_scope_target	8	C-only, OMPD

Type Definition

C / C++

```
typedef enum ompd_scope_t {
  ompd_scope_global         = 1,
  ompd_scope_address_space = 2,
  ompd_scope_thread         = 3,
  ompd_scope_parallel       = 4,
  ompd_scope_implicit_task = 5,
  ompd_scope_task           = 6,
  ompd_scope_teams          = 7,
  ompd_scope_target         = 8
} ompd_scope_t;
```

C / C++

Semantics

The **scope** OMPD type is used for scope handles to identify OpenMP scopes, including those related to parallel regions and tasks. When used in an OMPD routine or OMPD callback procedure, the **scope** OMPD type and the OMPD handle must match according to Table 39.1.

39.12 OMPD size Type

Name: size Properties: C/C++-only, OMPD	Base Type: c_uint64_t

TABLE 39.1: Mapping of Scope Type and OMPD Handles

Scope types	Handles
ompd_scope_global	Address space handle for the host device
ompd_scope_address_space	Any address space handle
ompd_scope_thread	Any native thread handle
ompd_scope_parallel	Any parallel handle
ompd_scope_implicit_task	Task handle for an implicit task
ompd_scope_teams	Parallel handle for an implicit parallel region generated from a **teams** construct
ompd_scope_target	Parallel handle for an implicit parallel region generated from a **target** construct
ompd_scope_task	Any task handle

Type Definition

C / C++

```
typedef uint64_t ompd_size_t;
```

C / C++

The **size** OMPD type specifies the number of bytes in opaque data objects that are passed across the OMPD API.

39.13 OMPD team_generator Type

Name: team_generator **Properties:** C/C++-only, OMPD	**Base Type:** enumeration

Values

Name	Value	Properties
ompd_generator_program	0	C-only, OMPD
ompd_generator_parallel	1	C-only, OMPD
ompd_generator_teams	2	C-only, OMPD
ompd_generator_target	3	C-only, OMPD

Type Definition

C / C++

```
typedef enum ompd_team_generator_t {
  ompd_generator_program  = 0,
  ompd_generator_parallel = 1,
  ompd_generator_teams    = 2,
  ompd_generator_target   = 3
} ompd_team_generator_t;
```

C / C++

Semantics

The **team_generator** OMPD type represents the value of the *team-generator-var* ICV. The **ompd_generator_program** value indicates that the team is the initial team created at the start of the OpenMP program. The **ompd_generator_parallel**, **ompd_generator_teams**, and **ompd_generator_target** values indicate that the team was created by an encountered **parallel** construct, **teams** construct, or **target** construct, respectively.

39.14 OMPD thread_context Type

Name: thread_context Properties: C/C++-only, handle, OMPD	Base Type: thread_cont

Type Definition

C / C++

```
typedef struct _ompd_thread_cont ompd_thread_context_t;
```

C / C++

Semantics

A third-party tool uses the **thread_context** OMPD type, which represents handles that are opaque to the OMPD library and that uniquely identify a native thread of the OpenMP process that it is monitoring.

39.15 OMPD thread_id Type

Name: thread_id Properties: C/C++-only, OMPD	Base Type: c_uint64_t

Type Definition

C / C++

```
typedef uint64_t ompd_thread_id_t;
```

C / C++

Semantics

The **thread_id** OMPD type provides information about native threads. OpenMP runtimes may use different native thread implementations. Native thread identifiers for these implementations can vary in size and format and, thus, are not explicitly represented in OMPD. Instead, a native thread identifier is passed across the interface via the **thread_id** OMPD type, its size in bytes and a pointer to where it is stored. The OMPD library and the third-party tool use the **thread_id** OMPD type to interpret the format of the native thread identifier that is referenced by the pointer argument. Each different native thread identifier kind is represented by a unique unsigned 64-bit integer value. Recommended values of the **thread_id** OMPD type and formats for some corresponding native thread identifiers are defined in the **ompd-types.h** header file, which is contained in the *Supplementary Source Code* package available via https://www.openmp.org/specifications/.

39.16 OMPD wait_id Type

Name: wait_id **Properties:** C/C++-only, OMPD	**Base Type:** c_uint64_t

Type Definition

C / C++

```
typedef uint64_t ompd_wait_id_t;
```

C / C++

Semantics

A variable of **wait_id** OMPD type identifies the object on which a thread waits. The values and meaning of **wait_id** are the same as those defined for the **wait_id** OMPT type.

Cross References

- OMPT **wait_id** Type, see Section 33.40

39.17 OMPD word Type

Name: word **Properties:** C/C++-only, OMPD	**Base Type:** c_int64_t

Type Definition

C / C++

```
typedef int64_t ompd_word_t;
```

C / C++

Semantics

The **word** OMPD type represents a data word from the OpenMP runtime as a signed integer.

39.18 OMPD Handle Types

The OMPD library defines handles, which have OMPD types that are handle types (i.e., they have the handle property). These handles are used to refer to address spaces, threads, parallel regions and tasks and are managed by the OpenMP runtime. The internal structures that these handles represent are opaque to the third-party tool. Defining externally visible type names in this way introduces type safety to the interface and helps to catch instances where incorrect handles are passed by a third-party tool to the OMPD library. The structures do not need to be defined; instead, the OMPD library must cast incoming (pointers to) handles to the appropriate internal, private types.

Each OMPD routine or OMPD callback procedure that applies to a particular address space, thread, parallel region or task must explicitly specify a corresponding handle. A handle remains constant and valid while the associated entity is managed by the OpenMP runtime or until it is released with the corresponding OMPD routine for releasing handles of that type. If a third-party tool receives notification of the end of the lifetime of a managed entity (see Chapter 42) or it releases the handle, the handle may no longer be referenced.

39.18.1 OMPD address_space_handle Type

Name: `address_space_handle` **Properties:** C/C++-only, handle, OMPD	**Base Type:** `aspace_handle`

Type Definition

C / C++

```
typedef struct _ompd_aspace_handle ompd_address_space_handle_t;
```

C / C++

Semantics

The **address_space_handle** OMPD type is used for handles that represent address spaces.

39.18.2 OMPD parallel_handle Type

Name: `parallel_handle` **Properties:** C/C++-only, handle, OMPD	**Base Type:** `parallel_handle`

Type Definition

C / C++

```
typedef struct _ompd_parallel_handle ompd_parallel_handle_t;
```

C / C++

Semantics

The **parallel_handle** OMPD type is used for parallel handles that represent parallel regions.

39.18.3 OMPD task_handle Type

Name: **task_handle** **Properties:** C/C++-only, handle, OMPD	**Base Type:** **task_handle**

Type Definition

C / C++

```
typedef struct _ompd_task_handle ompd_task_handle_t;
```

C / C++

Semantics

The **task_handle** OMPD type is used for handles that represent tasks.

39.18.4 OMPD thread_handle Type

Name: **thread_handle** **Properties:** C/C++-only, handle, OMPD	**Base Type:** **thread_handle**

Type Definition

C / C++

```
typedef struct _ompd_thread_handle ompd_thread_handle_t;
```

C / C++

Semantics

The **thread_handle** OMPD type is used for handles that represent threads.

40 OMPD Callback Interface

For the OMPD library to provide information about the internal state of the OpenMP runtime system in an OpenMP process or core file, it must be able to extract information from the OpenMP process that the third-party tool is examining. The process on which the tool is operating may be either a live process or a core file, and a thread may be either a live thread in a live process or a thread in a core file. To enable the OMPD library to extract state information from a process or core file, the tool must supply the OMPD library with callbacks to inquire about the size of primitive types in the device of the process, to look up the addresses of symbols, and to read and to write memory in the device. The OMPD library uses these callbacks to implement its interface operations. The OMPD library only invokes the OMPD callbacks in response to calls to the OMPD library. The names of the OMPD callbacks correspond to the names of the fields of the **callbacks** OMPD type.

Restrictions

The following restrictions apply to all OMPD callbacks:

- Unless explicitly specified otherwise, all OMPD callbacks must return **ompd_rc_ok** or **ompd_rc_stale_handle**.

40.1 Memory Management of OMPD Library

A tool provides **alloc_memory** and **free_memory** callbacks to obtain and to release heap memory. This mechanism ensures that the OMPD library does not interfere with any custom memory management scheme that the tool may use.

If the OMPD library is implemented in C++ then memory management operators, like **new** and **delete** and their variants, must all be overloaded and implemented in terms of the callbacks that the third-party tool provides. The OMPD library must be implemented such that any of its definitions of **new** and **delete** do not interfere with any that the tool defines. In some cases, the OMPD library must allocate memory to return results to the tool. The tool then owns this memory and has the responsibility to release it. Thus, the OMPD library and the tool must use the same memory manager.

The OMPD library creates OMPD handles, which are opaque to tools and may have a complex internal structure. A tool cannot determine if the handle pointers that OMPD returns correspond to discrete heap allocations. Thus, the tool must not simply deallocate a handle by passing an address that it receives from the OMPD library to its own memory manager. Instead, OMPD includes routines that the tool must use when it no longer needs a handle.

A tool creates tool contexts and passes them to the OMPD library. The OMPD library does not release tool contexts; instead the tool releases them after it releases any handles that may reference the tool contexts.

Cross References

- **alloc_memory** Callback, see Section 40.1.1
- **free_memory** Callback, see Section 40.1.2

40.1.1 alloc_memory Callback

Name: **alloc_memory** **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
nbytes	size	*default*
ptr	void	pointer-to-pointer

Type Signature

C

```c
typedef ompd_rc_t (*ompd_callback_memory_alloc_fn_t) (
  ompd_size_t nbytes, void **ptr);
```

C

Semantics

A tool provides an **alloc_memory** callback, which has the **memory_alloc** OMPD type, that the OMPD library may call to allocate memory. The *nbytes* argument is the size in bytes of the block of memory to allocate. The address of the newly allocated block of memory is returned in the location to which the *ptr* argument points. The newly allocated block is suitably aligned for any type of variable but is not guaranteed to be set to zero.

Cross References

- OMPD **rc** Type, see Section 39.9
- OMPD **size** Type, see Section 39.12

40.1.2 free_memory Callback

Name: **free_memory** **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
ptr	void	pointer

Type Signature

C

```
typedef ompd_rc_t (*ompd_callback_memory_free_fn_t) (void *ptr);
```

C

Semantics

A tool provides a **free_memory** callback, which has the **memory_free** OMPD type, that the OMPD library may call to deallocate memory that was obtained from a prior call to the **alloc_memory** callback. The *ptr* argument is the address of the block to be deallocated.

Cross References

- **alloc_memory** Callback, see Section 40.1.1
- OMPD **rc** Type, see Section 39.9

40.2 Accessing Program or Runtime Memory

The OMPD library cannot directly read from or write to memory of the OpenMP program. Instead the OMPD library must use callbacks into the third-party tool that perform the operation.

40.2.1 symbol_addr_lookup Callback

Name: **symbol_addr_lookup** Category: function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
address_space_context	address_space_context	pointer
thread_context	thread_context	pointer
symbol_name	char	intent(in), pointer
symbol_addr	address	pointer
file_name	char	intent(in), pointer

Type Signature

C

```
typedef ompd_rc_t (*ompd_callback_symbol_addr_fn_t) (
  ompd_address_space_context_t *address_space_context,
  ompd_thread_context_t *thread_context, const char *symbol_name,
  ompd_address_t *symbol_addr, const char *file_name);
```

C

Semantics

A tool provides a **symbol_addr_lookup** callback, which has the **symbol_addr** OMPD type, that the OMPD library may call to look up the address of the symbol provided in the *symbol_name* argument within the address space specified by the *address_space_context* argument. This argument provides the tool's representation of the address space of the process, core file, or device.

The *thread_context* argument is NULL for global memory accesses. If *thread_context* is not NULL, *thread_context* gives the native thread context for the symbol lookup for the purpose of calculating thread local storage addresses. In this case, the native thread to which *thread_context* refers must be associated with either the OpenMP process or the device that corresponds to the *address_space_context* argument.

The tool uses the *symbol_name* argument that the OMPD library supplies verbatim. In particular, no name mangling, demangling or other transformations are performed before the lookup. The *symbol_name* parameter must correspond to a statically allocated symbol within the specified address space. The symbol can correspond to any type of object, such as a variable, thread local storage variable, procedure, or untyped label. The symbol can have local, global, or weak binding. The callback returns the address of the symbol in the location to which *symbol_addr* points.

The *file_name* argument is an optional input argument that indicates the name of the shared library in which the symbol is defined, and it is intended to help the third-party tool disambiguate symbols that are defined multiple times across the executable or shared library files. The shared library name may not be an exact match for the name seen by the third-party tool. If *file_name* is NULL then the third-party tool first tries to find the symbol in the executable file, and, if the symbol is not found, the third-party tool tries to find the symbol in the shared libraries in the order in which the shared libraries are loaded into the address space. If *file_name* is a non-null value then the third-party tool first tries to find the symbol in the libraries that match the name in the *file_name* argument, and, if the symbol is not found, the third-party tool then uses the same lookup order as when *file_name* is NULL.

In addition to the general return codes for OMPD callbacks, **symbol_addr_lookup** callbacks may also return the following return codes:

- **ompd_rc_error** if the symbol that the *symbol_name* argument specifies is not found; or
- **ompd_rc_bad_input** if no symbol name is provided.

Restrictions

Restrictions on **symbol_addr_lookup** callbacks are as follows:

- The *address_space_context* argument must be a non-null value.
- The callback does not support finding either symbols that are dynamically allocated on the call stack or statically allocated symbols that are defined within the scope of a procedure.

Cross References

- OMPD **address** Type, see Section 39.2

- OMPD **address_space_context** Type, see Section 39.3
- OMPD **rc** Type, see Section 39.9
- OMPD **thread_context** Type, see Section 39.14

40.2.2 OMPD memory_read Type

Name: memory_read **Category:** function pointer	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
address_space_context	address_space_context	pointer
thread_context	thread_context	pointer
addr	address	intent(in), pointer
nbytes	size	*default*
buffer	void	pointer

Type Signature

C

```
typedef ompd_rc_t (*ompd_callback_memory_read_fn_t) (
  ompd_address_space_context_t *address_space_context,
  ompd_thread_context_t *thread_context,
  const ompd_address_t *addr, ompd_size_t nbytes, void *buffer);
```

C

Callbacks that have the **memory_read** OMPD type are memory-reading callbacks, which each have the memory-reading property. A tool provides these callbacks to read memory from an OpenMP program. The *thread_context* argument of this type should be NULL for global memory accesses. If it is a non-null value, the *thread_context* argument identifies the native thread context for the memory access for the purpose of accessing thread local storage. The data are returned through the *buffer* argument, which is allocated and owned by the OMPD library. The contents of the buffer are unstructured, raw bytes. The OMPD library must use the **device_to_host** callback to perform any transformations such as byte-swapping that may be necessary.

In addition to the general return codes for OMPD callbacks, memory-reading callbacks may also return the following return code:

- **ompd_rc_error** if unallocated memory is reached while reading *nbytes*.

Cross References

- OMPD **address** Type, see Section 39.2
- OMPD **address_space_context** Type, see Section 39.3
- **device_to_host** Callback, see Section 40.4.2
- OMPD **rc** Type, see Section 39.9
- OMPD **size** Type, see Section 39.12
- OMPD **thread_context** Type, see Section 39.14

40.2.2.1 read_memory Callback

Name: **read_memory** Category: function	**Properties:** C-only, common-type-callback, memory-reading, OMPD

Type Signature

memory_read

Semantics

A tool provides a **read_memory** callback, which is a memory-reading callback, that the OMPD library may call to copy a block of data from *addr* within the address space given by *address_space_context* to the tool *buffer*.

Cross References

- OMPD **address** Type, see Section 39.2
- OMPD **address_space_context** Type, see Section 39.3
- OMPD **memory_read** Type, see Section 40.2.2

40.2.2.2 read_string Callback

Name: **read_string** Category: function	**Properties:** C-only, common-type-callback, memory-reading, OMPD

Type Signature

memory_read

Semantics

A tool provides a **read_string** callback, which is a memory-reading callback, that the OMPD library may call to copy a string to which *addr* points, including the terminating null byte (**'\0'**), to the tool *buffer*. At most *nbytes* bytes are copied. If a null byte is not among the first *nbytes* bytes, the string placed in *buffer* is not null-terminated.

In addition to the general return codes for memory-reading callbacks, **read_string** callbacks may also return the following return code:

- **ompd_rc_incomplete** if no terminating null byte is found while reading *nbytes* using the **read_string** callback.

Cross References

- OMPD **rc** Type, see Section 39.9
- OMPD **size** Type, see Section 39.12

40.2.3 write_memory Callback

Name: write_memory **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
address_space_context	address_space_context	pointer
thread_context	thread_context	pointer
addr	address	intent(in), pointer
nbytes	size	*default*
buffer	void	pointer

Type Signature

C

```
typedef ompd_rc_t (*ompd_callback_memory_write_fn_t) (
  ompd_address_space_context_t *address_space_context,
  ompd_thread_context_t *thread_context,
  const ompd_address_t *addr, ompd_size_t nbytes, void *buffer);
```

C

Semantics

A tool provides a **write_memory** callback, which has the **memory_write** OMPD type, that the OMPD library may call to have the tool write a block of data to a location within an address space from a provided buffer. The address to which the data are to be written in the OpenMP program that *address_space_context* specifies is given by *addr*. The *nbytes* argument is the number of bytes to be transferred. The *thread_context* argument for global memory accesses should be NULL. If it is a non-null value, then *thread_context* identifies the native thread context for the memory access for the purpose of accessing thread local storage.

The data to be written are passed through *buffer*, which is allocated and owned by the OMPD library. The contents of the buffer are unstructured, raw bytes. The OMPD library must use the **host_to_device** callback to perform any transformations such as byte-swapping that may be necessary to render the data into a form that is compatible with the OpenMP runtime.

In addition to the general return codes for OMPD callbacks, **write_memory** callbacks may also return the following return codes:

- **ompd_rc_error** if unallocated memory is reached while writing *nbytes*.

Cross References

- OMPD **address** Type, see Section 39.2
- OMPD **address_space_context** Type, see Section 39.3
- **host_to_device** Callback, see Section 40.4.3
- OMPD **rc** Type, see Section 39.9
- OMPD **size** Type, see Section 39.12
- OMPD **thread_context** Type, see Section 39.14

40.3 Context Management and Navigation

Summary

A tool provides callbacks to manage and to navigate tool context relationships.

40.3.1 get_thread_context_for_thread_id Callback

Name: get_thread_context_for_thread_id Category: function	Properties: C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
address_space_context	address_space_context	opaque, pointer
kind	thread_id	*default*
sizeof_thread_id	size	*default*
thread_id	void	intent(in), pointer
thread_context	thread_context	pointer-to-pointer

Type Signature

C

```
typedef ompd_rc_t
  (*ompd_callback_get_thread_context_for_thread_id_fn_t) (
  ompd_address_space_context_t *address_space_context,
  ompd_thread_id_t kind, ompd_size_t sizeof_thread_id,
  const void *thread_id, ompd_thread_context_t **thread_context);
```

C

Semantics

A tool provides a **get_thread_context_for_thread_id** callback, which has the **get_thread_context_for_thread_id** OMPD type, that the OMPD library may call to map a native thread identifier to a third-party tool native thread context. The native thread identifier is within the address space that *address_space_context* identifies. The OMPD library can use the native thread context, for example, to access thread local storage.

The *address_space_context* argument is an opaque handle that the tool provides to reference an address space. The *kind*, *sizeof_thread_id*, and *thread_id* arguments represent a native thread identifier. On return, the *thread_context* argument provides a handle that maps a native thread identifier to a tool native thread context.

In addition to the general return codes for OMPD callbacks, **get_thread_context_for_thread_id** callbacks may also return the following return codes:

- **ompd_rc_bad_input** if a different value in *sizeof_thread_id* is expected for the native thread identifier kind given by *kind*; or
- **ompd_rc_unsupported** if the native thread identifier *kind* is not supported.

Restrictions

Restrictions on **get_thread_context_for_thread_id** callbacks are as follows:

- The provided *thread_context* must be valid until the OMPD library returns from the tool procedure.

Cross References

- OMPD **address_space_context** Type, see Section 39.3
- OMPD **rc** Type, see Section 39.9
- OMPD **size** Type, see Section 39.12
- OMPD **thread_context** Type, see Section 39.14
- OMPD **thread_id** Type, see Section 39.15

40.3.2 sizeof_type Callback

Name: **sizeof_type** Category: function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
address_space_context	address_space_context	pointer
sizes	device_type_sizes	pointer

Type Signature

C

```
typedef ompd_rc_t (*ompd_callback_sizeof_fn_t) (
  ompd_address_space_context_t *address_space_context,
  ompd_device_type_sizes_t *sizes);
```

C

Semantics

A tool provides a **sizeof_type** callback, which has the **sizeof** OMPD type, that the OMPD library may call to query the sizes of the basic primitive types for the address space that the *address_space_context* argument specifies in the location to which *sizes* points.

Cross References

- OMPD **address_space_context** Type, see Section 39.3
- OMPD **device_type_sizes** Type, see Section 39.6
- OMPD **rc** Type, see Section 39.9

40.4 Device Translating Callbacks

Summary

A tool provides device-translating callbacks, which have the device-translating property, to perform any necessary translations between devices on which the tool and OMPD library run and on which the OpenMP program runs.

40.4.1 OMPD device_host Type

Name: **device_host** Category: function pointer	Properties: C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
address_space_context	address_space_context	pointer
input	void	intent(in), pointer
unit_size	size	*default*
count	size	*default*
output	void	pointer

Type Signature

C

```
typedef ompd_rc_t (*ompd_callback_device_host_fn_t) (
  ompd_address_space_context_t *address_space_context,
  const void *input, ompd_size_t unit_size, ompd_size_t count,
  void *output);
```

C

Semantics

The architecture on which the third-party tool and the OMPD library execute may be different from the architecture on which the OpenMP program that is being examined executes. Thus, the conventions for representing data may differ. The callback interface includes operations to convert between the conventions, such as the byte order (endianness), that the tool and OMPD library use and the ones that the OpenMP program uses. The **device_host** OMPD type is the type signature of the **device_to_host** and **host_to_device** callbacks that the tool provides to convert data between formats.

The *address_space_context* argument specifies the address space that is associated with the data. The *input* argument is the source buffer and the *output* argument is the destination buffer. The *unit_size* argument is the size of each of the elements to be converted. The *count* argument is the number of elements to be transformed.

The OMPD library allocates and owns the input and output buffers. It must ensure that the buffers have the correct size and are eventually deallocated when they are no longer needed.

Cross References

- OMPD **address_space_context** Type, see Section 39.3
- **device_to_host** Callback, see Section 40.4.2
- **host_to_device** Callback, see Section 40.4.3
- OMPD **rc** Type, see Section 39.9
- OMPD **size** Type, see Section 39.12

40.4.2 device_to_host Callback

Name: **device_to_host** Category: function	**Properties:** C-only, common-type-callback, device-translating, OMPD

Type Signature

device_host

Semantics

The **device_to_host** is the device-translating callback that translates data that is read from the OpenMP program.

Cross References

- OMPD **device_host** Type, see Section 40.4.1

40.4.3 host_to_device Callback

Name: **host_to_device** Category: function	**Properties:** C-only, common-type-callback, device-translating, OMPD

Type Signature

device_host

Semantics

The **host_to_device** is the device-translating callback that translates data that is to be written to the OpenMP program.

Cross References

- OMPD **device_host** Type, see Section 40.4.1

40.5 print_string Callback

Name: **print_string** Category: function	Properties: C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
string	char	intent(in), pointer
category	integer	*default*

Type Signature

C

```
typedef ompd_rc_t (*ompd_callback_print_string_fn_t) (
  const char *string, int category);
```

C

Semantics

A tool provides a **print_string** callback, which has the **print_string** OMPD type, that the OMPD library may call to emit output, such as logging or debug information. The tool may set the **print_string** callback to NULL to prevent the OMPD library from emitting output. The OMPD library may not write to file descriptors that it did not open. The *string* argument is the null-terminated string to be printed; no conversion or formatting is performed on the string. The *category* argument is the implementation defined category of the string to be printed.

Cross References

- OMPD **rc** Type, see Section 39.9

41 OMPD Routines

This chapter defines the OMPD routines, which are routines that have the OMPD property and, thus, are provided by the OMPD library to be used by third-party tools. Some OMPD routines require one or more specified threads to be *stopped* for the returned values to be meaningful. In this context, a stopped thread is a thread that is not modifying the observable OpenMP runtime state.

41.1 OMPD Library Initialization and Finalization

The OMPD library must be initialized exactly once after it is loaded, and finalized exactly once before it is unloaded. Per OpenMP process or core file initialization and finalization are also required. Once loaded, the tool can determine the version of the OMPD API that the library supports by calling **ompd_get_api_version**. If the tool supports the version that **ompd_get_api_version** returns, the tool starts the initialization by calling **ompd_initialize** using the version of the OMPD API that the library supports. If the tool does not support the version that **ompd_get_api_version** returns, it may attempt to call **ompd_initialize** with a different version.

Cross References

- **ompd_get_api_version** Routine, see Section 41.1.2
- **ompd_initialize** Routine, see Section 41.1.1

41.1.1 ompd_initialize Routine

Name: **ompd_initialize** **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
api_version	word	*default*
callbacks	callbacks	intent(in), pointer

Prototypes

C

```
ompd_rc_t ompd_initialize(ompd_word_t api_version,
  const ompd_callbacks_t *callbacks);
```

C

Semantics

A tool that uses OMPD calls **ompd_initialize** to initialize each OMPD library that it loads. More than one library may be present in a third-party tool because the tool may control multiple devices, which may use different runtime systems that require different OMPD libraries. This initialization must be performed exactly once before the tool can begin to operate on an OpenMP process or core file.

The *api_version* argument is the OMPD API version that the tool requests to use. The tool may call **ompd_get_api_version** to obtain the latest OMPD API version that the OMPD library supports.

The tool provides the OMPD library with a set of callbacks in the *callbacks* input argument, which enables the OMPD library to allocate and to deallocate memory in the address space of the tool, to lookup the sizes of basic primitive types in the device, to lookup symbols in the device, and to read and to write memory in the device.

This routine returns **ompd_rc_bad_input** if invalid callbacks are provided. In addition to the return codes permitted for all OMPD routines, this routine may return **ompd_rc_unsupported** if the requested API version cannot be provided.

Cross References

- OMPD **callbacks** Type, see Section 39.4
- **ompd_get_api_version** Routine, see Section 41.1.2
- OMPD **rc** Type, see Section 39.9
- OMPD **word** Type, see Section 39.17

41.1.2 ompd_get_api_version Routine

Name: ompd_get_api_version **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
api_version	word	pointer

Prototypes

C

```
ompd_rc_t ompd_get_api_version(ompd_word_t *api_version);
```

C

Semantics

The tool may call the **ompd_get_api_version** routine to obtain the latest OMPD API version number of the OMPD library. The OMPD API version number is equal to the value of the **_OPENMP** macro defined in the associated OpenMP implementation, if the C preprocessor is supported. If the associated OpenMP implementation compiles Fortran codes without the use of a C preprocessor, the OMPD API version number is equal to the value of the **openmp_version** predefined identifier. The latest version number is returned into the location to which the *version* argument points.

Cross References

- **ompd_initialize** Routine, see Section 41.1.1
- OMPD **rc** Type, see Section 39.9
- OMPD **word** Type, see Section 39.17

41.1.3 ompd_get_version_string Routine

Name: **ompd_get_version_string** **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
string	const char	intent(out), pointer-to-pointer

Prototypes

C

```
ompd_rc_t ompd_get_version_string(const char **string);
```

C

Semantics

The **ompd_get_version_string** routine returns a pointer to a descriptive version string of the OMPD library vendor, implementation, internal version, date, or any other information that may be useful to a tool user or vendor. An implementation should provide a different string for every change to its source code or build that could be visible to the OMPD user.

A pointer to a descriptive version string is placed into the location to which the *string* output argument points. The OMPD library owns the string that the OMPD library returns; the tool must not modify or release this string. The string remains valid for as long as the library is loaded. The **ompd_get_version_string** routine may be called before **ompd_initialize**. Accordingly, the OMPD library must not use heap or stack memory for the string.

The signatures of **ompd_get_api_version** and **ompd_get_version_string** are guaranteed not to change in future versions of OMPD. In contrast, the type definitions and prototypes in the rest of OMPD do not carry the same guarantee. Therefore a tool that uses OMPD should check the version of the loaded OMPD library before it calls any other OMPD routine.

Cross References

- OMPD **address_space_handle** Type, see Section 39.18.1
- **ompd_get_api_version** Routine, see Section 41.1.2
- OMPD **rc** Type, see Section 39.9

41.1.4 ompd_finalize Routine

Name: ompd_finalize Category: function	Properties: C-only, OMPD

Return Type

Name	Type	Properties
<return type>	rc	*default*

Prototypes

C

```
ompd_rc_t ompd_finalize(void);
```

C

Semantics

When the tool is finished with the OMPD library, it should call **ompd_finalize** before it unloads the library. The call to the **ompd_finalize** routine must be the last OMPD call that the tool makes before it unloads the library. This routine allows the OMPD library to free any resources that it may be holding. The OMPD library may implement a *finalizer* section, which executes as the library is unloaded and therefore after the call to **ompd_finalize**. During finalization, the OMPD library may use the callbacks that the tool provided earlier during the call to **ompd_initialize**. In addition to the return codes permitted for all OMPD routines, this routine returns **ompd_rc_unsupported** if the OMPD library is not initialized.

Cross References

- OMPD **rc** Type, see Section 39.9

41.2 Process Initialization and Finalization

41.2.1 ompd_process_initialize Routine

Name: ompd_process_initialize Category: function	Properties: C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
context	address_space_context	opaque, pointer
host_handle	address_space_handle	opaque, pointer-to-pointer

Prototypes

C

```
ompd_rc_t ompd_process_initialize(
  ompd_address_space_context_t *context,
  ompd_address_space_handle_t **host_handle);
```

C

Semantics

A tool calls **ompd_process_initialize** to obtain an address space handle for the host device when it initializes a session on an OpenMP process or core file. On return from **ompd_process_initialize**, the tool owns the address space handle, which it must release with **ompd_rel_address_space_handle**. The initialization function must be called before any OMPD operations are performed on the OpenMP process or core file. This routine allows the OMPD library to confirm that it can handle the OpenMP process or core file that *context* identifies.

The *context* argument is an opaque handle that the tool provides to address an address space from the host device. On return, the *host_handle* argument provides an opaque handle to the tool for this address space, which the tool must release when it is no longer needed.

In addition to the return codes permitted for all OMPD routines, this routine returns **ompd_rc_incompatible** if the OMPD library is incompatible with the runtime library loaded in the process.

Cross References

- OMPD **address_space_context** Type, see Section 39.3
- OMPD **address_space_handle** Type, see Section 39.18.1
- **ompd_rel_address_space_handle** Routine, see Section 41.8.1
- OMPD **rc** Type, see Section 39.9

41.2.2 ompd_device_initialize Routine

Name: **ompd_device_initialize** Category: function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
host_handle	address_space_handle	opaque, pointer
device_context	address_space_context	opaque, pointer
kind	device	*default*
sizeof_id	size	pointer
id	void	pointer
device_handle	address_space_handle	opaque, pointer-to-pointer

Prototypes

C

```c
ompd_rc_t ompd_device_initialize(
  ompd_address_space_handle_t *host_handle,
  ompd_address_space_context_t *device_context,
  ompd_device_t kind, ompd_size_t *sizeof_id, void *id,
  ompd_address_space_handle_t **device_handle);
```

C

Semantics

A tool calls **ompd_device_initialize** to obtain an address space handle for a non-host device that has at least one active **target** region. On return from **ompd_device_initialize**, the tool owns the address space handle. The *host_handle* argument is an opaque handle that the tool provides to reference the host device address space associated with an OpenMP process or core file. The *device_context* argument is an opaque handle that the tool provides to reference a non-host device address space. The *kind*, *sizeof_id*, and *id* arguments represent a device identifier. On return the *device_handle* argument provides an opaque handle to the tool for this address space.

In addition to the return codes permitted for all OMPD routines, this routine may return **ompd_rc_unsupported** if the OMPD library has no support for the specific device.

Cross References

- OMPD **address_space_context** Type, see Section 39.3
- OMPD **address_space_handle** Type, see Section 39.18.1
- OMPD **device** Type, see Section 39.5
- OMPD **rc** Type, see Section 39.9
- OMPD **size** Type, see Section 39.12

41.2.3 ompd_get_device_thread_id_kinds Routine

Name: **ompd_get_device_thread_id_kinds** **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
device_handle	address_space_handle	opaque, pointer
kinds	thread_id	pointer-to-pointer
thread_id_sizes	size	pointer-to-pointer
count	integer	pointer

Prototypes

C

```
ompd_rc_t ompd_get_device_thread_id_kinds(
  ompd_address_space_handle_t *device_handle,
  ompd_thread_id_t **kinds, ompd_size_t **thread_id_sizes,
  int *count);
```

C

Semantics

The **ompd_get_device_thread_id_kinds** routine returns an array of supported native thread identifier kinds and a corresponding array of their respective sizes for a given device. The OMPD library allocates storage for the arrays with the memory allocation callback that the tool provides. Each supported native thread identifier kind is guaranteed to be recognizable by the OMPD library and may be mapped to and from any OpenMP thread that executes on the device. The third-party tool owns the storage for the array of kinds and the array of sizes that is returned via the *kinds* and *thread_id_sizes* arguments, and it is responsible for freeing that storage.

The *device_handle* argument is a pointer to an opaque address space handle that represents a host device (returned by **ompd_process_initialize**) or a non-host device (returned by **ompd_device_initialize**). On return, the *kinds* argument is the address of a pointer to an array of native thread identifier kinds, the *thread_id_sizes* argument is the address of a pointer to an array of the corresponding native thread identifier sizes used by the OMPD library, and the *count* argument is the address of a variable that indicates the sizes of the returned arrays.

Cross References

- OMPD **address_space_handle** Type, see Section 39.18.1
- **ompd_device_initialize** Routine, see Section 41.2.2
- **ompd_process_initialize** Routine, see Section 41.2.1
- OMPD **rc** Type, see Section 39.9

- OMPD **size** Type, see Section 39.12
- OMPD **thread_id** Type, see Section 39.15

41.3 Address Space Information

41.3.1 ompd_get_omp_version Routine

Name: ompd_get_omp_version **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
address_space	address_space_handle	opaque, pointer
omp_version	word	pointer

Prototypes

C

```
ompd_rc_t ompd_get_omp_version(
  ompd_address_space_handle_t *address_space,
  ompd_word_t *omp_version);
```

C

Semantics

The tool may call the **ompd_get_omp_version** routine to obtain the version of the OpenMP API that is associated with the address space *address_space*. The *address_space* argument is an opaque handle that the tool provides to reference the address space of the process or device. Upon return, the *omp_version* argument contains the version of the OpenMP runtime in the **_OPENMP** version macro format.

Cross References

- OMPD **address_space_handle** Type, see Section 39.18.1
- OMPD **rc** Type, see Section 39.9
- OMPD **word** Type, see Section 39.17

41.3.2 ompd_get_omp_version_string Routine

Name: ompd_get_omp_version_string **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
address_space	address_space_handle	opaque, pointer
string	const char	intent(out), pointer-to-pointer

Prototypes

C

```
ompd_rc_t ompd_get_omp_version_string(
  ompd_address_space_handle_t *address_space, const char **string);
```

C

Semantics

The **ompd_get_omp_version_string** routine returns a descriptive string for the OpenMP API version that is associated with an address space. The *address_space* argument is an opaque handle that the tool provides to reference the address space of a process or device. A pointer to a descriptive version string is placed into the location to which the *string* output argument points. After returning from the routine, the tool owns the string. The OMPD library must use the memory allocation callback that the tool provides to allocate the string storage. The tool is responsible for releasing the memory.

Cross References

- OMPD Handle Types, see Section 39.18
- OMPD **rc** Type, see Section 39.9

41.4 Thread Handle Routines

41.4.1 ompd_get_thread_in_parallel Routine

Name: ompd_get_thread_in_parallel **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
parallel_handle	parallel_handle	opaque, pointer
thread_num	integer	*default*
thread_handle	thread_handle	opaque, pointer-to-pointer

Prototypes

C

```
ompd_rc_t ompd_get_thread_in_parallel(
  ompd_parallel_handle_t *parallel_handle, int thread_num,
  ompd_thread_handle_t **thread_handle);
```

C

Semantics

The **ompd_get_thread_in_parallel** routine enables a tool to obtain handles for OpenMP threads that are associated with a parallel region. A successful invocation of **ompd_get_thread_in_parallel** returns a pointer to a native thread handle in the location to which *thread_handle* points. This routine yields meaningful results only if all OpenMP threads in the team that is executing the parallel region are stopped.

The *parallel_handle* argument is an opaque handle for a parallel region and selects the parallel region on which to operate. The *thread_num* argument represents the thread number and selects the thread, the handle for which is to be returned. On return, the *thread_handle* argument is a handle for the selected thread.

This routine returns **ompd_rc_bad_input** if the *thread_num* argument is greater than or equal to the *team-size-var* ICV or negative, in which case the value returned in *thread_handle* is invalid.

Cross References

- **ompd_get_icv_from_scope** Routine, see Section 41.11.2
- OMPD **parallel_handle** Type, see Section 39.18.2
- OMPD **rc** Type, see Section 39.9
- OMPD **thread_handle** Type, see Section 39.18.4

41.4.2 ompd_get_thread_handle Routine

Name: ompd_get_thread_handle **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
handle	address_space_handle	pointer
kind	thread_id	*default*
sizeof_thread_id	size	*default*
thread_id	void	intent(in), pointer
thread_handle	thread_handle	pointer-to-pointer

Prototypes

C

```
ompd_rc_t ompd_get_thread_handle(
  ompd_address_space_handle_t *handle, ompd_thread_id_t kind,
  ompd_size_t sizeof_thread_id, const void *thread_id,
  ompd_thread_handle_t **thread_handle);
```

C

Semantics

The **ompd_get_thread_handle** routine maps a native thread to a native thread handle. Further, the routine determines if the native thread identifier to which *thread_id* points represents an OpenMP thread. If so, the routine returns **ompd_rc_ok** and the location to which *thread_handle* points is set to the native thread handle for the native thread to which the OpenMP thread is mapped.

The *handle* argument is a handle that the tool provides to reference an address space. The *kind*, *sizeof_thread_id*, and *thread_id* arguments represent a native thread identifier. On return, the *thread_handle* argument provides a handle to the native thread within the provided address space.

The native thread identifier to which *thread_id* points must be valid for the duration of the call to the routine. If the OMPD library must retain the native thread identifier, it must copy it.

This routine returns **ompd_rc_bad_input** if a different value in *sizeof_thread_id* is expected for a thread kind of *kind*. In addition to the return codes permitted for all OMPD routines, this routine returns **ompd_rc_unsupported** if the *kind* of thread is not supported and it returns **ompd_rc_unavailable** if the native thread is not an OpenMP thread.

Cross References

- OMPD **address_space_handle** Type, see Section 39.18.1
- OMPD **rc** Type, see Section 39.9
- OMPD **size** Type, see Section 39.12
- OMPD **thread_handle** Type, see Section 39.18.4
- OMPD **thread_id** Type, see Section 39.15

41.4.3 ompd_get_thread_id Routine

Name: ompd_get_thread_id **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
thread_handle	thread_handle	pointer
kind	thread_id	*default*
sizeof_thread_id	size	*default*
thread_id	void	pointer

Prototypes

C

```
ompd_rc_t ompd_get_thread_id(ompd_thread_handle_t *thread_handle,
  ompd_thread_id_t kind, ompd_size_t sizeof_thread_id,
  void *thread_id);
```

C

Semantics

The **ompd_get_thread_id** routine maps a native thread handle to a native thread identifier. This routine yields meaningful results only if the referenced OpenMP thread is stopped. The *thread_handle* argument is a native thread handle. The *kind* argument represents the native thread identifier. The *sizeof_thread_id* argument represents the size of the native thread identifier. On return, the *thread_id* argument is a buffer that represents a native thread identifier.

This routine returns **ompd_rc_bad_input** if a different value in *sizeof_thread_id* is expected for a native thread kind of *kind*. In addition to the return codes permitted for all OMPD routines, this routine returns **ompd_rc_unsupported** if the *kind* of native thread is not supported.

Cross References

- OMPD **rc** Type, see Section 39.9
- OMPD **size** Type, see Section 39.12
- OMPD **thread_handle** Type, see Section 39.18.4
- OMPD **thread_id** Type, see Section 39.15

41.4.4 ompd_get_device_from_thread Routine

Name: ompd_get_device_from_thread **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
thread_handle	thread_handle	pointer
device	address_space_handle	pointer-to-pointer

Prototypes

C

```
ompd_rc_t ompd_get_device_from_thread(
  ompd_thread_handle_t *thread_handle,
  ompd_address_space_handle_t **device);
```

C

Semantics

The **ompd_get_device_from_thread** routine obtains a pointer to the address space handle for a device on which an OpenMP thread is executing. The returned pointer will be the same as the address space handle pointer that was previously returned by a call to **ompd_process_initialize** (for a host device) or a call to **ompd_device_initialize** (for a non-host device). This routine yields meaningful results only if the referenced OpenMP thread is stopped.

The *thread_handle* argument is a pointer to a native thread handle that represents a native thread to which an OpenMP thread is mapped. On return, the *device* argument is the address of a pointer to an address space handle.

Cross References

- OMPD **address_space_handle** Type, see Section 39.18.1
- OMPD **rc** Type, see Section 39.9
- OMPD **thread_handle** Type, see Section 39.18.4

41.5 Parallel Region Handle Routines

41.5.1 ompd_get_curr_parallel_handle Routine

Name: ompd_get_curr_parallel_handle **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
thread_handle	thread_handle	opaque, pointer
parallel_handle	parallel_handle	opaque, pointer-to-pointer

Prototypes

C

```
ompd_rc_t ompd_get_curr_parallel_handle(
  ompd_thread_handle_t *thread_handle,
  ompd_parallel_handle_t **parallel_handle);
```

C

Semantics

The **ompd_get_curr_parallel_handle** routine enables a tool to obtain a pointer to the parallel handle for the innermost parallel region that is associated with an OpenMP thread. This routine yields meaningful results only if the referenced OpenMP thread is stopped. The parallel handle is owned by the tool and it must be released by calling **ompd_rel_parallel_handle**.

The *thread_handle* argument is an opaque handle for a thread and selects the thread on which to operate. On return, the *parallel_handle* argument is set to a handle for the parallel region that the associated thread is currently executing, if any.

In addition to the return codes permitted for all OMPD routines, this routine returns **ompd_rc_unavailable** if the thread is not currently part of a team.

Cross References

- **ompd_rel_parallel_handle** Routine, see Section 41.8.2
- OMPD **parallel_handle** Type, see Section 39.18.2
- OMPD **rc** Type, see Section 39.9
- OMPD **thread_handle** Type, see Section 39.18.4

41.5.2 ompd_get_enclosing_parallel_handle Routine

Name: **ompd_get_enclosing_parallel_handle** **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
parallel_handle	parallel_handle	opaque, pointer
enclosing_parallel_handle	parallel_handle	opaque, pointer-to-pointer

Prototypes

C

```
ompd_rc_t ompd_get_enclosing_parallel_handle(
  ompd_parallel_handle_t *parallel_handle,
  ompd_parallel_handle_t **enclosing_parallel_handle);
```

C

Semantics

The **ompd_get_enclosing_parallel_handle** routine enables a tool to obtain a pointer to the parallel handle for the parallel region that encloses the parallel region that *parallel_handle* specifies. This routine yields meaningful results only if at least one thread in the team that is executing the parallel region is stopped. A pointer to the parallel handle for the enclosing region is returned in the location to which *enclosing_parallel_handle* points. After a call to this routine, the tool owns the handle; the tool must release the handle with **ompd_rel_parallel_handle** when it is no longer required. The *parallel_handle* argument is an opaque handle for a parallel region that selects the parallel region on which to operate.

In addition to the return codes permitted for all OMPD routines, this routine returns **ompd_rc_unavailable** if no enclosing parallel region exists.

Cross References

- **ompd_rel_parallel_handle** Routine, see Section 41.8.2
- OMPD **parallel_handle** Type, see Section 39.18.2
- OMPD **rc** Type, see Section 39.9

41.5.3 ompd_get_task_parallel_handle Routine

Name: ompd_get_task_parallel_handle **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
task_handle	task_handle	pointer
task_parallel_handle	parallel_handle	pointer-to-pointer

Prototypes

C

```
ompd_rc_t ompd_get_task_parallel_handle(
  ompd_task_handle_t *task_handle,
  ompd_parallel_handle_t **task_parallel_handle);
```

C

Semantics

The **ompd_get_task_parallel_handle** routine enables a tool to obtain a pointer to the parallel handle for the parallel region that encloses the task region that *task_handle* specifies. This routine yields meaningful results only if at least one thread in the team that is executing the parallel region is stopped. A pointer to the parallel handle is returned in the location to which *task_parallel_handle* points. The tool owns that parallel handle, which it must release with **ompd_rel_parallel_handle**.

Cross References

- **ompd_rel_parallel_handle** Routine, see Section 41.8.2
- OMPD **parallel_handle** Type, see Section 39.18.2
- OMPD **rc** Type, see Section 39.9
- OMPD **task_handle** Type, see Section 39.18.3

41.6 Task Handle Routines

41.6.1 ompd_get_curr_task_handle Routine

Name: **ompd_get_curr_task_handle** Category: function	Properties: C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
thread_handle	thread_handle	opaque, pointer
task_handle	task_handle	opaque, pointer-to-pointer

Prototypes

C

```c
ompd_rc_t ompd_get_curr_task_handle(
  ompd_thread_handle_t *thread_handle,
  ompd_task_handle_t **task_handle);
```

C

Semantics

The **ompd_get_curr_task_handle** routine obtains a pointer to the task handle for the current task region that is associated with an OpenMP thread. This routine yields meaningful results only if the thread for which the handle is provided is stopped. The task handle must be released with **ompd_rel_task_handle**. The *thread_handle* argument is an opaque handle that selects the thread on which to operate. On return, the *task_handle* argument points to a location that points to a handle for the task that the thread is currently executing. In addition to the return codes permitted for all OMPD routines, this routine returns **ompd_rc_unavailable** if the thread is currently not executing a task.

Cross References

- **ompd_rel_task_handle** Routine, see Section 41.8.3
- OMPD **rc** Type, see Section 39.9
- OMPD **task_handle** Type, see Section 39.18.3
- OMPD **thread_handle** Type, see Section 39.18.4

41.6.2 ompd_get_generating_task_handle Routine

Name: ompd_get_generating_task_handle Category: function	Properties: C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
task_handle	task_handle	pointer
generating_task_handle	task_handle	pointer-to-pointer

Prototypes

C

```
ompd_rc_t ompd_get_generating_task_handle(
  ompd_task_handle_t *task_handle,
  ompd_task_handle_t **generating_task_handle);
```

C

Semantics

The **ompd_get_generating_task_handle** routine obtains a pointer to the task handle of the generating task region. The generating task is the task that was active when the task specified by *task_handle* was created. This routine yields meaningful results only if the thread that is executing the task that *task_handle* specifies is stopped while executing the task. The generating task handle must be released with **ompd_rel_task_handle**. On return, the *generating_task_handle* argument points to a location that points to a handle for the generating task. In addition to the return codes permitted for all OMPD routines, this routine returns **ompd_rc_unavailable** if no generating task region exists.

Cross References

- **ompd_rel_task_handle** Routine, see Section 41.8.3
- OMPD **rc** Type, see Section 39.9
- OMPD **task_handle** Type, see Section 39.18.3

41.6.3 `ompd_get_scheduling_task_handle` Routine

Name: `ompd_get_scheduling_task_handle` Category: function	Properties: C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
task_handle	task_handle	pointer
scheduling_task_handle	task_handle	pointer-to-pointer

Prototypes

C

```c
ompd_rc_t ompd_get_scheduling_task_handle(
  ompd_task_handle_t *task_handle,
  ompd_task_handle_t **scheduling_task_handle);
```

C

Semantics

The **`ompd_get_scheduling_task_handle`** routine obtains a task handle for the task that was active when the task that *task_handle* represents was scheduled. An implicit task does not have a scheduling task. This routine yields meaningful results only if the thread that is executing the task that *task_handle* specifies is stopped while executing the task. On return, the *scheduling_task_handle* argument points to a location that points to a handle for the task that is still on the stack of execution on the same thread and was deferred in favor of executing the selected task. This task handle must be released with **`ompd_rel_task_handle`**. In addition to the return codes permitted for all OMPD routines, this routine returns **`ompd_rc_unavailable`** if no scheduling task exists.

Cross References

- **`ompd_rel_task_handle`** Routine, see Section 41.8.3
- OMPD **`rc`** Type, see Section 39.9
- OMPD **`task_handle`** Type, see Section 39.18.3

41.6.4 `ompd_get_task_in_parallel` Routine

Name: `ompd_get_task_in_parallel` Category: function	Properties: C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
parallel_handle	parallel_handle	opaque, pointer
thread_num	integer	*default*
task_handle	task_handle	opaque, pointer-to-pointer

Prototypes

C

```c
ompd_rc_t ompd_get_task_in_parallel(
  ompd_parallel_handle_t *parallel_handle, int thread_num,
  ompd_task_handle_t **task_handle);
```

C

Semantics

The **ompd_get_task_in_parallel** routine obtains handles for the implicit tasks that are associated with a parallel region. A successful invocation of **ompd_get_task_in_parallel** returns a pointer to a task handle in the location to which *task_handle* points. This routine yields meaningful results only if all OpenMP threads in the parallel region are stopped. The *parallel_handle* argument is an opaque handle that selects the parallel region on which to operate. The *thread_num* argument selects the implicit task of the team to be returned. The *thread_num* argument is equal to the *thread-num-var* ICV value of the selected implicit task. This routine returns **ompd_rc_bad_input** if the *thread_num* argument is greater than or equal to the *team-size-var* ICV or negative.

Cross References

- **ompd_get_icv_from_scope** Routine, see Section 41.11.2
- OMPD **parallel_handle** Type, see Section 39.18.2
- OMPD **rc** Type, see Section 39.9
- OMPD **task_handle** Type, see Section 39.18.3

41.6.5 ompd_get_task_function Routine

Name: **ompd_get_task_function** Category: function	Properties: C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
task_handle	task_handle	opaque, pointer
entry_point	address	pointer

Prototypes

C

```c
ompd_rc_t ompd_get_task_function(ompd_task_handle_t *task_handle,
  ompd_address_t *entry_point);
```

C

Semantics

The **ompd_get_task_function** routine returns the entry point of the code that corresponds to the body of code that the task executes. This routine returns meaningful results only if the thread that is executing the task that *task_handle* specifies is stopped while executing the task. That argument is an opaque handle that selects the task on which to operate. On return, the *entry_point* argument is set to an address that describes the beginning of application code that executes the task region.

Cross References

- OMPD **address** Type, see Section 39.2
- OMPD **rc** Type, see Section 39.9
- OMPD **task_handle** Type, see Section 39.18.3

41.6.6 ompd_get_task_frame Routine

Name: ompd_get_task_frame **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<*return type*>	rc	*default*
task_handle	task_handle	pointer
exit_frame	frame_info	pointer
enter_frame	frame_info	pointer

Prototypes

C

```c
ompd_rc_t ompd_get_task_frame(ompd_task_handle_t *task_handle,
  ompd_frame_info_t *exit_frame, ompd_frame_info_t *enter_frame);
```

C

Semantics

The **ompd_get_task_frame** routine extracts the frame pointers of a task. An OpenMP implementation maintains an object of **frame** OMPT type for every implicit task and explicit task. The **ompd_get_task_frame** routine extracts the **enter_frame** and **exit_frame** fields of the **frame** object of the task that *task_handle* identifies. This routine yields meaningful results only if the thread that is executing the task that *task_handle* specifies is stopped while executing the task.

On return, the *exit_frame* argument points to a **frame_info** object that has the frame information with the same semantics as the **exit_frame** field in the **frame** object that is associated with the specified task. On return, the *enter_frame* argument points to a **frame_info** object that has the frame information with the same semantics as the **enter_frame** field in the **frame** object that is associated with the specified task.

Cross References

- OMPD **address** Type, see Section 39.2
- OMPT **frame** Type, see Section 33.15
- OMPD **frame_info** Type, see Section 39.7
- OMPD **rc** Type, see Section 39.9
- OMPD **task_handle** Type, see Section 39.18.3

41.7 Handle Comparing Routines

This section describes handle-comparing routines, which are routines that have the handle-comparing property and, thus, enable the comparison of two handles. The internal structure of handles is opaque to tools. While tools can easily compare pointers to handles, they cannot determine whether handles at two different addresses refer to the same underlying context and instead must use a handle-comparing routine.

On success, a handle-comparing routine returns, in the location to which its *cmp_value* argument points, a signed integer value that indicates how the underlying contexts compare. A value less than, equal to, or greater than 0 indicates that the context to which *<handle-type>_handle_1* corresponds is, respectively, less than, equal to, or greater than that to which *<handle-type>_handle_2* corresponds. The *<handle-type>_handle_1* and *<handle-type>_handle_2* arguments are handles that correspond to the type of handle that the routine compares. In each handle-comparing routine, *<handle-type>* is replaced with the name of the type of handle that the routine compares. For all types of handles, the means by which two handles are ordered is implementation defined.

41.7.1 ompd_parallel_handle_compare Routine

Name: ompd_parallel_handle_compare **Category:** function	**Properties:** C-only, handle-comparing, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
parallel_handle_1	parallel_handle	opaque, pointer
parallel_handle_2	parallel_handle	opaque, pointer
cmp_value	integer	pointer

Prototypes

C

```
ompd_rc_t ompd_parallel_handle_compare(
  ompd_parallel_handle_t *parallel_handle_1,
  ompd_parallel_handle_t *parallel_handle_2, int *cmp_value);
```

C

Semantics

The **ompd_parallel_handle_compare** routine compares two parallel handles. The *parallel_handle_1* and *parallel_handle_2* arguments are parallel handles that correspond to parallel regions.

Cross References

- OMPD **parallel_handle** Type, see Section 39.18.2
- OMPD **rc** Type, see Section 39.9

41.7.2 ompd_task_handle_compare Routine

Name: ompd_task_handle_compare Category: function	Properties: C-only, handle-comparing, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
task_handle_1	task_handle	opaque, pointer
task_handle_2	task_handle	opaque, pointer
cmp_value	integer	pointer

Prototypes

C

```
ompd_rc_t ompd_task_handle_compare(
  ompd_task_handle_t *task_handle_1,
  ompd_task_handle_t *task_handle_2, int *cmp_value);
```

C

Semantics

The **ompd_task_handle_compare** routine compares two task handles. The *task_handle_1* and *task_handle_2* arguments are task handles that correspond to tasks.

Cross References

- OMPD **rc** Type, see Section 39.9
- OMPD **task_handle** Type, see Section 39.18.3

41.7.3 `ompd_thread_handle_compare` Routine

Name: `ompd_thread_handle_compare`	Properties: C-only, handle-comparing,
Category: function	OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
thread_handle_1	thread_handle	opaque, pointer
thread_handle_2	thread_handle	opaque, pointer
cmp_value	integer	pointer

Prototypes

C

```
ompd_rc_t ompd_thread_handle_compare(
  ompd_thread_handle_t *thread_handle_1,
  ompd_thread_handle_t *thread_handle_2, int *cmp_value);
```

C

Semantics

The **`ompd_thread_handle_compare`** routine compares two native thread handles. The *thread_handle_1* and *thread_handle_2* arguments are native thread handles that correspond to native threads.

Cross References

- OMPD **`rc`** Type, see Section 39.9
- OMPD **`thread_handle`** Type, see Section 39.18.4

41.8 Handle Releasing Routines

This section describes handle-releasing routines, which are routines that have the handle-releasing property and, thus, release a handle owned by a tool. When a tool finishes with a handle that a *handle* argument identifies, it should release it with the corresponding handle-releasing routine so the OMPD library can release any resources that it has related to the corresponding context.

Restrictions

Restrictions to handle-releasing routines are as follows:

- A context must not be used after its corresponding handle is released.

41.8.1 `ompd_rel_address_space_handle` Routine

Name: `ompd_rel_address_space_handle`	Properties: C-only, handle-releasing,
Category: function	OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
handle	address_space_handle	opaque, pointer

Prototypes

C

```
ompd_rc_t ompd_rel_address_space_handle(
  ompd_address_space_handle_t *handle);
```

C

Semantics

A tool calls **ompd_rel_address_space_handle** to release an address space handle.

Cross References

- OMPD **address_space_handle** Type, see Section 39.18.1
- OMPD **rc** Type, see Section 39.9

41.8.2 ompd_rel_parallel_handle Routine

Name: **ompd_rel_parallel_handle** **Category:** function	**Properties:** C-only, handle-releasing, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
parallel_handle	parallel_handle	opaque, pointer

Prototypes

C

```
ompd_rc_t ompd_rel_parallel_handle(
  ompd_parallel_handle_t *parallel_handle);
```

C

Semantics

A tool calls **ompd_rel_parallel_handle** to release a parallel handle.

Cross References

- OMPD **parallel_handle** Type, see Section 39.18.2
- OMPD **rc** Type, see Section 39.9

41.8.3 ompd_rel_task_handle Routine

Name: **ompd_rel_task_handle** **Category:** function	**Properties:** C-only, handle-releasing, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
task_handle	task_handle	opaque, pointer

Prototypes

C

```
ompd_rc_t ompd_rel_task_handle(ompd_task_handle_t *task_handle);
```

C

Semantics

A tool calls **ompd_rel_task_handle** to release a task handle.

Cross References

- OMPD **rc** Type, see Section 39.9
- OMPD **task_handle** Type, see Section 39.18.3

41.8.4 ompd_rel_thread_handle Routine

Name: ompd_rel_thread_handle **Category:** function	**Properties:** C-only, handle-releasing, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
thread_handle	thread_handle	opaque, pointer

Prototypes

C

```
ompd_rc_t ompd_rel_thread_handle(
  ompd_thread_handle_t *thread_handle);
```

C

Semantics

A tool calls **ompd_rel_thread_handle** to release a native thread handle.

Cross References

- OMPD **rc** Type, see Section 39.9
- OMPD **thread_handle** Type, see Section 39.18.4

41.9 Querying Thread States

41.9.1 ompd_enumerate_states Routine

Name: ompd_enumerate_states **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
address_space_handle	address_space_handle	opaque, pointer
current_state	word	*default*
next_state	word	pointer
next_state_name	const char	intent(out), pointer-to-pointer
more_enums	word	pointer

Prototypes

C

```
ompd_rc_t ompd_enumerate_states(
  ompd_address_space_handle_t *address_space_handle,
  ompd_word_t current_state, ompd_word_t *next_state,
  const char **next_state_name, ompd_word_t *more_enums);
```

C

Semantics

An OpenMP implementation may support only a subset of the states that the **state** OMPT type defines. In addition, an OpenMP implementation may support implementation defined states. The **ompd_enumerate_states** routine enumerates the thread states that an OpenMP implementation supports.

When the *current_state* argument is a thread state that an OpenMP implementation supports, the routine assigns the value and string name of the next thread state in the enumeration to the locations to which the *next_state* and *next_state_name* arguments point. On return, the tool owns the *next_state_name* string. The OMPD library allocates storage for the string with the **alloc_memory** callback that the tool provides. The tool is responsible for releasing the storage. On return, the location to which the *more_enums* argument points has the value 1 whenever one or more states are left in the enumeration. On return, the location to which the *more_enums* argument points has the value 0 when *current_state* is the last state in the enumeration.

The *address_space_handle* argument identifies the address space. The *current_state* argument must be a thread state that the OpenMP implementation supports. To begin enumerating the supported states, a tool should pass **ompt_state_undefined** as the value of *current_state*. Subsequent calls to **ompd_enumerate_states** by the tool should pass the value that the routine returned in the *next_state* argument. This routine returns **ompd_rc_bad_input** if an unknown value is provided in *current_state*.

Cross References

- OMPD **address_space_handle** Type, see Section 39.18.1
- OMPD **rc** Type, see Section 39.9
- OMPT **state** Type, see Section 33.31
- OMPD **word** Type, see Section 39.17

41.9.2 ompd_get_state Routine

Name: **ompd_get_state** **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
thread_handle	thread_handle	opaque, pointer
state	word	pointer
wait_id	wait_id	pointer

Prototypes

C

```
ompd_rc_t ompd_get_state(ompd_thread_handle_t *thread_handle,
  ompd_word_t *state, ompd_wait_id_t *wait_id);
```

C

Semantics

The **ompd_get_state** routine returns the state of an OpenMP thread. This routine yields meaningful results only if the referenced thread is stopped. The *thread_handle* argument identifies the thread. The *state* argument represents the state of that thread as represented by a value that **ompd_enumerate_states** returns. On return, if the *wait_id* argument is a non-null value then it points to a handle that corresponds to the *wait_id* wait identifier of the thread. If the thread state is not one of the specified wait states, the value to which *wait_id* points is undefined.

Cross References

- **ompd_enumerate_states** Routine, see Section 41.9.1
- OMPD **rc** Type, see Section 39.9
- OMPD **thread_handle** Type, see Section 39.18.4
- OMPD **wait_id** Type, see Section 39.16
- OMPD **word** Type, see Section 39.17

41.10 Display Control Variables

41.10.1 ompd_get_display_control_vars Routine

Name: ompd_get_display_control_vars Category: function	Properties: C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
address_space_handle	address_space_handle	opaque, pointer
control_vars	const char * const *	intent(out), pointer

Prototypes

C

```
ompd_rc_t ompd_get_display_control_vars(
  ompd_address_space_handle_t *address_space_handle,
  const char * const * *control_vars);
```

C

Semantics

The **ompd_get_display_control_vars** routine returns a list of OpenMP control variables as a NULL-terminated vector of null-terminated strings of name/value pairs. These control variables have user-controllable settings and are important to the operation or performance of an OpenMP runtime system. The control variables that this interface exposes include all OpenMP environment variables, settings that may come from vendor or platform-specific environment variables, and other settings that affect the operation or functioning of an OpenMP runtime. The format of the strings is *NAME* '=' *VALUE*. *NAME* corresponds to the control variable name, optionally prepended with a bracketed *DEVICE*. *VALUE* corresponds to the value of the control variable.

On return, the tool owns the vector and the strings. The OMPD library must satisfy the termination constraints; it may use static or dynamic memory for the vector and/or the strings and is unconstrained in how it arranges them in memory. If it uses dynamic memory then the OMPD library must use the **alloc_memory** callback that the tool provides. The tool must use the **ompd_rel_display_control_vars** routine to release the vector and the strings.

The *address_space_handle* argument identifies the address space. On return, the *control_vars* argument points to the vector of display control variables.

Cross References

- OMPD **address_space_handle** Type, see Section 39.18.1
- **ompd_initialize** Routine, see Section 41.1.1
- **ompd_rel_display_control_vars** Routine, see Section 41.10.2
- OMPD **rc** Type, see Section 39.9

41.10.2 ompd_rel_display_control_vars Routine

Name: **ompd_rel_display_control_vars** **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
control_vars	const char * const *	pointer

Prototypes

C

```c
ompd_rc_t ompd_rel_display_control_vars(
  const char * const * *control_vars);
```

C

Semantics

After a tool calls **ompd_get_display_control_vars**, it owns the vector and strings that it acquires. The tool must call **ompd_rel_display_control_vars** to release them. The *control_vars* argument is the vector of display control variables to be released.

Cross References

- **ompd_get_display_control_vars** Routine, see Section 41.10.1
- OMPD **rc** Type, see Section 39.9

41.11 Accessing Scope-Specific Information

41.11.1 ompd_enumerate_icvs Routine

Name: **ompd_enumerate_icvs** **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
handle	address_space_handle	opaque, pointer
current	icv_id	*default*
next_id	icv_id	pointer
next_icv_name	const char	intent(out), pointer-to-pointer
next_scope	scope	pointer
more	integer	pointer

Prototypes

C

```
ompd_rc_t ompd_enumerate_icvs(
  ompd_address_space_handle_t *handle, ompd_icv_id_t current,
  ompd_icv_id_t *next_id, const char **next_icv_name,
  ompd_scope_t *next_scope, int *more);
```

C

Semantics

The **ompd_enumerate_icvs** routine enables a tool to enumerate the ICVs that an OpenMP implementation supports and their related scopes. An OpenMP implementation must support all ICVs listed in Section 3.1. An OpenMP implementation may support additional implementation defined ICVs. An implementation may store ICVs in a different scope than Section 3.1 indicates.

When the *current* argument is set to the identifier of a supported ICV, **ompd_enumerate_icvs** assigns the value, string name, and scope of the next ICV in the enumeration to the locations to which the *next_id*, *next_icv_name*, and *next_scope* arguments point. On return, the tool owns the *next_icv_name* string. The OMPD library uses the **alloc_memory** callback that the tool provides to allocate the string storage; the tool is responsible for releasing the memory.

On return, the location to which the *more* argument points has the value of 1 whenever one or more ICVs are left in the enumeration. On return, that location has the value 0 when *current* is the last ICV in the enumeration. The *address_space_handle* argument identifies the address space. The *current* argument must be an ICV that the OpenMP implementation supports. To begin enumerating the ICVs, a tool should pass **ompd_icv_undefined** as the value of *current*. Subsequent calls to **ompd_enumerate_icvs** should pass the value returned by the routine in the *next_id* output argument. On return, the *next_id* argument points to an integer with the value of the ID of the next ICV in the enumeration. On return, the *next_icv_name* argument points to a character string with the name of the next ICV. On return, the value to which the *next_scope* argument points identifies the scope of the next ICV. On return, the *more_enums* argument points to an integer with the value of 1 when more ICVs are left to enumerate and the value of 0 when no more ICVs are left. This routine returns **ompd_rc_bad_input** if an unknown value is provided in *current*.

Cross References

- OMPD **address_space_handle** Type, see Section 39.18.1
- OMPD **icv_id** Type, see Section 39.8
- OMPD **rc** Type, see Section 39.9
- OMPD **scope** Type, see Section 39.11

41.11.2 ompd_get_icv_from_scope Routine

Name: **ompd_get_icv_from_scope** **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
handle	void	opaque, pointer
scope	scope	*default*
icv_id	icv_id	*default*
icv_value	word	pointer

Prototypes

C

```
ompd_rc_t ompd_get_icv_from_scope(void *handle,
  ompd_scope_t scope, ompd_icv_id_t icv_id, ompd_word_t *icv_value);
```

C

Summary

The **ompd_get_icv_from_scope** routine returns the value of an ICV. The *handle* argument provides an OpenMP scope handle. The *scope* argument specifies the kind of scope provided in *handle*. The *icv_id* argument specifies the ID of the requested ICV. On return, the *icv_value* argument points to a location with the value of the requested ICV.

This routine returns **ompd_rc_bad_input** if an unknown value is provided in *icv_id*. In addition to the return codes permitted for all OMPD routines, this routine returns **ompd_rc_incomplete** if only the first item of the ICV is returned in the integer (e.g., if *nthreads-var* has more than one list item). Further, it returns **ompd_rc_incompatible** if the ICV cannot be represented as an integer or if the scope of the handle matches neither the scope as defined in Section 39.8 nor the scope for *icv_id* as identified by **ompd_enumerate_icvs**.

Cross References

- OMPD Handle Types, see Section 39.18
- OMPD **icv_id** Type, see Section 39.8
- **ompd_enumerate_icvs** Routine, see Section 41.11.1
- OMPD **rc** Type, see Section 39.9
- OMPD **scope** Type, see Section 39.11
- OMPD **word** Type, see Section 39.17

41.11.3 ompd_get_icv_string_from_scope Routine

Name: ompd_get_icv_string_from_scope **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
handle	void	opaque, pointer
scope	scope	*default*
icv_id	icv_id	*default*
icv_string	const char	intent(out), pointer-to-pointer

Prototypes

C

```
ompd_rc_t ompd_get_icv_string_from_scope(void *handle,
  ompd_scope_t scope, ompd_icv_id_t icv_id,
  const char **icv_string);
```

C

Semantics

The **ompd_get_icv_string_from_scope** routine returns the value of an ICV. The *handle* argument provides an OpenMP scope handle. The *scope* argument specifies the kind of scope provided in *handle*. The *icv_id* argument specifies the ID of the requested ICV. On return, the *icv_string* argument points to a string representation of the requested ICV; on return, the tool owns the string. The OMPD library allocates the string storage with the **alloc_memory** callback that the tool provides. The tool is responsible for releasing the memory.

This routine returns **ompd_rc_bad_input** if an unknown value is provided in *icv_id*. In addition to the return codes permitted for all OMPD routines, this routine returns **ompd_rc_incompatible** if the scope of the handle does not match the *scope* as defined in Section 39.8 or if it does not match the scope for *icv_id* as identified by **ompd_enumerate_icvs**.

Cross References

- OMPD Handle Types, see Section 39.18
- OMPD **icv_id** Type, see Section 39.8
- **ompd_enumerate_icvs** Routine, see Section 41.11.1
- OMPD **rc** Type, see Section 39.9
- OMPD **scope** Type, see Section 39.11

41.11.4 `ompd_get_tool_data` Routine

Name: `ompd_get_tool_data` **Category:** function	**Properties:** C-only, OMPD

Return Type and Arguments

Name	Type	Properties
<return type>	rc	*default*
handle	void	opaque, pointer
scope	scope	*default*
value	word	pointer
ptr	address	pointer

Prototypes

C

```
ompd_rc_t ompd_get_tool_data(void *handle, ompd_scope_t scope,
  ompd_word_t *value, ompd_address_t *ptr);
```

C

Semantics

The **`ompd_get_tool_data`** routine provides access to the OMPT tool data stored for each scope. The *handle* argument provides an OpenMP scope handle. The *scope* argument specifies the kind of scope provided in *handle*. On return, the *value* argument points to the **`value`** field of the **`data`** OMPT type stored for the selected scope. On return, the *ptr* argument points to the **`ptr`** field of the **`data`** OMPT type stored for the selected scope. In addition to the return codes permitted for all OMPD routines, this routine returns **`ompd_rc_unsupported`** if the runtime library does not support OMPT.

Cross References

- OMPD **`address`** Type, see Section 39.2
- OMPT **`data`** Type, see Section 33.8
- OMPD Handle Types, see Section 39.18
- OMPD **`rc`** Type, see Section 39.9
- OMPD **`scope`** Type, see Section 39.11
- OMPD **`word`** Type, see Section 39.17

42 OMPD Breakpoint Symbol Names

The OpenMP implementation must define several symbols through which execution must pass when particular events occur and data collection for OMPD is enabled. A third-party tool can enable notification of an event by setting a breakpoint at the address of the symbol.

OMPD symbols have external **C** linkage and do not require demangling or other transformations to look up their names to obtain the address in the OpenMP program. While each OMPD symbol conceptually has a function type signature, it may not be a function. It may be a labeled location.

42.1 `ompd_bp_thread_begin` Breakpoint

Format

C

```c
void ompd_bp_thread_begin(void);
```

C

Semantics

When starting a native thread that will be used as an OpenMP thread, the implementation must execute **`ompd_bp_thread_begin`**. Thus, the OpenMP implementation must execute **`ompd_bp_thread_begin`** at every *native-thread-begin* and *initial-thread-begin* event. This execution occurs before the thread starts the execution of any OpenMP region.

42.2 `ompd_bp_thread_end` Breakpoint

Format

C

```c
void ompd_bp_thread_end(void);
```

C

Semantics

When terminating an OpenMP thread or a native thread that has been used as an OpenMP thread, the implementation must execute **`ompd_bp_thread_end`**. Thus, the OpenMP implementation must execute **`ompd_bp_thread_end`** at every *native-thread-end* and *initial-thread-end* event. This execution occurs after the thread completes the execution of all OpenMP regions. After executing **`ompd_bp_thread_end`**, any *thread_handle* that was acquired for this thread is invalid and should be released by calling **`ompd_rel_thread_handle`**.

Cross References

- **`ompd_rel_thread_handle`** Routine, see Section 41.8.4

42.3 ompd_bp_device_begin Breakpoint

Format

C

```c
void ompd_bp_device_begin(void);
```

C

Semantics

When initializing a device for execution of **target** regions, the implementation must execute **ompd_bp_device_begin**. Thus, the OpenMP implementation must execute **ompd_bp_device_begin** at every *device-initialize* event. This execution occurs before the work associated with any OpenMP region executes on the device.

Cross References

- Device Initialization, see Section 15.4
- **target** Construct, see Section 15.8

42.4 ompd_bp_device_end Breakpoint

Format

C

```c
void ompd_bp_device_end(void);
```

C

Semantics

When terminating use of a device, the implementation must execute **ompd_bp_device_end**. Thus, the OpenMP implementation must execute **ompd_bp_device_end** at every *device-finalize* event. This execution occurs after the device executes all OpenMP regions. After execution of **ompd_bp_device_end**, any *address_space_handle* that was acquired for this device is invalid and should be released by calling **ompd_rel_address_space_handle**.

Cross References

- Device Initialization, see Section 15.4
- **ompd_rel_address_space_handle** Routine, see Section 41.8.1

42.5 ompd_bp_parallel_begin Breakpoint

Format

C

```c
void ompd_bp_parallel_begin(void);
```

C

Semantics

Before starting execution of a **parallel** region, the implementation must execute **ompd_bp_parallel_begin**. Thus, the OpenMP implementation must execute **ompd_bp_parallel_begin** at every *parallel-begin* event. When the implementation reaches **ompd_bp_parallel_begin**, the binding region for **ompd_get_curr_parallel_handle** is the **parallel** region that is beginning and the binding task set for **ompd_get_curr_task_handle** is the encountering task for the **parallel** construct.

Cross References

- **ompd_get_curr_parallel_handle** Routine, see Section 41.5.1
- **ompd_get_curr_task_handle** Routine, see Section 41.6.1
- **parallel** Construct, see Section 12.1

42.6 ompd_bp_parallel_end Breakpoint

Format

C

```
void ompd_bp_parallel_end(void);
```

C

Semantics

After finishing execution of a **parallel** region, the implementation must execute **ompd_bp_parallel_end**. Thus, the OpenMP implementation must execute **ompd_bp_parallel_end** at every *parallel-end* event. When the implementation reaches **ompd_bp_parallel_end**, the binding region for **ompd_get_curr_parallel_handle** is the **parallel** region that is ending and the binding task set for **ompd_get_curr_task_handle** is the encountering task for the **parallel** construct. After execution of **ompd_bp_parallel_end**, any *parallel_handle* that was acquired for the **parallel** region is invalid and should be released by calling **ompd_rel_parallel_handle**.

Cross References

- **ompd_get_curr_parallel_handle** Routine, see Section 41.5.1
- **ompd_get_curr_task_handle** Routine, see Section 41.6.1
- **ompd_rel_parallel_handle** Routine, see Section 41.8.2
- **parallel** Construct, see Section 12.1

42.7 ompd_bp_teams_begin Breakpoint

Format

C

```
void ompd_bp_teams_begin(void);
```

C

Semantics

Before starting execution of a **teams** region, the implementation must execute **ompd_bp_teams_begin**. Thus, the OpenMP implementation must execute **ompd_bp_teams_begin** at every *teams-begin* event. When the implementation reaches **ompd_bp_teams_begin**, the binding region for **ompd_get_curr_parallel_handle** is the **teams** region that is beginning and the binding task set for **ompd_get_curr_task_handle** is the encountering task for the **teams** construct.

Cross References

- **ompd_get_curr_parallel_handle** Routine, see Section 41.5.1
- **ompd_get_curr_task_handle** Routine, see Section 41.6.1
- **teams** Construct, see Section 12.2

42.8 ompd_bp_teams_end Breakpoint

Format

C

```
void ompd_bp_teams_end(void);
```

C

Semantics

After finishing execution of a **teams** region, the implementation must execute **ompd_bp_teams_end**. Thus, the OpenMP implementation must execute **ompd_bp_teams_end** at every *teams-end* event. When the implementation reaches **ompd_bp_teams_end**, the binding region for **ompd_get_curr_parallel_handle** is the **teams** region that is ending and the binding task set for **ompd_get_curr_task_handle** is the encountering task for the **teams** construct. After execution of **ompd_bp_teams_end**, any *parallel_handle* that was acquired for the **teams** region is invalid and should be released by calling **ompd_rel_parallel_handle**.

Cross References

- **ompd_get_curr_parallel_handle** Routine, see Section 41.5.1
- **ompd_get_curr_task_handle** Routine, see Section 41.6.1
- **ompd_rel_parallel_handle** Routine, see Section 41.8.2
- **teams** Construct, see Section 12.2

42.9 ompd_bp_task_begin Breakpoint

Format

C

```c
void ompd_bp_task_begin(void);
```

C

Semantics

Before starting execution of a task region, the implementation must execute **ompd_bp_task_begin**. Thus, the OpenMP implementation must execute **ompd_bp_task_begin** immediately before starting execution of a structured block that is associated with a non-merged task. When the implementation reaches **ompd_bp_task_begin**, the binding task set for **ompd_get_curr_task_handle** is the task that is scheduled to execute.

Cross References

- **ompd_get_curr_task_handle** Routine, see Section 41.6.1

42.10 ompd_bp_task_end Breakpoint

Format

C

```c
void ompd_bp_task_end(void);
```

C

Semantics

After finishing execution of a task region, the implementation must execute **ompd_bp_task_end**. Thus, the OpenMP implementation must execute **ompd_bp_task_end** immediately after completion of a structured block that is associated with a non-merged task. When the implementation reaches **ompd_bp_task_end**, the binding task set for **ompd_get_curr_task_handle** is the task that finished execution. After execution of **ompd_bp_task_end**, any *task_handle* that was acquired for the task region is invalid and should be released by calling **ompd_rel_task_handle**.

Cross References

- **ompd_get_curr_task_handle** Routine, see Section 41.6.1
- **ompd_rel_task_handle** Routine, see Section 41.8.3

42.11 ompd_bp_target_begin Breakpoint

Format

C

```c
void ompd_bp_target_begin(void);
```

C

Semantics

Before starting execution of a **target** region, the implementation must execute **ompd_bp_target_begin**. Thus, the OpenMP implementation must execute **ompd_bp_target_begin** at every *initial-task-begin* event that results from the execution of an initial task enclosing a **target** region. When the implementation reaches **ompd_bp_target_begin**, the binding region for **ompd_get_curr_parallel_handle** is the **target** region that is beginning and the binding task set for **ompd_get_curr_task_handle** is the initial task on the device.

Cross References

- **ompd_get_curr_parallel_handle** Routine, see Section 41.5.1
- **ompd_get_curr_task_handle** Routine, see Section 41.6.1
- **target** Construct, see Section 15.8

42.12 ompd_bp_target_end Breakpoint

Format

C

```
void ompd_bp_target_end(void);
```

C

Semantics

After finishing execution of a **target** region, the implementation must execute **ompd_bp_target_end**. Thus, the OpenMP implementation must execute **ompd_bp_target_end** at every *initial-task-end* event that results from the execution of an initial task enclosing a **target** region. When the implementation reaches **ompd_bp_target_end**, the binding region for **ompd_get_curr_parallel_handle** is the **target** region that is ending and the binding task set for **ompd_get_curr_task_handle** is the initial task on the device. After execution of **ompd_bp_target_end**, any *parallel_handle* that was acquired for the **target** region is invalid and should be released by calling **ompd_rel_parallel_handle**.

Cross References

- **ompd_get_curr_parallel_handle** Routine, see Section 41.5.1
- **ompd_get_curr_task_handle** Routine, see Section 41.6.1
- **ompd_rel_parallel_handle** Routine, see Section 41.8.2
- **target** Construct, see Section 15.8

Part VI

Appendices

A OpenMP Implementation-Defined Behaviors

This appendix summarizes the behaviors that are described as implementation defined in the OpenMP API. Each behavior is cross-referenced back to its description in the main specification. An implementation is required to define and to document its behavior in these cases.

Chapter 1:

- **Memory model**: The minimum size at which a memory update may also read and write back adjacent variables that are part of an aggregate variable is implementation defined but is no larger than the base language requires. The manner in which a program can obtain the referenced device address from a device pointer, outside the mechanisms specified by OpenMP, is implementation defined (see Section 1.3.1).
- **Device data environments**: Whether a variable with static storage duration that is accessible on a device and is not a device-local variable is mapped with a persistent self map at the beginning of the program is implementation defined (see Section 1.3.2).

Chapter 2:

- **Processor**: A hardware unit that is implementation defined (see Chapter 2).
- **Device**: An implementation defined logical execution engine (see Chapter 2).
- **Device pointer**: An implementation defined handle that refers to a device address (see Chapter 2).
- **Supported active levels of parallelism**: The maximum number of active parallel regions that may enclose any region of code in an OpenMP program is implementation defined (see Chapter 2).
- **Deprecated features**: For any deprecated feature, whether any modifications provided by its replacement feature (if any) apply to the deprecated feature is implementation defined (see Chapter 2).

Chapter 3:

- **Internal control variables**: The initial values of *dyn-var*, *nthreads-var*, *run-sched-var*, *bind-var*, *stacksize-var*, *wait-policy-var*, *thread-limit-var*, *max-active-levels-var*, *place-partition-var*, *affinity-format-var*, *default-device-var*, *num-procs-var* and *def-allocator-var* are implementation defined (see Section 3.2).

Chapter 4:

- **OMP_DYNAMIC environment variable**: If the value is neither **true** nor **false**, the behavior of the program is implementation defined (see Section 4.1.2).
- **OMP_NUM_THREADS environment variable**: If any value of the specified list leads to a number of threads that is greater than the implementation can support, or if any value is not a positive integer, then the behavior of the program is implementation defined (see Section 4.1.3).
- **OMP_THREAD_LIMIT environment variable**: If the requested value is greater than the number of threads that an implementation can support, or if the value is not a positive integer, the behavior of the program is implementation defined (see Section 4.1.4).
- **OMP_MAX_ACTIVE_LEVELS environment variable**: If the value is a negative integer or is greater than the maximum number of nested active levels that an implementation can support then the behavior of the program is implementation defined (see Section 4.1.5).
- **OMP_PLACES environment variable**: The meaning of the numbers specified in the environment variable and how the numbering is done are implementation defined. The precise definitions of the abstract names are implementation defined. An implementation may add implementation defined abstract names as appropriate for the target platform. When creating a place list of *n* elements by appending the number *n* to an abstract name, the determination of which resources to include in the place list is implementation defined. When requesting more resources than available, the length of the place list is also implementation defined. The behavior of the program is implementation defined when the execution environment cannot map a numerical value (either explicitly defined or implicitly derived from an interval) within the **OMP_PLACES** list to a processor on the target platform, or if it maps to an unavailable processor. The behavior is also implementation defined when the **OMP_PLACES** environment variable is defined using an abstract name (see Section 4.1.6).
- **OMP_PROC_BIND environment variable**: If the value is not **true**, **false**, or a comma separated list of **primary**, **close**, or **spread**, the behavior is implementation defined. The behavior is also implementation defined if an initial thread cannot be bound to the first place in the OpenMP place list. The thread affinity policy is implementation defined if the value is **true** (see Section 4.1.7).
- **OMP_SCHEDULE environment variable**: If the value does not conform to the specified format then the behavior of the program is implementation defined (see Section 4.3.1).
- **OMP_STACKSIZE environment variable**: If the value does not conform to the specified format or the implementation cannot provide a stack of the specified size then the behavior is implementation defined (see Section 4.3.2).
- **OMP_WAIT_POLICY environment variable**: The details of the **active** and **passive** behaviors are implementation defined (see Section 4.3.3).
- **OMP_DISPLAY_AFFINITY environment variable**: For all values of the environment variable other than **true** or **false**, the display action is implementation defined (see Section 4.3.4).

- **OMP_AFFINITY_FORMAT environment variable**: Additional implementation defined field types can be added (see Section 4.3.5).
- **OMP_CANCELLATION environment variable**: If the value is set to neither **true** nor **false**, the behavior of the program is implementation defined (see Section 4.3.6).
- **OMP_TARGET_OFFLOAD environment variable**: The support of **disabled** is implementation defined (see Section 4.3.9).
- **OMP_THREADS_RESERVE environment variable**: If the requested values are greater than **OMP_THREAD_LIMIT**, the behavior of the program is implementation defined (see Section 4.3.10).
- **OMP_TOOL_LIBRARIES environment variable**: Whether the value of the environment variable is case sensitive is implementation defined (see Section 4.5.2).
- **OMP_TOOL_VERBOSE_INIT environment variable**: Support for logging to **stdout** or **stderr** is implementation defined. Whether the value of the environment variable is case sensitive when it is treated as a filename is implementation defined. The format and detail of the log is implementation defined (see Section 4.5.3).
- **OMP_DEBUG environment variable**: If the value is neither **disabled** nor **enabled**, the behavior is implementation defined (see Section 4.6.1).
- **OMP_NUM_TEAMS environment variable**: If the value is not a positive integer or is greater than the number of teams that an implementation can support, the behavior of the program is implementation defined (see Section 4.2.1).
- **OMP_TEAMS_THREAD_LIMIT environment variable**: If the value is not a positive integer or is greater than the number of threads that an implementation can support, the behavior of the program is implementation defined (see Section 4.2.2).

Chapter 5:

C / C++

- A pragma directive that uses **ompx** as the first processing token is implementation defined (see Chapter 5).
- The attribute namespace of an attribute specifier or the optional namespace qualifier within a **sequence** attribute that uses **ompx** is implementation defined (see Chapter 5).

C / C++

C++

- Whether a **throw** executed inside a region that arises from an exception-aborting directive results in runtime error termination is implementation defined (see Chapter 5).

C++

Fortran

- Any directive that uses **omx** or **ompx** in the sentinel is implementation defined (see Chapter 5).

Fortran

Chapter 6:

- **Collapsed loops**: The particular integer type used to compute the iteration count for the collapsed loop is implementation defined (see Section 6.4.3).

Chapter 7:

Fortran

- **data-sharing attributes**: The data-sharing attributes of dummy arguments that do not have the **VALUE** attribute are implementation defined if the associated actual argument is shared unless the actual argument is a scalar variable, structure, an array that is not a pointer or assumed-shape array, or a simply contiguous array section (see Section 7.1.2).
- **threadprivate directive**: If the conditions for values of data in the threadprivate memories of threads (other than an initial thread) to persist between two consecutive active parallel regions do not all hold, the allocation status of an allocatable variable in the second region is implementation defined (see Section 7.3).

Fortran

- **is_device_ptr clause**: Support for pointers created outside of the OpenMP device memory routines is implementation defined (see Section 7.5.7).

Fortran

- **has_device_addr and use_device_addr clauses**: The result of inquiring about list item properties other than the **CONTIGUOUS** attribute, storage location, storage size, array bounds, character length, association status and allocation status is implementation defined (see Section 7.5.9 and Section 7.5.10).

Fortran

- **aligned clause**: If the *alignment* modifier is not specified, the default alignments for SIMD instructions on the target platforms are implementation defined (see Section 7.12).

Chapter 8:

- **Memory spaces**: The actual storage resources that each memory space defined in Table 8.1 represents are implementation defined. The mechanism that provides the constant value of the variables allocated in the **omp_const_mem_space** memory space is implementation defined (see Section 8.1).
- **Memory allocators**: The minimum size for partitioning allocated memory over storage resources is implementation defined. The default value for the **omp_atk_pool_size** allocator trait (see Table 8.2) is implementation defined. The memory spaces associated with the predefined **omp_cgroup_mem_alloc**, **omp_pteam_mem_alloc** and **omp_thread_mem_alloc** allocators (see Table 8.3) are implementation defined (see Section 8.2).

Chapter 9:

- **OpenMP context**: The accepted *isa-name* values for the *isa* trait, the accepted *arch-name* values for the *arch* trait and the accepted *extension-name* values for the *extension* trait are implementation defined (see Section 9.1).
- **Metadirectives**: The number of times that each expression of the context selector of a **when** clause is evaluated is implementation defined (see Section 9.4.1).
- **Declare variant directives**: If two replacement candidates have the same score then their order is implementation defined. The number of times each expression of the context selector of a **match** clause is evaluated is implementation defined. For calls to **constexpr** base functions that are evaluated in constant expressions, whether any variant replacement occurs is implementation defined. Any differences that the specific OpenMP context requires in the prototype of the variant from the base function prototype are implementation defined (see Section 9.6).
- **declare_simd directive**: If a SIMD version is created and the **simdlen** clause is not specified, the number of concurrent arguments for the procedure is implementation defined (see Section 9.8).
- **Declare target directives**: Whether the same version is generated for different devices, or whether a version that is called in a **target** region differs from the version that is called outside a **target** region, is implementation defined (see Section 9.9).

Chapter 10:

- **requires directive**: Support for any feature specified by a *requirement* clause on a **requires** directive is implementation defined (see Section 10.5).

Chapter 11:

- **stripe construct**: If a generated offsetting loop and a generated grid loop are associated with the same construct, the grid loops may execute additional empty logical iterations. The number of empty logical iterations is implementation defined (see Section 11.7).
- **tile construct**: If a generated grid loop and a generated tile loop are associated with the same construct, the tile loops may execute additional empty logical iterations. The number of empty logical iterations is implementation defined (see Section 11.8).
- **unroll construct**: If no clauses are specified, if and how the loop is unrolled is implementation defined. If the **partial** clause is specified without an *unroll-factor* argument then the unroll factor is a positive integer that is implementation defined (see Section 11.9).

Chapter 12:

- **Default safesync for non-host devices**: Unless indicated otherwise by a **device_safesync** *requirement* clause, if the **parallel** construct is encountered on a non-host device then the default behavior is as if the **safesync** clause appears on the directive with a *width* value that is implementation defined (see Section 12.1).

- **Dynamic adjustment of threads**: Providing the ability to adjust the number of threads dynamically is implementation defined (see Section 12.1.1).
- **Compile-time message**: If the implementation determines that the requested number of threads can never be provided and therefore performs compile-time error termination, the effect of any **message** clause associated with the directive is implementation defined (see Section 12.1.2).
- **Thread affinity**: If another OpenMP thread is bound to the place associated with its position, the place to which a free-agent thread is bound is implementation defined. For the **spread** thread affinity, if $T \leq P$ and T does not divide P evenly, which subpartitions contain $\lceil P/T \rceil$ places is implementation defined. For the **close** and **spread** thread affinity policies, if ET is not zero, which sets have AT positions and which sets have BT positions is implementation defined. Further, the positions assigned to the groups that are assigned sets with BT positions to make the number of positions assigned to each group AT is implementation defined. The determination of whether the thread affinity request can be fulfilled is implementation defined. If the thread affinity request cannot be fulfilled, then the thread affinity of threads in the team is implementation defined (see Section 12.1.3).
- **teams construct**: The number of teams that are created is implementation defined, but it is greater than or equal to the lower bound and less than or equal to the upper bound values of the **num_teams** clause if specified. If the **num_teams** clause is not specified, the number of teams is less than or equal to the value of the *nteams-var* ICV if its value is positive. Otherwise it is an implementation defined positive value (see Section 12.2).
- **simd construct**: The number of iterations that are executed concurrently at any given time is implementation defined (see Section 12.4).

Chapter 13:

- **single construct**: The method of choosing a thread to execute the structured block each time the team encounters the construct is implementation defined (see Section 13.1).
- **sections construct**: The method of scheduling the structured block sequences among threads in the team is implementation defined (see Section 13.3).
- **Worksharing-loop construct**: The schedule that is used is implementation defined if the **schedule** clause is not specified or if the specified schedule has the kind **auto**. The value of *simd_width* for the **simd** schedule modifier is implementation defined (see Section 13.6).
- **distribute construct**: If no **dist_schedule** clause is specified then the schedule for the **distribute** construct is implementation defined (see Section 13.7).

Chapter 14:

- **taskloop construct**: The number of logical iterations assigned to a task created from a **taskloop** construct is implementation defined, unless the **grainsize** or **num_tasks** clause is specified (see Section 14.2).

C++

- **`taskloop` construct**: For firstprivate variables of class type, the number of invocations of copy constructors to perform the initialization is implementation defined (see Section 14.2).
- **`taskgraph` construct**: Whether foreign tasks are recorded or not in a taskgraph record and the manner in which they are executed during a replay execution if they are recorded is implementation defined (see Section 14.3).

C++

Chapter 15:

- **`thread_limit` clause**: The maximum number of threads that participate in executing tasks in the contention group that each team initiates is implementation defined if no `thread_limit` clause is specified on the construct. Otherwise, it has the implementation defined upper bound of the *teams-thread-limit-var* ICV, if the value of this ICV is positive (see Section 15.3).
- **`target` construct**: If a **`device`** clause is specified with the **`ancestor`** *device-modifier*, whether a storage block on the encountering device that has no corresponding storage on the specified device may be mapped is implementation defined (see Section 15.8).

Chapter 16:

- ***prefer-type* modifier**: The supported preference specifications are implementation defined, including the supported foreign runtime identifiers, which may be non-standard names compatible with the modifier. The default preference specification when the implementation supports multiple values is implementation defined (see Section 16.1.3).

Chapter 17:

- **`atomic` construct**: A compliant implementation may enforce exclusive access between `atomic` regions that update different storage locations. The circumstances under which this occurs are implementation defined. If the storage location designated by *x* is not size-aligned (that is, if the byte alignment of *x* is not a multiple of the size of *x*), then the behavior of the `atomic` region is implementation defined (see Section 17.8.5).

Chapter 18:

- None.

Chapter 19:

- None.

Chapter 20:

- **Runtime routines**: Routine names that begin with the **ompx_** prefix are implementation defined extensions to the OpenMP Runtime API (see Chapter 20).

C / C++

- **Runtime library definitions**: The types for the **allocator_handle**, **event_handle**, **interop_fr**, **memspace_handle** and **interop** OpenMP types are implementation defined. The value of the **omp_invalid_device** predefined identifier is implementation defined. The value of the **omp_unassigned_thread** predefined identifier is implementation defined (see Chapter 20).

C / C++

Fortran

- **Runtime library definitions**: Whether the deprecated include file **omp_lib.h** or the module **omp_lib** (or both) is provided is implementation defined. Whether the **omp_lib.h** file provides derived-type definitions or those routines that require an explicit interface is implementation defined. Whether any of the OpenMP API routines that take an argument are extended with a generic interface so arguments of different **KIND** type can be accommodated is implementation defined. The value of the **omp_invalid_device** predefined identifier is implementation defined (see Chapter 20).

Fortran

- **Routine arguments**: The behavior is implementation defined if a routine argument is specified with a value that does not conform to the constraints that are implied by the properties of the argument (see Section 20.3).
- **Interoperability objects**: Implementation defined properties may use non-negative values for properties associated with an interoperability object (see Section 20.7).

Chapter 21:

- **omp_set_schedule routine**: For any implementation defined schedule types, the values and associated meanings of the second argument are implementation defined (see Section 21.9).
- **omp_get_schedule routine**: The value returned by the second argument is implementation defined for any schedule types other than **omp_sched_static**, **omp_sched_dynamic** and **omp_sched_guided** (see Section 21.10).
- **omp_get_supported_active_levels routine**: The number of active levels supported by the implementation is implementation defined, but must be positive (see Section 21.11).
- **omp_set_max_active_levels routine**: If the argument is a negative integer then the behavior is implementation defined. If the argument is less than the *active-levels-var* ICV, the *max-active-levels-var* ICV is set to an implementation defined value between the value of the argument and the value of *active-levels-var*, inclusive (see Section 21.12).

Chapter 22:

- **omp_set_num_teams routine**: If the argument does not evaluate to a positive integer, the behavior of this routine is implementation defined (see Section 22.2).
- **omp_set_teams_thread_limit routine**: If the argument is not a positive integer, the behavior is implementation defined (see Section 22.6).

Chapter 23:

- None.

Chapter 24:

- None.

Chapter 25:

- **Rectangular-memory-copying routine**: The maximum number of dimensions supported is implementation defined, but must be at least three (see Section 25.7).

Chapter 26:

- None.

Chapter 27:

- None.

Chapter 28:

- **Lock routines**: If a lock contains a synchronization hint, the effect of the hint is implementation defined (see Chapter 28).

Chapter 29:

- **omp_get_place_proc_ids routine**: The meaning of the non-negative numerical identifiers returned by the **omp_get_place_proc_ids** routine is implementation defined. The order of the numerical identifiers returned in the array *ids* is implementation defined (see Section 29.4).
- **omp_set_affinity_format routine**: When called from within any **parallel** or **teams** region, the binding thread set (and binding region, if required) for the **omp_set_affinity_format** region and the effect of this routine are implementation defined (see Section 29.8).
- **omp_get_affinity_format routine**: When called from within any **parallel** or **teams** region, the binding thread set (and binding region, if required) for the **omp_get_affinity_format** region is implementation defined (see Section 29.9).
- **omp_display_affinity routine**: If the *format* argument does not conform to the specified format then the result is implementation defined (see Section 29.10).

- **`omp_capture_affinity` routine**: If the *format* argument does not conform to the specified format then the result is implementation defined (see Section 29.11).

Chapter 30:

- **`omp_display_env` routine**: Whether ICVs with the same value are combined or displayed in multiple lines is implementation defined (see Section 30.4).

Chapter 31:

- None.

Chapter 32:

- **Tool callbacks**: If a tool attempts to register a callback not listed in Table 32.2, whether the registered callback may never, sometimes or always invoke this callback for the associated events is implementation defined (see Section 32.2.4).
- **Device tracing**: Whether a target device supports tracing or not is implementation defined. If a target device does not support tracing, a NULL may be supplied for the *lookup* function to the device initializer of a tool (see Section 32.2.5).
- **`set_trace_ompt` and `get_record_ompt` entry points**: Whether a device-specific tracing interface defines this entry point, indicating that it can collect traces in standard trace format, is implementation defined. The kinds of trace records available for a device is implementation defined (see Section 32.2.5).

Chapter 33:

- **`dispatch_chunk` OMPT type**: Whether the chunk of a **`taskloop`** region is contiguous is implementation defined (see Section 33.14).
- **`record_abstract` OMPT type**: The meaning of a **`hwid`** value for a device is implementation defined (see Section 33.24).
- **`state` OMPT type**: The set of OMPT thread states supported is implementation defined (see Section 33.31).

Chapter 34:

- **`sync_region_wait` callback**: For the *implicit-barrier-wait-begin* and *implicit-barrier-wait-end* events at the end of a parallel region, whether the *parallel_data* argument is NULL or points to the parallel data of the current parallel region is implementation defined (see Section 34.7.5).

Chapter 35:

- **`target_data_op_emi` callbacks**: Whether *dev1_addr* or *dev2_addr* points to an intermediate buffer in some operations is implementation defined (see Section 35.7).

Chapter 36:

- **`get_place_proc_ids` entry point**: The meaning of the numerical identifiers returned is implementation defined. The order of *ids* returned in the array is implementation defined (see Section 36.9).
- **`get_partition_place_nums` entry point**: The order of the identifiers returned in the *place_nums* array is implementation defined (see Section 36.11).
- **`get_proc_id` entry point**: The meaning of the numerical identifier returned is implementation defined (see Section 36.12).

Chapter 37:

- None.

Chapter 38:

- None.

Chapter 39:

- None.

Chapter 40:

- **`print_string` callback**: The value of the *category* argument is implementation defined (see Section 40.5).

Chapter 41:

- **handle-comparing routines**: For all types of handles, the means by which two handles are ordered is implementation defined (see Section 41.7).

Chapter 42:

- None.

B Features History

This appendix summarizes the major changes between OpenMP API versions since version 2.5.

B.1 Deprecated Features

The following features were deprecated in Version 6.0:

Fortran

- Omitting the optional white space to separate adjacent keywords in the *directive-name* in free source form and fixed source form directives is deprecated (see Section 5.1.1 and Section 5.1.2).

Fortran

- The syntax of the **declare_reduction** directive that specifies the combiner expression in the directive argument was deprecated (see Section 7.6.14).
- The Fortran include file **omp_lib.h** has been deprecated (see Chapter 20).
- The **target**, **target_data_op**, **target_submit** and **target_map** values of the **callbacks** OMPT types and the associated trace record OMPT type names were deprecated (see Section 33.6).
- The **ompt_target_data_transfer_to_device**, **ompt_target_data_transfer_from_device**, **ompt_target_data_transfer_to_device_async**, and **ompt_target_data_transfer_from_device_async** values in the **target_data_op** OMPT type were deprecated (see Section 33.35).
- The **target_data_op**, **target**, **target_map** and **target_submit** callbacks and the associated trace record OMPT type names were deprecated (see Section 35.7, Section 35.8, Section 35.9 and Section 35.10).

B.2 Version 5.2 to 6.0 Differences

- All features deprecated in versions 5.0, 5.1 and 5.2 were removed.
- Full support for C23, C++23, and Fortran 2023 was added (see Section 1.6).
- Full support of Fortran 2018 was completed (see Section 1.6).
- The environment variable syntax was extended to support initializing ICVs for the host device and non-host devices with a single environment variable (see Section 3.2 and Chapter 4).

- The handling of the *nthreads-var* ICV was updated (see Section 3.4) and the *nthreads* argument of the **num_threads** clause was changed to a list (see Section 12.1.2) to support context-specific reservation of inner parallelism.
- Numeric abstract name values are now allowed for the **OMP_NUM_THREADS**, **OMP_THREAD_LIMIT** and **OMP_TEAMS_THREAD_LIMIT** environment variables (see Section 4.1.3, Section 4.1.4 and Section 4.2.2).
- The environment variable **OMP_PLACES** was extended to support an increment between consecutive places when creating a place list from an abstract name (see Section 4.1.6).
- The environment variable **OMP_AVAILABLE_DEVICES** was added and the environment variable **OMP_DEFAULT_DEVICE** was extended to support device selection by traits (see Section 4.3.7 and Section 4.3.8).
- The *uid* trait was added to the permissible traits in the environment variables **OMP_AVAILABLE_DEVICES** and **OMP_DEFAULT_DEVICE** and to the target device trait set (see Section 4.3.7, Section 4.3.8 and Section 9.2).
- The environment variable **OMP_THREADS_RESERVE** was added to reserve a number of structured threads and free-agent threads (see Section 4.3.10).

C++

- The **decl** attribute was added to improve the attribute syntax for declarative directives (see Section 5.1).

C++

C

- The OpenMP directive syntax was extended to include C attribute specifiers (see Section 5.1).

C

Fortran

- Support for directives with the pure property in **DO CONCURRENT** constructs has been added (see Section 5.1).

Fortran

- To improve consistency in clause format, all inarguable clauses were extended to take an optional argument for which the default value yields equivalent semantics to the existing inarguable semantics (see Section 5.2).
- The **adjust_args** clause was extended to support positional specification of arguments (see Section 5.2.1 and Section 9.6.2)

Fortran

- The definitions of locator list items and assignable OpenMP types were extended to include function references that have data pointer results (see Section 5.2.1).

Fortran

C / C++

- The array section definition was extended to permit, where explicitly allowed, omission of the length when the size of the array dimension is not known (see Section 5.2.5).

C / C++

- To support greater specificity on compound constructs, all clauses were extended to accept the *directive-name-modifier*, which identifies the constituent directives to which the clause applies (see Section 5.4).
- To allow specification of all modifiers of the **init** clause, extensions to the **interop** operation of the **append_args** clause were added (see Section 5.6 and Section 9.6.3).
- The **init** clause was added to the **depobj** construct, and the construct now permits repeatable **init**, **update**, and **destroy** clauses (see Section 5.6 and Section 17.9.3).
- The syntax that omits the argument to the **destroy** clause for the **depobj** construct was undeprecated (see Section 5.7).

Fortran

- Atomic structured blocks were extended to allow the **BLOCK** construct, pointer assignments and two intrinsic functions for enum and enumeration types (see Section 6.3.3).
- *conditional-update-statement* was extended to allow more forms and comparisons (see Section 6.3.3).

Fortran

- The concept of canonical loop sequences and the **looprange** clause were defined (see Section 6.4.2 and Section 6.4.7).

Fortran

- For polymorphic types, restrictions were changed and behavior clarified for data-sharing attribute clauses and data-mapping attribute clauses (see Chapter 7).

Fortran

- The *saved* modifier, the **replayable** clause, and the **taskgraph** construct were added to support the recording and efficient replay execution of a sequence of task-generating constructs (see Section 7.2, Section 14.6, and Section 14.3).
- The **default** clause is now allowed on the **target** directive, and, similarly to the **defaultmap** clause, now accepts the *variable-category* modifier (see Section 7.5.1).
- The semantics of the **use_device_ptr** and **use_device_addr** clauses on a **target_data** construct were altered to imply a reference count update on entry and exit from the region for the corresponding objects that they reference in the device data environment (see Section 7.5.8 and Section 7.5.10).
- Support for induction operations was added (see Section 7.6) through the **induction** clause (see Section 7.6.13) and the **declare_induction** directive (see Section 7.6.17), which supports user-defined induction.
- Support for reductions over private variables with the **reduction** clause has been added (see Section 7.6).

C++

- The circumstances under which implicitly declared reduction identifiers are supported for variables of class type were clarified (see Section 7.6.3 and Section 7.6.6).

C++

- The **scan** directive was extended to accept the **init_complete** clause to enable the identification of an initialization phase within the *final-loop-body* of an enclosing **simd** construct or worksharing-loop construct (or a composite construct that combines them) (see Section 7.7 and Section 7.7.3).
- The **storage** *map-type* modifier was added as the preferred *map-type* when the mapping operation only allocates or releases storage on the target device (see Section 7.9.1).
- The **ref** modifier was added to the **map** clause to add more control over how the clause affects list items that are C++ references or Fortran pointer/allocatable variables (see Section 7.9.5 and Section 7.9.6).
- The property of the *map-type* modifier was changed to *default* so that it can be freely placed and omitted even if other modifiers are used (see Section 7.9.6).
- The **self** *map-type-modifier* was added to the **map** clause and the **self** *implicit-behavior* was added to the **defaultmap** clause to request explicitly that the corresponding list item refers to the same object as the original list item (see Section 7.9.6 and Section 7.9.9).
- The **map** clause was extended to permit mapping of assumed-size arrays (see Section 7.9.6).
- The **delete** keyword on the **map** clause was reformulated to be the *delete-modifier* (see Section 7.9.6).

Fortran

- The **automap** modifier was added to the **enter** clause to support automatic mapping and unmapping of Fortran allocatable variables when allocated and deallocated, respectively (see Section 7.9.7).

Fortran

- The **groupprivate** directive was added to specify that variables should be privatized with respect to a contention group (see Section 7.13).
- The **local** clause was added to the **declare_target** directive to specify that variables should be replicated locally for each device (see Section 7.14).
- The allocator trait **omp_atk_part_size** was added to specify the size of the **omp_atv_interleaved** allocator partitions (see Section 8.2).
- The **omp_atk_pin_device**, **omp_atk_preferred_device** and **omp_atk_target_access** memory allocator traits were defined to provide greater control of memory allocations that may be accessible from multiple devices (see Section 8.2).

- The **device** value of the **access** allocator trait was defined as the default **access** allocator trait and to provide the semantics that an allocator with the trait corresponds to memory that all threads on a specific device can access. The semantics of an allocator with the **all** value were updated to correspond to memory that all threads in the system can access (see Section 8.2).
- The **omp_atv_partitioner** value was added to the possible values of the **omp_atk_partition** allocator trait to allow ad-hoc user partitions (see Section 8.2).
- The **uses_allocators** clause was extended to permit more than one *clause-argument-specification* (see Section 8.8).
- The **need_device_addr** modifier was added to the **adjust_args** clause to support adjustment of arguments passed by reference (see Section 9.6.2).
- The **dispatch** construct was extended with the **interop** clause to support appending arguments specific to a call site (see Section 9.7 and Section 9.7.1).

C / C++

- A **declare_target** directive that specifies list items must now be placed at the same scope as the declaration of those list items, and if the directive does not specify list items then it is treated as declaration-associated (see Section 9.9.1).

C / C++

- The **message** and **severity** clauses were added to the **parallel** directive to support customization of any error termination associated with the directive (see Section 10.3, Section 10.4, and Section 12.1).
- The **self_maps** *requirement* clause was added to require that all mapping operations are self maps (see Section 10.5.1.6).
- The *assumption* clause group was extended with the **no_openmp_constructs** clause to support identification of regions in which no constructs will be encountered (see Section 10.6.1 and Section 10.6.1.5).
- A restriction for loop-transforming constructs was added that the generated loop must not be a doacross-affected loop, which implies that, in an **unroll** construct with an *unroll-factor* of one, a stand-alone **ordered** directive is now non-conforming (see Chapter 11, Section 11.9 and Section 17.10.1).
- The **apply** clause was added to enable more flexible composition of loop-transforming constructs (see Section 11.1).
- The **sizes** clause was updated to allow non-constant list items (see Section 11.2).
- The **fuse** construct was added to fuse two or more loops in a canonical loop sequence (see Section 11.3).
- The **interchange** construct was added to permute the order of loops in a loop nest (see Section 11.4).
- The **reverse** construct was added to reverse the iteration order of a loop (see Section 11.5).

- The **split** loop-transforming construct was added to apply index-set splitting to canonical loop nests (see Section 11.6).
- The **stripe** loop-transforming construct was added to apply striping to canonical loop nests (see Section 11.7).
- The **tile** construct was extended to allow grid loops and tile loops to be affected by the same construct (see Section 11.8).
- The *prescriptiveness* modifier was added to the **num_threads** clause and **strict** semantics were defined for the clause (see Section 12.1.2).
- To control which synchronizing threads are guaranteed to make progress eventually, the **safesync** clause on the **parallel** construct (see Section 12.1.5), the **omp_curr_progress_width** identifier (see Section 20.1) and the **omp_get_max_progress_width** routine were addded (see Section 24.6).
- To make the **loop** construct and other constructs that specify the **order** clause with **concurrent** *ordering* more usable, calls to procedures in the region may now contain certain OpenMP directives (see Section 12.3).
- To support a wider range of synchronization choices, the **atomic** construct was added to the constructs that may be encountered inside a region that corresponds to a construct with an **order** clause that specifies **concurrent** (see Section 12.3).
- The constructs that may be encountered during the execution of a region that corresponds to a construct on which the **order** clause is specified with **concurrent** *ordering*, when the corresponding regions are not strictly nested regions, are no longer restricted (see Section 12.3).

Fortran

- The **workdistribute** directive was added to support Fortran array expressions in **teams** constructs (see Section 13.5).
- The **loop** construct was extended to allow a **DO CONCURRENT** loop as the collapsed loop (see Section 13.8).

Fortran

- The **taskloop** construct now includes the **task_iteration** directive as a subsidiary directive so that the tasks that it generates can include the semantics of the **affinity** and **depend** clauses (see Section 14.2, Section 14.2.3, Section 14.10 and Section 17.9.5).
- The **threadset** clause was added to task-generating constructs to specify the binding thread set of the generated task (see Section 14.8).
- The **priority** clause was added to the **target_enter_data**, **target_exit_data**, **target_data**, **target** and **target_update** directives (see Section 14.9, Section 15.5, Section 15.6, Section 15.7, Section 15.8 and Section 15.9).
- The **device_type** clause was added to the clauses that may appear on the **target** construct (see Section 15.1 and Section 15.8).

- When the **device** clause is specified with the **ancestor** *device-modifier* on the **target** construct, the **nowait** clause may now also be specified (see Section 15.2, Section 15.8 and Section 17.6).
- The **target_data** directive description was updated to make it a composite construct, to include a **taskgroup** region and to make the clauses that may appear on it reflect its constituent constructs and the **taskgroup** region (see Section 15.7).
- The *prefer-type* modifier of the **init** clause was updated to allow preferences other than foreign runtime identifiers (see Section 16.1.3).
- The *do_not_synchronize* argument for the **nowait** clause (see Section 17.6) and **nogroup** clause (see Section 17.7) was updated to permit non-constant expressions.
- The **memscope** clause was added to the **atomic** and **flush** constructs to allow the binding thread set to span multiple devices (see Section 17.8.4, Section 17.8.5 and Section 17.8.6).
- The **transparent** clause was added to support multi-generational task dependence graphs (see Section 17.9.6).
- The **cancel** construct was extended to complete tasks that have not yet been fulfilled through an event variable and the **omp_fulfill_event** routine was restricted such that an event handle must be fulfilled before execution continues beyond a barrier (see Section 18.2 and Section 23.2.1).
- The rules for compound-directive names were simplified to be more intuitive and to allow more valid combinations of immediately nested constructs (see Section 19.1).
- The **omp_is_free_agent** and **omp_ancestor_is_free_agent** routines were added to test whether the encountering thread, or the ancestor thread, is a free-agent thread (see Section 23.1.4 and Section 23.1.5).
- The **omp_get_device_from_uid** and **omp_get_uid_from_device** routines were added to convert between unique identifiers and device numbers of devices (see Section 24.7 and Section 24.8).
- The **omp_get_device_num_teams**, **omp_set_device_num_teams**, **omp_get_device_teams_thread_limit**, and **omp_set_device_teams_thread_limit** routine were added to support getting and setting the *nteams-var* and *teams-thread-limit-var* ICVs for specific devices (see Section 24.11, Section 24.12, Section 24.13, and Section 24.14).
- The **omp_target_memset** and **omp_target_memset_async** routines were added to fill memory in a device data environment of a device (see Section 25.8.1 and Section 25.8.2).

Fortran

- Fortran versions of the runtime routines to operate on interoperability objects were added (see Chapter 26).

Fortran

- New routines were added to obtain memory spaces and memory allocators to allocate remote and shared memory (see Chapter 27).
- The **omp_get_memspace_num_resources** routine was added to support querying the number of available resources of a memory space (see Section 27.2).
- The **omp_get_memspace_pagesize** routine was added to obtain the page size supported by a given memory space (see Section 27.3).
- The **omp_get_submemspace** routine was added to obtain a memory space with a subset of the original storage resources (see Section 27.4).
- The **omp_init_mempartitioner**, **omp_destroy_mempartitioner**, **omp_init_mempartition**, **omp_destroy_mempartition**, **omp_mempartition_set_part**, **omp_mempartition_get_user_data** routines were added to manipulate the **mempartitioner** and **mempartition** objects (see Section 27.5).
- The set of callbacks for which **set_callback** must return **ompt_set_always** no longer includes the **target_data_op**, **target**, **target_map** and **target_submit** callbacks, which were deprecated (see Section 32.2.4, Section 35.7, Section 35.8, Section 35.9 and Section 35.10).
- The more general values **ompt_target_data_transfer** and **ompt_target_data_transfer_async** were added to the **target_data_op** OMPT type and supersede the values **ompt_target_data_transfer_to_device**, **ompt_target_data_transfer_from_device**, **ompt_target_data_transfer_to_device_async** and **ompt_target_data_transfer_from_device_async** (see Section 33.35). The superseded values were deprecated.
- The **get_buffer_limits** entry point was added to the OMPT device tracing interface so that a first-party tool can obtain an upper limit on the sizes of the trace buffers that it should make available to the implementation (see Section 37.6).

B.3 Version 5.1 to 5.2 Differences

- Major reorganization and numerous changes were made to improve the quality of the specification of OpenMP syntax and to increase consistency of restrictions and their wording. These changes frequently result in the possible perception of differences to preceding versions of the OpenMP specification. However, those differences almost always resolve ambiguities, which may nonetheless have implications for existing implementations and programs.
- The *explicit-task-var* ICV replaced the *implicit-task-var* ICV, with the opposite meaning and semantics (see Chapter 3). The **omp_in_explicit_task** routine was added to query if a code region is executed from an explicit task region (see Section 23.1.2).

Fortran

- Expanded the directives that may be encountered in a pure procedure (see Chapter 5) by adding the pure property to metadirectives (see Section 9.4.3), assumption directives (see Section 10.6), the **nothing** directive (see Section 10.7), the **error** directive (see Section 10.1) and loop-transforming constructs (see Chapter 11).

Fortran

- For OpenMP directives, the **omp** sentinel and, for implementation defined directives that extend the OpenMP directives, the **ompx** sentinel for C/C++ and free source form Fortran and the **omx** sentinel for fixed source form Fortran (to accommodate character position requirements) were reserved (see Chapter 5. Reserved clause names that begin with the **ompx_** prefix for implementation defined clauses on OpenMP directives (see Chapter 5. Reserved names in the base language that start with the **omp_**,**ompt_**, **ompd_** and **ompx_** prefixes and reserved the **omp**, **ompx**, **ompt** and **ompd** namespaces for the OpenMP runtime API and for implementation defined extensions to that API (see Chapter 5.
- Allowed any clause that can be specified on a paired end directive to be specified on the directive (see Section 5.1), including, in Fortran, the **copyprivate** clause (see Section 7.8.2) and the **nowait** clause (see Section 17.6).
- Allowed the **if** clause on the **teams** construct (see Section 5.5 and Section 12.2).
- For consistency with the syntax of other definitions of the clause, the syntax of the **destroy** clause on the **depobj** construct with no argument was deprecated (see Section 5.7).
- For consistency with the syntax of other clauses, the syntax of the **linear** clause that specifies its argument and *linear-modifier* as *linear-modifier* **(***list***)** was deprecated and the *step* modifier was added for specifying the linear step (see Section 7.5.6).
- The *minus* (**-**) operator for reductions was deprecated (see Section 7.6.6).
- The syntax of modifiers without comma separators in the **map** clause was deprecated (see Section 7.9.6).
- To support the complete range of user-defined mappers and to improve consistency of **map** clause usage, the **declare_mapper** directive was extended to accept *iterator* modifiers and the **present** *map-type-modifier* (see Section 7.9.6 and Section 7.9.10).
- Mapping of a pointer list item was updated such that if a matched candidate is not found in the data environment, firstprivate semantics apply and the pointer retains its original value (see Section 7.9.6).
- The **enter** clause was added as a synonym for the **to** clause on declare target directives, and the corresponding **to** clause was deprecated to reduce parsing ambiguity (see Section 7.9.7 and Section 9.9).

Fortran

- The **allocators** construct was added to support the use of OpenMP allocators for variables that are allocated by a Fortran **ALLOCATE** statement, and the application of **allocate** directives to an **ALLOCATE** statement was deprecated (see Section 8.7).

Fortran

- To support the full range of allocators and to improve consistency with the syntax of other clauses, the argument that specified the arguments of the **uses_allocators** clause as a comma-separated list in which each list item is a *clause-argument-specification* of the form *allocator[* **(***traits***)** *]* was deprecated (see Section 8.8).
- To improve code clarity and to reduce ambiguity in this specification, the **otherwise** clause was added as a synonym for the **default** clause on metadirectives and the corresponding **default** clause syntax was deprecated (see Section 9.4.2).

Fortran

- For consistency with other constructs with associated base language code, the **dispatch** construct was extended to allow an optional paired end directive to be specified (see Section 9.7).

Fortran

C / C++

- To improve overall syntax consistency and to reduce redundancy, the delimited form of the **declare_target** directive was deprecated (see Section 9.9.2).

C / C++

- The behavior of the **order** clause with the *concurrent* argument was changed so that it only affects whether a loop schedule is reproducible if a modifier is explicitly specified (see Section 12.3).
- Support for the **allocate** and **firstprivate** clauses on the **scope** directive was added (see Section 13.2).
- The **work** OMPT type values for worksharing-loop constructs were added (see Section 13.6).
- To simplify usage, the **map** clause on a **target_enter_data** or **target_exit_data**, construct now has a default map type that provides the same behavior as the **to** or **from** map types, respectively (see Section 15.5 and Section 15.6).
- The **interop** construct was updated to allow the **init** clause to accept an *interop_type* in any position of the modifier list (see Section 16.1).
- The **doacross** clause was added as a synonym for the **depend** clause with the keywords **source** and **sink** as *dependence-type* modifiers and the corresponding **depend** clause syntax was deprecated to improve code clarity and to reduce parsing ambiguity. Also, the **omp_cur_iteration** keyword was added to represent a logical iteration vector that refers to the current logical iteration (see Section 17.9.7).
- The **omp_pause_stop_tool** value was added to the **pause_resource** OpenMP type (see Section 20.11.1).

B.4 Version 5.0 to 5.1 Differences

- Full support of C11, C++11, C++14, C++17, C++20 and Fortran 2008 was completed (see Section 1.6).

- Various changes throughout the specification were made to provide initial support of Fortran 2018 (see Section 1.6).
- To support device-specific ICV settings the environment variable syntax was extended to support device-specific environment variables (see Section 3.2 and Chapter 4).
- The **OMP_PLACES** syntax was extended (see Section 4.1.6).
- The **OMP_NUM_TEAMS** and **OMP_TEAMS_THREAD_LIMIT** environment variables were added to control the number and size of teams on the **teams** construct (see Section 4.2.1 and Section 4.2.2).
- The OpenMP directive syntax was extended to include C++ attribute specifiers (see Section 5.1).
- The **omp_all_memory** reserved locator was added (see Section 5.2.2), and the **depend** clause was extended to allow its use (see Section 17.9.5).
- Support for **private** and **firstprivate** as an argument to the **default** clause in C and C++ was added (see Section 7.5.1).
- The **has_device_addr** clause was added to the **target** construct to allow access to variables or array sections that already have a device address (see Section 7.5.9 and Section 15.8).
- Support was added so that iterators may be defined and used in **map** clauses (see Section 7.9.6) or in data-motion clauses on a **target_update** directive (see Section 15.9).
- The **present** argument was added to the **defaultmap** clause (see Section 7.9.9).
- Support for the **align** clause on the **allocate** directive and *allocator* and *align* modifiers on the **allocate** clause was added (see Chapter 8).
- The *target_device* trait set was added to the OpenMP context (see Section 9.1), and the **target_device** selector set was added to context selectors (see Section 9.2).
- For C/C++, the declare variant directives were extended to support elision of preprocessed code and to allow enclosed function definitions to be interpreted as function variants (see Section 9.6).
- The **declare_variant** directive was extended with new clauses (**adjust_args** and **append_args**) that support adjustment of the interface between the original function and its function variants (see Section 9.6.4).
- The **dispatch** construct was added to allow users to control when variant substitution happens and to define additional information that can be passed as arguments to function variants (see Section 9.7).
- Support was added for indirect calls to the device version of a procedure in **target** regions (see Section 9.9).
- To allow users to control the compilation process and runtime error actions, the **error** directive was added (see Section 10.1).
- Assumption directives were added to allow users to specify invariants (see Section 10.6).

- To support clarity in metadirectives, the **nothing** directive was added (see Section 10.7).
- Loop-transforming constructs were added (see Chapter 11).
- The **masked** construct was added to support restricting execution to a specific thread to replace the deprecated **master** construct (see Section 12.5).
- The **scope** directive was added to support reductions without requiring a **parallel** or worksharing region (see Section 13.2).
- The **grainsize** and **num_tasks** clauses for the **taskloop** construct were extended with a **strict** *prescriptiveness* modifier to ensure a deterministic distribution of logical iterations to tasks (see Section 14.2).
- The **thread_limit** clause was added to the **target** construct to control the upper bound on the number of threads in the created contention group (see Section 15.8).
- The **interop** directive was added to enable portable interoperability with foreign execution contexts (see Section 16.1). Runtime routines that facilitate use of interoperability objects were also added (see Chapter 26).
- The **nowait** clause was added to the **taskwait** directive to support insertion of non-blocking join operations in a task dependence graph (see Section 17.5).
- Specification of the **seq_cst** clause on a **flush** construct was allowed, with the same meaning as a **flush** construct without a list and without a clause (see Section 17.8.1.5 and Section 17.8.6).
- Support was added for compare-and-swap and (for C and C++) minimum and maximum atomic operations through the **compare** clause. Support was also added for the specification of the memory order to apply to a failed atomic conditional update with the **fail** clause (see Section 17.8.3.2 and Section 17.8.3.3).
- To support inout sets, the **inoutset** *task-dependence-type* modifier was added to the **depend** clause (see Section 17.9.5).
- For the **allocatrait_key** OpenMP type, the **omp_atv_serialized** value was added and the **omp_atv_default** value was changed (see Section 20.8).
- The **omp_set_num_teams** and **omp_set_teams_thread_limit** routines were added to control the number of teams and the size of those teams on the **teams** construct (see Section 22.2 and Section 22.6). Additionally, the **omp_get_max_teams** and **omp_get_teams_thread_limit** routines were added to retrieve the values that will be used in the next **teams** construct (see Section 22.4 and Section 22.5).
- The **omp_target_is_accessible** routine was added to test whether a host address is accessible from a given device (see Section 25.2.2).
- The **omp_get_mapped_ptr** routine was added to support obtaining the device pointer that is associated with a host pointer for a given device (see Section 25.2.3).
- To support asynchronous device memory management, **omp_target_memcpy_async** and **omp_target_memcpy_rect_async** routines were added (see Section 25.7.3 and Section 25.7.4).

- The **omp_calloc**, **omp_realloc**, **omp_aligned_alloc** and **omp_aligned_calloc** routines were added (see Chapter 27).
- The **omp_display_env** routine was added to provide information about ICVs and settings of environment variables (see Section 30.4).
- The **ompt_scope_beginend** value was added to the **scope_endpoint** OMPT type to indicate the coincident beginning and end of a scope (see Section 33.27).
- The **ompt_state_wait_barrier_implementation** and **ompt_state_wait_barrier_teams** values were added to the **state** OMPT type (see Section 33.31).
- The **ompt_sync_region_barrier_implicit_workshare**, **ompt_sync_region_barrier_implicit_parallel**, and **ompt_sync_region_barrier_teams** values were added to the **sync_region** OMPT type (see Section 33.33).
- Values for asynchronous data transfers were added to the **target_data_op** OMPT type (see Section 33.35).
- The **error** callback was added (see Section 34.2).
- The **target_data_op_emi**, **target_emi**, **target_map_emi**, and **target_submit_emi** callbacks were added to support external monitoring interfaces (see Section 35.7, Section 35.8, Section 35.9 and Section 35.10).

B.5 Version 4.5 to 5.0 Differences

- The memory model was extended to distinguish different types of flushes according to specified flush properties (see Section 1.3.4) and to define a happens-before order based on synchronizing flushes (see Section 1.3.5).
- Various changes throughout the specification were made to provide initial support of C11, C++11, C++14, C++17 and Fortran 2008 (see Section 1.6).
- Full support of Fortran 2003 was completed (see Section 1.6).
- The *target-offload-var* ICV (see Chapter 3) and the **OMP_TARGET_OFFLOAD** environment variable (see Section 4.3.9) were added to support runtime control of the execution of device constructs.
- Control over whether nested parallelism is enabled or disabled was integrated into the *max-active-levels-var* ICV (see Section 3.2), the default value of which was made implementation defined, unless determined according to the values of the **OMP_NUM_THREADS** (see Section 4.1.3) or **OMP_PROC_BIND** (see Section 4.1.7) environment variables.
- The **OMP_DISPLAY_AFFINITY** (see Section 4.3.4) and **OMP_AFFINITY_FORMAT** (see Section 4.3.5) environment variables and the **omp_set_affinity_format** (see Section 29.8), **omp_get_affinity_format** (see Section 29.9),

omp_display_affinity (see Section 29.10), and omp_capture_affinity (see Section 29.11) routines were added to provide OpenMP runtime thread affinity information.

- The omp_set_nested and omp_get_nested routines and the OMP_NESTED environment variable were deprecated.
- Support for array shaping (see Section 5.2.4) and for array sections with non-unit strides in C and C++ (see Section 5.2.5) was added to facilitate specification of discontiguous storage, and the target_update construct (see Section 15.9) and the depend clause (see Section 17.9.5) were extended to allow the use of shape-operators (see Section 5.2.4).
- The *iterator* modifier (see Section 5.2.6) was added to support expressions in a list that expand to multiple expressions.
- The canonical loop nest form was defined for Fortran and, for all base languages, extended to permit non-rectangular loops (see Section 6.4.1).
- The *relational-op* in a canonical loop nest for C/C++ was extended to include != (see Section 6.4.1).
- To support conditional assignment to lastprivate variables, the *conditional* modifier was added to the lastprivate clause (see Section 7.5.5).
- The semantics of the use_device_ptr clause for pointer variables was clarified and the use_device_addr clause for using the device address of non-pointer variables inside the target_data construct was added (see Section 7.5.8, Section 7.5.10 and Section 15.7).
- The *inscan* modifier for the reduction clause (see Section 7.6.10) and the scan directive (see Section 7.7) were added to support inclusive scan and exclusive scan computations.
- To support task reductions, the *task* modifier was added to the reduction clause (see Section 7.6.10), the task_reduction clause (see Section 7.6.11) was added to the taskgroup construct (see Section 17.4), and the in_reduction clause (see Section 7.6.12) was added to the task (see Section 14.1) and target (see Section 15.8) constructs.
- To support taskloop reductions, the reduction (see Section 7.6.10) and in_reduction (see Section 7.6.12) clauses were added to the taskloop construct (see Section 14.2).
- The description of the map clause was modified to clarify the mapping order when multiple *map-type* modifiers are specified for a variable or structure members of a variable on the same construct. The *close-modifier* was added as a hint for the runtime to allocate memory close to the target device (see Section 7.9.6).
- The capability to map C/C++ pointer variables and to assign the address of device memory that is mapped by an array section to them was added. Support for mapping of Fortran pointer and allocatable variables, including pointer and allocatable components of variables, was added (see Section 7.9.6).
- All uses of the map clause (see Section 7.9.6), as well as the to and from clauses on the target_update construct (see Section 15.9) and the depend clause on task-generating

constructs (see Section 17.9.5) were extended to allow any lvalue expression as a list item for C/C++.

- The **defaultmap** clause (see Section 7.9.9) was extended to allow specification of the data-mapping attributes or data-sharing attributes for any of the scalar, aggregate, pointer, or allocatable classes on a per-region basis. Additionally, the **none** argument was added to support the requirement that all variables referenced in the construct must be explicitly mapped or privatized.
- The **declare_mapper** directive was added to support mapping of data types with direct and indirect members (see Section 7.9.10).
- Predefined memory spaces, predefined memory allocators and allocator traits and directives, clauses and routines (see Chapter 8 and Chapter 27) to use them were added to support different kinds of memories.
- Metadirectives (see Section 9.4) and declare variant directives (see Section 9.6) were added to support selection of directive variants and function variants at a call site, respectively, based on compile-time traits of the enclosing context.
- Support for nested declare target directives was added (see Section 9.9).
- To reduce programmer effort, implicit declare target directives for some procedures were added (see Section 9.9 and Section 15.8).
- The **requires** directive (see Section 10.5) was added to support applications that require implementation-specific features.
- The **teams** construct (see Section 12.2) was extended to support execution on the host device without an enclosing **target** construct (see Section 15.8).
- The **loop** construct and the **order** clause with the **concurrent** argument were added to support compiler optimization and parallelization of loops for which logical iterations may execute in any order, including concurrently (see Section 12.3 and Section 13.8).
- The collapse of affected loops that are imperfectly nested loops was defined for **simd** constructs (see Section 12.4), worksharing-loop constructs (see Section 13.6), **distribute** constructs (see Section 13.7) and **taskloop** constructs (see Section 14.2).
- The **simd** construct (see Section 12.4) was extended to accept the **if** and **nontemporal** clauses and, with the **concurrent** argument, **order** clauses and to allow the use of **atomic** constructs within it.
- The default *ordering-modifier* for the **schedule** clause on worksharing-loop constructs when the *kind* argument is not **static** and the **ordered** clause does not appear on the construct was changed to **nonmonotonic** (see Section 13.6.3).
- The clauses that can be specified on the **task** construct (see Section 14.1) were extended with the **affinity** clause (see Section 14.10) to support hints that indicate data affinity of explicit tasks.
- To support execution of detachable tasks, the **detach** clause for the **task** construct (see Section 14.1) and the **omp_fulfill_event** routine (see Section 23.2.1) were added.

- The **taskloop** construct (see Section 14.2) was added to the list of constructs that can be canceled by the **cancel** constructs (see Section 18.2).
- To support reverse-offload regions, the *ancestor* modifier was added to the **device** clause for the **target** construct (see Section 15.2 and Section 15.8).
- The **target_update** construct (see Section 15.9) was modified to allow array sections that specify discontiguous storage.
- The **taskwait** construct was extended to accept the **depend** clause (see Section 17.5 and Section 17.9.5).
- To support acquire and release semantics with weak memory ordering, the **acq_rel**, **acquire**, and **release** clauses (see Section 17.8.1) were added to the **atomic** construct (see Section 17.8.5) and **flush** construct (see Section 17.8.6), and the memory ordering semantics of implicit flushes on various constructs and routines were clarified (see Section 17.8.7).
- The **atomic** construct was extended with the **hint** clause (see Section 17.8.5).
- To support mutually exclusive inout sets, a **mutexinoutset** *task-dependence-type* was added to the **depend** clause (see Section 17.9.1 and Section 17.9.5).
- The **depend** clause (see Section 17.9.5) was extended to support *iterator* modifiers and to support depend objects that can be created with the new **depobj** construct (see Section 17.9.3).
- New combined constructs (**master taskloop**, **parallel master**, **parallel master taskloop**, **master taskloop simd** and **parallel master taskloop simd**) (see Section 19.1) were added.
- Lock hints were renamed to synchronization hints, and the old names were deprecated (see Section 20.9.5).
- The **omp_get_supported_active_levels** routine was added to query the number of active levels of parallelism supported by the implementation (see Section 21.11).
- The **omp_get_device_num** routine (see Section 24.4) was added to support determination of the device on which a thread is executing.
- The **omp_pause_resource** and **omp_pause_resource_all** routines were added to allow the runtime to relinquish resources used by OpenMP (see Section 30.2.1 and Section 30.2.2).
- Support for a first-party tool interface (see Chapter 32) was added.
- Support for a third-party tool interface (see Chapter 38) was added.
- Stubs for runtime library routines (previously Appendix A) were moved to a separate document.
- Interface declarations (previously Appendix B) were moved to a separate document.

B.6 Version 4.0 to 4.5 Differences

- Support for several features of Fortran 2003 was added (see Section 1.6).
- The **OMP_MAX_TASK_PRIORITY** environment variable was added to control the maximum task priority value allowed (see Section 4.3.11).
- The **if** clause was extended to accept a *directive-name-modifier* that allows it to apply to combined constructs (see Section 5.4 and Section 5.5).
- An argument was added to the **ordered** clause of the worksharing-loop construct and the **ordered** construct was modified to support doacross loop nests (see Section 6.4.6, Section 13.6 and Section 17.10.2)
- The implicitly determined data-sharing attribute for scalar variables in **target** regions was changed to firstprivate (see Section 7.1.1).
- Use of some C++ reference types was allowed in some data-sharing attribute clauses (see Section 7.5).
- The **private**, **firstprivate** and **defaultmap** clauses were added to the **target** construct (see Section 7.5.3, Section 7.5.4, Section 7.9.9 and Section 15.8).
- The *linear-modifier* was added to the **linear** clause (see Section 7.5.6).
- The **linear** clause was added to the worksharing-loop construct (see Section 7.5.6 and Section 13.6).
- To support interaction with native device implementations, the **is_device_ptr** clause was added to the **target** construct and the **use_device_ptr** clause was added to the **target_data** construct (see Section 7.5.7, Section 7.5.8, Section 15.7 and Section 15.8).
- Semantics for reductions on C/C++ array sections were added and restrictions on the use of arrays and pointers in reductions were removed (see Section 7.6.10).
- Support was added to the **map** clause to handle structure elements (see Section 7.9.6).
- To support unstructured data mapping for devices, the **map** clause (see Section 7.9.6) was updated and the **target_enter_data** (see Section 15.5) and **target_exit_data** (see Section 15.6) constructs were added.
- The **declare_target** directive was extended to allow mapping of global variables to be deferred to specific device executions and to allow an *extended-list* to be specified in C/C++ (see Section 9.9).
- The **simdlen** clause was added to the **simd** construct to support specification of the exact number of logical iterations desired per SIMD chunk (see Section 12.4).
- To support the use of the **simd** construct on loops with loop-carried backward dependences with or without a worksharing-loop construct, clauses were added to the **ordered** construct (see Section 12.4, Section 13.6) and Section 17.10).
- The **task** construct was extended to accept hints that the **priority** clause specifies (see Section 14.1 and Section 14.9).

- The **taskloop** construct was added to support nestable parallel loops that create explicit tasks (see Section 14.2).
- To improve support for asynchronous execution of **target** regions, the **target** construct was extended to accept the **nowait** and **depend** clauses (see Section 15.8, Section 17.6 and Section 17.9.5).
- The **hint** clause was added to the **critical** construct (see Section 17.2).
- The **source** and **sink** dependence types were added to the **depend** clause to support doacross loop nests (see Section 17.9.5).
- To support a more complete set of compound constructs for devices, the compound constructs **target parallel**, **target parallel for** (C/C++), **target parallel for simd** (C/C++), **target parallel do** (Fortran) and **target parallel do simd** (Fortran) were added (see Section 19.1).
- The **omp_get_max_task_priority** routine was added to return the maximum supported task priority value (see Section 23.1.1).
- Device memory routines were added to allow explicit memory allocations, deallocations and transfers and memory associations (see Chapter 25).
- The lock API was extended with lock routines that support storing a hint with a lock to select a desired lock implementation for the intended usage of the lock by the application code (see Section 28.1.3 and Section 28.1.4).
- Query routines for thread affinity were added (see Section 29.2 to Section 29.7).
- C/C++ grammar (previously Appendix B) was moved to a separate document.

B.7 Version 3.1 to 4.0 Differences

- Various changes throughout the specification were made to provide initial support of Fortran 2003 (see Section 1.6).
- The **OMP_PLACES** environment variable (see Section 4.1.6), the **proc_bind** clause (see Section 12.1.3), and the **omp_get_proc_bind** routine (see Section 29.1) were added to support thread affinity policies.
- The **OMP_CANCELLATION** environment variable (see Section 4.3.6), the **cancel** construct (see Section 18.2), the **cancellation point** construct (see Section 18.3), and the **omp_get_cancellation** routine (see Section 30.1) were added to support the concept of cancellation.
- The **OMP_DEFAULT_DEVICE** environment variable (see Section 4.3.8), device constructs (see Chapter 15), and the **omp_get_num_teams**, **omp_get_team_num**, **omp_set_default_device**, **omp_get_default_device**, **omp_get_num_devices**, and **omp_is_initial_device** routines (see Chapter 22 and Chapter 24) were added to support execution on devices.

- The **OMP_DISPLAY_ENV** environment variable (see Section 4.7) was added to display the value of ICVs associated with the OpenMP environment variables.
- C/C++ array syntax was extended to support array sections (see Section 5.2.5).
- The **reduction** clause (see Section 7.6.10) was extended and the **declare_reduction** construct (see Section 7.6.14) was added to support user-defined reductions.
- SIMD directives were added to support SIMD parallelism (see Section 12.4).
- Implementation defined task scheduling points for untied tasks were removed (see Section 14.14).
- The **taskgroup** construct (see Section 17.4) was added to support deep task synchronization.
- The **atomic** construct was extended to support atomic captured updates with the **capture** clause, to allow new atomic update forms, and to support sequentially consistent atomic operations with the **seq_cst** clause (see Section 17.8.1.5, Section 17.8.3.1 and Section 17.8.5).
- The **depend** clause (see Section 17.9.5) was added to support task dependences.
- Examples (previously Appendix A) were moved to a separate document.

B.8 Version 3.0 to 3.1 Differences

- The *bind-var* ICV (see Section 3.1) and the **OMP_PROC_BIND** environment variable (see Section 4.1.7) were added to support control of whether threads are bound to processors.
- The *nthreads-var* ICV was modified to be a list of the number of threads to use at each nested parallel region level (see Section 3.1) and the algorithm for determining the number of threads used in a parallel region was modified to handle a list (see Section 12.1.1).
- Data environment restrictions were changed to allow **intent(in)** and **const**-qualified types for the **firstprivate** clause (see Section 7.5.4).
- Data environment restrictions were changed to allow Fortran pointers in **firstprivate** (see Section 7.5.4) and **lastprivate** (see Section 7.5.5) clauses.
- New reduction operators **min** and **max** were added for C/C++ (see Section 7.6.3).
- The **mergeable** and **final** clauses (see Section 14.5 and Section 14.7) were added to the **task** construct (see Section 14.1) to support optimization of task data environments.
- The **taskyield** construct was added to allow user-defined task scheduling points (see Section 14.12).
- The **atomic** construct was extended to include **read**, **write**, and **capture** forms, and an **update** clause was added to apply the already existing form of the **atomic** construct (see Section 17.8.2, Section 17.8.3.1 and Section 17.8.5).

- The nesting restrictions were clarified to disallow closely nested regions within an **atomic** region so that an **atomic** region can be consistently defined with other regions to include all code in the **atomic** construct (see Section 19.1).
- The **omp_in_final** routine was added to support specialization of final task regions (see Section 23.1.3).
- Descriptions of examples (previously Appendix A) were expanded and clarified.
- Incorrect use of **omp_integer_kind** in Fortran interfaces was replaced with **selected_int_kind(8)**.

B.9 Version 2.5 to 3.0 Differences

- The concept of tasks was added to the execution model (see Section 1.2 and Chapter 2).
- The OpenMP memory model was extended to cover atomicity of memory accesses (see Section 1.3.1). The description of the behavior of **volatile** in terms of flushes was removed.
- The definition of active parallel region was changed so that a **parallel** region is active if it is executed by a team to which more than one thread is assigned (see Chapter 2).
- The definition of the *nest-var*, *dyn-var*, *nthreads-var* and *run-sched-var* ICVs were modified to provide one copy of these ICVs per task instead of one copy for the whole OpenMP program (see Section 3.1). The **omp_set_num_threads** and **omp_set_dynamic** routines were specified to support their use from inside a **parallel** region (see Section 21.1 and Section 21.7).
- The *thread-limit-var* ICV, the **OMP_THREAD_LIMIT** environment variable and the **omp_get_thread_limit** routine were added to support control of the maximum number of threads (see Section 3.1, Section 4.1.4 and Section 21.5).
- The *max-active-levels-var* ICV, the **OMP_MAX_ACTIVE_LEVELS** environment variable and the **omp_set_max_active_levels** and **omp_get_max_active_levels** routines, and were added to support control of the number of nested active parallel regions (see Section 3.1, Section 4.1.5, Section 21.12 and Section 21.13).
- The *stacksize-var* ICV and the **OMP_STACKSIZE** environment variable were added to support control of thread stack sizes (see Section 3.1 and Section 4.3.2).
- The *wait-policy-var* ICV and the **OMP_WAIT_POLICY** environment variable were added to control the desired behavior of waiting threads (see Section 3.1 and Section 4.3.3).
- Predetermined data-sharing attributes were defined for Fortran assumed-size arrays (see Section 7.1.1).
- Static class member variables were allowed in **threadprivate** directives (see Section 7.3).
- Invocations of constructors and destructors for private and threadprivate class type variables were clarified (see Section 7.3, Section 7.5.3, Section 7.5.4, Section 7.8.1 and Section 7.8.2).

- The use of Fortran allocatable arrays was allowed in **private**, **firstprivate**, **lastprivate**, **reduction**, **copyin** and **copyprivate** clauses (see Section 7.3, Section 7.5.3, Section 7.5.4, Section 7.5.5, Section 7.6.10, Section 7.8.1 and Section 7.8.2).
- Support for **firstprivate** was added to the **default** clause in Fortran (see Section 7.5.1).
- Implementations were precluded from using the storage of the original list item to hold the new list item on the primary thread for list item in the **private** clause, and the value was made well defined on exit from the **parallel** region if no attempt is made to reference the original list item inside the **parallel** region (see Section 7.5.3).
- Determination of the number of threads in **parallel** regions was updated (see Section 12.1.1).
- The assignment of logical iterations to threads in a worksharing-loop construct with a **static** schedule kind was made deterministic (see Section 13.6).
- The worksharing-loop construct was extended to support association with more than one perfectly nested loop through the **collapse** clause (see Section 13.6).
- Loop-iteration variables for worksharing-loop constructs were allowed to be random access iterators or of unsigned integer type (see Section 13.6).
- The schedule kind **auto** was added to allow the implementation to choose any possible mapping of logical iterations in a worksharing-loop constructs to threads in the team (see Section 13.6).
- The **task** construct was added to support explicit tasks (see Section 14.1).
- The **taskwait** construct was added to support task synchronization (see Section 17.5).
- The **omp_set_schedule** and **omp_get_schedule** routines were added to set and to retrieve the value of the *run-sched-var* ICV (see Section 21.9 and Section 21.10).
- The **omp_get_level** routine was added to return the number of nested parallel regions that enclose the task that contains the call (see Section 21.14).
- The **omp_get_ancestor_thread_num** routine was added to return the thread number of the ancestor thread of the current thread (see Section 21.15).
- The **omp_get_team_size** routine was added to return the size of the team to which the ancestor thread of the current thread belongs (see Section 21.16).
- The **omp_get_active_level** routine was added to return the number of active parallel regions that enclose the task that contains the call (see Section 21.17).
- Lock ownership was defined in terms of tasks instead of threads (see Chapter 28).

C Nesting of Regions

This appendix describes a set of restrictions on the nesting of regions. The restrictions on nesting are as follows:

- A **teams** region must be strictly nested either within the implicit parallel region that surrounds the whole OpenMP program or within a **target** region. If a **teams** construct is nested within a **target** construct, that **target** construct must contain no statements, declarations or directives outside of the **teams** construct (see Section 12.2).
- Only regions that are generated by **teams**-nestable constructs or **teams**-nestable routines may be strictly nested regions of **teams** regions (see Section 12.2).
- The only routines for which a call may be nested inside a region that corresponds to a construct on which the **order** clause is specified with **concurrent** as the *ordering* argument are **order-concurrent**-nestable routines (see Section 12.3).
- Only regions that correspond to **order-concurrent**-nestable constructs or **order-concurrent**-nestable routines may be strictly nested regions of regions that correspond to constructs on which the **order** clause is specified with **concurrent** as the *ordering* argument (see Section 12.3).
- The only OpenMP constructs that can be encountered during execution of a **simd** region are SIMDizable constructs (see Section 12.4).
- A team-executed region may not be closely nested inside a partitioned worksharing region, a region that corresponds to a thread-exclusive construct, or a region that corresponds to a task-generating construct that is not a team-generating construct. This follows from various restrictions requiring, in general, that team-executed regions (which include worksharing regions and barrier regions) are executed by all threads in a team or by none at all (see Chapter 13 and Section 17.3.1).
- A **distribute** region must be strictly nested inside a **teams** region (see Section 13.7).
- A **loop** region that binds to a **teams** region must be strictly nested inside a **teams** region (see Section 13.8.1).
- During execution of a **target** region, other than **target** constructs for which a **device** clause on which the *ancestor* *device-modifier* appears, device-affecting constructs must not be encountered (see Section 15.8).
- A **critical** region must not be nested (closely or otherwise) inside a **critical** region with the same *name* (see Section 17.2).

- OpenMP constructs may not be encountered during execution of an **atomic** region (see Section 17.8.5).
- An **ordered** region that corresponds to an **ordered** construct with the **threads** or **doacross** clause may not be closely nested inside a **critical**, **ordered**, **loop**, **task**, or **taskloop** region (see Section 17.10).
- If the **simd** *parallelization-level* clause is specified on an **ordered** construct, the **ordered** region must bind to a **simd** region or one that corresponds to a compound construct for which the **simd** construct is a leaf construct (see Section 17.10.2).
- If the **threads** *parallelization-level* clause is specified on an **ordered** construct, the **ordered** region must bind to a worksharing-loop region or one that corresponds to a compound construct for which a worksharing-loop construct is a leaf construct (see Section 17.10.2).
- If the **threads** *parallelization-level* clause is specified on an **ordered** construct and the binding region corresponds to a compound construct then the **simd** construct must not be a leaf construct unless the **simd** *parallelization-level* clause is also specified (see Section 17.10.2).
- If *cancel-directive-name* is **taskgroup**, the **cancel** construct must be closely nested inside a **task** construct and the **cancel** region must be closely nested inside a **taskgroup** region. Otherwise, the **cancel** construct must be closely nested inside a construct for which *directive-name* is *cancel-directive-name* (see Section 18.2).
- A **cancellation point** construct for which *cancel-directive-name* is **taskgroup** must be closely nested inside a **task** construct, and the **cancellation point** region must be closely nested inside a **taskgroup** region. Otherwise, a **cancellation point** construct must be closely nested inside a construct for which *directive-name* is *cancel-directive-name* (see Section 18.3).

D Conforming Compound Directive Names

This appendix provides the grammar from which one may derive the full list of conforming compound-directive names (see Section 19.1) after excluding any productions for compound-directive name that would violate the following constraints:

- Leaf-directive names must be unique.
- The nesting of constructs indicated by the compound construct must be conforming.
- For Fortran, where spaces are optional, the resulting compound-directive name must have unambiguous leaf-directive names (e.g., plus signs should be used to separate leaf-directive names to disambiguate **taskloop** and **task loop** as constituent-directive names).

compound-dir-name :
    composite-loop-dir-name
    parallelism-generating-combined-dir-name
    thread-selecting-combined-dir-name

composite-loop-dir-name :
    distribute-composite-dir-name
    taskloop-composite-dir-name
    worksharing-loop-composite-dir-name

parallelism-generating-combined-dir-name :
    parallel-combined-dir-name
    target-combined-dir-name
    target_data-combined-dir-name
    task-combined-dir-name
    teams-combined-dir-name

thread-selecting-combined-dir-name :
    masked-combined-dir-name
    single-combined-dir-name

distribute-composite-dir-name :
    distribute *parallel-worksharing-loop-dir-name*
    distribute *simd-dir-name*

taskloop-composite-dir-name :
  taskloop *simd-dir-name*

worksharing-loop-composite-dir-name :
  for *simd-dir-name*
  do *simd-dir-name*

parallel-combined-dir-name :
  parallel *partitioned-worksharing-dir-name*
  parallel *simd-dir-name*
  parallel *target-task-generating-dir-name*
  parallel *task-dir-name*
  parallel *taskloop-dir-name*
  parallel *thread-selecting-dir-name*

target-combined-dir-name :
  target *loop-dir-name*
  target *parallel-dir-name*
  target *simd-dir-name*
  target *task-dir-name*
  target *taskloop-dir-name*
  target *teams-dir-name*

target_data-combined-dir-name :
  target_data *loop-dir-name*
  target_data *parallel-dir-name*
  target_data *simd-dir-name*

task-combined-dir-name :
  task *loop-dir-name*
  task *parallel-dir-name*
  task *simd-dir-name*

teams-combined-dir-name :
  teams *parallel-dir-name*
  teams *partitioned-nonworksharing-workdist-dir-name*
  teams *simd-dir-name*
  teams *target-task-generating-dir-name*
  teams *task-dir-name*
  teams *taskloop-dir-name*

masked-combined-dir-name :
  masked *loop-dir-name*

masked *parallel-dir-name*
masked *simd-dir-name*
masked *target-task-generating-dir-name*
masked *task-dir-name*
masked *taskloop-dir-name*

single-combined-dir-name :
single *loop-dir-name*
single *parallel-dir-name*
single *simd-dir-name*
single *target-task-generating-dir-name*
single *task-dir-name*
single *taskloop-dir-name*

parallel-worksharing-loop-dir-name :
parallel *worksharing-loop-dir-name*

simd-dir-name :
simd

partitioned-worksharing-dir-name :
loop-dir-name
single-dir-name
worksharing-loop-dir-name
sections
workshare

target-task-generating-dir-name :
target_data-dir-name
target-dir-name
target_enter_data
target_exit_data
target_update

task-dir-name :
task-combined-dir-name
task

taskloop-dir-name :
taskloop-composite-dir-name
taskloop

thread-selecting-dir-name :

masked-dir-name
single-dir-name

loop-dir-name :
`loop`

parallel-dir-name :
parallel-combined-dir-name
`parallel`

teams-dir-name :
teams-combined-dir-name
`teams`

partitioned-nonworksharing-workdist-dir-name :
distribute-dir-name
loop-dir-name
`workdistribute`

worksharing-loop-dir-name :
worksharing-loop-composite-dir-name
`for`
`do`

single-dir-name :
single-combined-dir-name
`single`

target_data-dir-name :
target_data-combined-dir-name
`target_data`

target-dir-name :
target-combined-dir-name
`target`

masked-dir-name :
masked-combined-dir-name
`masked`

distribute-dir-name :
distribute-composite-dir-name
`distribute`

Index

D