C / C++

If a list item in a **map** clause has a base pointer and it is a scalar variable with a predetermined data-sharing attribute of firstprivate (see Section 2.20.1.1 on page 271), then on entry to the **target** region:

- If the list item is not a zero-length array section, the corresponding private variable is initialized such that the corresponding list item in the device data environment can be accessed through the pointer in the **target** region.
- If the list item is a zero-length array section , the corresponding private variable is initialized according to Section 2.20.7.2 on page 326.

C / C++

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 2.11.1 on page 135.

Events associated with the *initial task* that executes the **target** region are defined in Section 2.11.5 on page 149.

The *target-begin* event occurs when a thread enters a **target** region.

The *target-end* event occurs when a thread exits a **target** region.

The ~~*target-submit*~~ *target-submit-begin* event occurs prior to ~~creating~~ 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.

Tool Callbacks

Callbacks associated with events for *target tasks* are the same as for the **task** construct defined in Section 2.11.1 on page 135; **(***flags* **& ompt_task_target)** always evaluates to *true* in the dispatched callback.

A thread dispatches a registered **ompt_callback_target** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_target** 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. Similarly, a thread dispatches a registered **ompt_callback_target** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_target** 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. These callbacks have type signature **ompt_callback_target_t**.

A thread dispatches a registered **ompt_callback_target_submit** callback for each occurrence of a ~~*target-submit*~~ *target-submit-begin* and *target-submit-end* event in that thread. ~~The~~ Each callback has type signature **ompt_callback_target_submit_t**. Each callback receives **ompt_scope_begin** or **ompt_scope_end** as its endpoint argument, as appropriate.

Restrictions

- If a **target update**, **target data**, **target enter data**, or **target exit data** construct is encountered during execution of a **target** region, the behavior is unspecified.
- 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 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.
- At most one **defaultmap** clause for each category can appear on the directive.
- At most one **nowait** clause can appear on the directive.
- At most one **if** clause can appear on the directive.
- A *map-type* in a **map** clause must be **to**, **from**, **tofrom** or **alloc**.
- A list item that appears in an **is_device_ptr** clause must be a valid device pointer in the device data environment.
- At most one **device** clause can appear on the directive. The **device** clause expression must evaluate to a non-negative integer value less than the value of **omp_get_num_devices()** or to the value of **omp_get_initial_device()**.

C / C++

If a new list item is created then a new list item of the same type, with automatic storage duration, is allocated for the construct. The size and alignment of the new list item are determined by the static type of the variable. This allocation occurs if the region references the list item in any statement. Initialization and assignment of the new list item are through bitwise copy.

C / C++

Fortran

If a new list item is created then a new list item of the same type, type parameter, and rank is allocated. The new list item inherits all default values for the type parameters from the original list item. The value of the new list item becomes that of the original list item in the map initialization and assignment.

If the allocation status of the original list item with the **ALLOCATABLE** attribute is changed in the host device data environment and the corresponding list item is already present in the device data environment, the allocation status of the corresponding list item is unspecified until a mapping operation is performed with a **map** clause on entry to a **target**, **target data**, or **target enter data** region.

Fortran

The *map-type* determines how the new list item is initialized.

If a *map-type* is not specified, the *map-type* defaults to **tofrom**.

The **close** *map-type-modifier* is a hint to the runtime to allocate memory close to the target device.

Execution Model Events

The *target-map* event occurs when a thread maps data to or from a target device.

The ~~*target-data-op* event occurs when~~ *target-data-op-begin* event occurs before a thread initiates a data operation on a target device.

The *target-data-op-end* event occurs after a thread initiates a data operation on a target device.

Tool Callbacks

A thread dispatches a registered **ompt_callback_target_map** callback for each occurrence of a *target-map* event in that thread. The callback occurs in the context of the target task and has type signature **ompt_callback_target_map_t**.

A thread dispatches a registered **ompt_callback_target_data_op** callback for each occurrence of a ~~*target-data-op*~~ *target-data-op-begin* and *target-data-op-end* event in that thread. ~~The~~ Each callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**. Each callback receives **ompt_scope_begin** or **ompt_scope_end** as its endpoint argument, as appropriate.

Format

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(c_size_t), value :: size
integer(c_int), value  :: device_num
```

Fortran

Effect

The **omp_target_alloc** routine returns 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*, which must be greater than or equal to zero and less than the result of **omp_get_num_devices()** or the result of **omp_get_initial_device()**. When called from within a **target** region the effect of this routine is unspecified.

The **omp_target_alloc** routine returns **NULL** (or, **C_NULL_PTR**, for Fortran) if it cannot dynamically allocate the memory in the device data environment.

The device address returned by **omp_target_alloc** can be used in an **is_device_ptr** clause, Section 2.13.5 on page 170.

C / C++

Unless **unified_address** clause appears on a **requires** directive in the compilation unit, pointer arithmetic is not supported on the device address returned by **omp_target_alloc**.

C / C++

Freeing the storage returned by **omp_target_alloc** with any routine other than **omp_target_free** results in unspecified behavior.

Execution Model Events

The ~~*target-data-allocation* event occurs when a thread allocates data~~ *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 invokes a registered **ompt_callback_target_data_op** callback for each occurrence of a ~~*target-data-allocation*~~ *target-data-allocation-begin* and *target-data-allocation-end* event in that thread. ~~The~~ Each callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**. Each callback receives **ompt_scope_begin** or **ompt_scope_end** as its endpoint argument, as appropriate.

Cross References

- **target** construct, see Section 2.13.5 on page 170.
- **omp_get_num_devices** routine, see Section 3.2.36 on page 372.
- **omp_get_initial_device** routine, see Section 3.2.41 on page 376.
- **omp_target_free** routine, see Section 3.6.2 on page 398.
- **ompt_callback_target_data_op_t**, see Section 4.5.2.25 on page 492.

3.6.2 omp_target_free

Summary

The **omp_target_free** routine frees the device memory allocated by the **omp_target_alloc** routine.

Format

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(c_int), value :: device_num
```

Fortran

Constraints on Arguments

A program that calls **omp_target_free** with a non-null pointer that does not have a value returned from **omp_target_alloc** is non-conforming. The *device_num* must be greater than or equal to zero and less than the result of **omp_get_num_devices()** or the result of **omp_get_initial_device()**.

Effect

The **omp_target_free** routine frees the memory in the device data environment associated with *device_ptr*. If *device_ptr* is **NULL** (or **C_NULL_PTR**, for Fortran), the operation is ignored.

Synchronization must be inserted to ensure that all accesses to *device_ptr* are completed before the call to **omp_target_free**.

When called from within a **target** region the effect of this routine is unspecified.

Execution Model Events

The ~~*target-data-free* event occurs when a thread frees data~~ *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 invokes a registered **ompt_callback_target_data_op** callback for each occurrence of a ~~*target-data-free*~~ *target-data-free-begin* and *target-data-free-end* event in that thread. ~~The~~ Each callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**. Each callback receives **ompt_scope_begin** or **ompt_scope_end** as its endpoint argument, as appropriate.

Cross References

- **target** construct, see Section 2.13.5 on page 170.
- **omp_get_num_devices** routine, see Section 3.2.36 on page 372.
- **omp_get_initial_device** routine, see Section 3.2.41 on page 376.
- **omp_target_alloc** routine, see Section 3.6.1 on page 396.
- **ompt_callback_target_data_op_t**, see Section 4.5.2.25 on page 492.

3.6.3 omp_target_is_present

Summary

The **omp_target_is_present** routine tests whether a host pointer has corresponding storage on a given device.

Execution Model Events

The ~~*target-data-op* event occurs when~~ *target-data-op-begin* event occurs before a thread transfers data on a target device.

The *target-data-op-end* event occurs after a thread transfers data on a target device.

Tool Callbacks

A thread invokes a registered **ompt_callback_target_data_op** callback for each occurrence of a ~~*target-data-op*~~ *target-data-op-begin* and *target-data-op-end* event in that thread. ~~The~~ Each callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**. Each callback receives **ompt_scope_begin** or **ompt_scope_end** as its endpoint argument, as appropriate.

Cross References

- **target** construct, see Section 2.13.5 on page 170.
- **omp_get_initial_device** routine, see Section 3.2.41 on page 376.
- **ompt_callback_target_data_op_t**, see Section 4.5.2.25 on page 492.

3.6.5 omp_target_memcpy_rect

Summary

The **omp_target_memcpy_rect** routine copies a rectangular subvolume from a multi-dimensional array to another multi-dimensional array. The **omp_target_memcpy_rect** routine performs a copy between any combination of host and device pointers.

Format

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,
```

Execution Model Events

The ~~*target-data-op* event occurs when~~ *target-data-op-begin* event occurs before a thread transfers data on a target device.

The *target-data-op-end* event occurs after a thread transfers data on a target device.

Tool Callbacks

A thread invokes a registered **`ompt_callback_target_data_op`** callback for each occurrence of a ~~*target-data-op*~~ *target-data-op-begin* and *target-data-op-end* event in that thread. ~~The~~ Each callback occurs in the context of the target task and has type signature **`ompt_callback_target_data_op_t`**. Each callback receives **`ompt_scope_begin`** or **`ompt_scope_end`** as its endpoint argument, as appropriate.

Cross References

- **`target`** construct, see Section 2.13.5 on page 170.
- **`omp_get_initial_device`** routine, see Section 3.2.41 on page 376.
- **`ompt_callback_target_data_op_t`**, see Section 4.5.2.25 on page 492.

3.6.6 `omp_target_memcpy_async`

Summary

The **`omp_target_memcpy_async`** routine asynchronously performs a copy between any combination of host and device pointers.

Format

```
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
);
```

Constraints on Arguments

Each device pointer specified must be valid for the device on the same side of the copy. The *dst_device_num* and *src_device_num* arguments must be greater than or equal to zero and less than the result of **`omp_get_num_devices()`** or equal to the result of **`omp_get_initial_device()`**.

Effect

This routine performs an asynchronous memory copy where *length* bytes of memory at offset *src_offset* from *src* in the device data environment of device *src_device_num* are copied to *dst* starting at offset *dst_offset* in the device data environment of device *dst_device_num*. Logically the **`omp_target_memcpy_async`** routine generates a target task with an implicit nowait. Task dependecies are expressed with zero or more **`omp_depend_t`** objects. The dependencies are specified by passing the number of **`omp_depend_t`** objects followed by an array of **`omp_depend_t`** objects. The generated target task is not a dependent task if the program passes in a count of zero and value of **`NULL`** for *depobj_count* and *depobj_list*, respectively.

The routine returns zero if successful. Otherwise, it returns a non-zero value. The routine contains a task scheduling point.

Execution Model Events

The ~~*target-data-op* event occurs when~~ *target-data-op-begin* event occurs before a thread transfers data on a target device.

The *target-data-op-end* event occurs after a thread transfers data on a target device.

Tool Callbacks

A thread invokes a registered **`ompt_callback_target_data_op`** callback for each occurrence of a ~~*target-data-op*~~ *target-data-op-begin* and *target-data-op-end* event in that thread. ~~The~~ Each callback occurs in the context of the target task and has type signature **`ompt_callback_target_data_op_t`**. Each callback receives **`ompt_scope_begin`** or **`ompt_scope_end`** as its endpoint argument, as appropriate.

Cross References

- **`target`** construct, see Section 2.13.5 on page 170.
- **`Depend objects`**, see Section 2.18.10 on page 255.
- **`omp_get_initial_device`** routine, see Section 3.2.41 on page 376.
- **`ompt_callback_target_data_op_t`**, see Section 4.5.2.25 on page 492.

omp_target_memcpy_rect_async routine generates a target task with an implicit nowait. Task dependecies are expressed with zero or more **omp_depend_t** objects. The dependencies are specified by passing the number of **omp_depend_t** objects followed by an array of **omp_depend_t** objects. The generated target task is not a dependent task if the program passes in a count of zero and value of **NULL** for *depobj_count* and *depobj_list*, respectively.

The routine returns zero if successful. Otherwise, it returns a non-zero value. The routine contains a task scheduling point.

When called from within a **target** region the effect of this routine is unspecified.

An application can determine the number of inclusive dimensions supported by an implementation by passing **NULL** pointers 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.

Execution Model Events

The ~~*target-data-op* event occurs when~~ *target-data-op-begin* event occurs before a thread transfers data on a target device.

The *target-data-op-end* event occurs after a thread transfers data on a target device.

Tool Callbacks

A thread invokes a registered **ompt_callback_target_data_op** callback for each occurrence of a ~~*target-data-op*~~ *target-data-op-begin* and *target-data-op-end* event in that thread. ~~The~~ Each callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**. Each callback receives **ompt_scope_begin** or **ompt_scope_end** as its endpoint argument, as appropriate.

Cross References

- **target** construct, see Section 2.13.5 on page 170.
- **Depend objects**, see Section 2.18.10 on page 255.
- **omp_get_initial_device** routine, see Section 3.2.41 on page 376.
- **ompt_callback_target_data_op_t**, see Section 4.5.2.25 on page 492.

3.6.8 omp_target_associate_ptr

Summary

The **omp_target_associate_ptr** routine maps a device pointer, which may be returned from **omp_target_alloc** or implementation-defined runtime routines, to a host pointer.

share underlying storage will result in unspecified behavior. The **omp_target_is_present** function can be used to test whether a given host pointer has a corresponding variable in the device data environment.

Execution Model Events

The ~~*target-data-associate* event occurs when a thread associates data~~ *target-data-associate-begin* event occurs before a thread initiates a device pointer association on a target device.

The *target-data-associate-end* event occurs after a thread initiates a device pointer association on a target device.

Tool Callbacks

A thread invokes a registered **ompt_callback_target_data_op** callback for each occurrence of a ~~*target-data-associate*~~ *target-data-associate-begin* and *target-data-associate-end* event in that thread. ~~The~~ Each callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**. Each callback receives **ompt_scope_begin** or **ompt_scope_end** as its endpoint argument, as appropriate.

Cross References

- **target** construct, see Section 2.13.5 on page 170.
- **map** clause, see Section 2.20.7.1 on page 317.
- **omp_target_alloc** routine, see Section 3.6.1 on page 396.
- **omp_target_disassociate_ptr** routine, see Section 3.6.8 on page 407.
- **ompt_callback_target_data_op_t**, see Section 4.5.2.25 on page 492.

3.6.9 omp_target_disassociate_ptr

Summary

The **omp_target_disassociate_ptr** removes the associated pointer for a given device from a host pointer.

Format

C / C++

```
int omp_target_disassociate_ptr(const void *ptr, int device_num);
```

C / C++

Fortran

```
integer(c_int) function omp_target_disassociate_ptr(ptr, &
  device_num) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_int
type(c_ptr), value :: ptr
integer(c_int), value :: device_num
```

Fortran

Constraints on Arguments

The *device_num* must be greater than or equal to zero and less than the result of **omp_get_num_devices()** or equal to the result of a call to **omp_get_initial_device()**.

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** (or **C_NULL_PTR**, for Fortran) 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.

When called from within a **target** region the effect of this routine is unspecified.

The routine returns zero if successful. Otherwise it returns a non-zero value.

After a call to **omp_target_disassociate_ptr**, the contents of the device buffer are invalidated.

Execution Model Events

The ~~*target-data-disassociate* event occurs when a thread disassociates data~~ *target-data-disassociate-begin* event occurs before a thread initiates a device pointer disassociation on a target device.

The *target-data-disassociate-end* event occurs after a thread initiates a device pointer disassociation on a target device.

Tool Callbacks

A thread invokes a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-disassociate-begin* and *target-data-disassociate-end* event in that thread. Each callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**. Each callback receives **ompt_scope_begin** or **ompt_scope_end** as its endpoint argument, as appropriate.

Cross References

- **target** construct, see Section 2.13.5 on page 170.
- **omp_target_associate_ptr** routine, see Section 3.6.8 on page 407.
- **ompt_callback_target_data_op_t**, see Section 4.5.2.25 on page 492.

3.7 Memory Management Routines

This section describes routines that support memory management on the current device.

Instances of memory management types must be accessed only through the routines described in this section; programs that otherwise access instances of these types are non-conforming.

3.7.1 Memory Management Types

The following type definitions are used by the memory management routines:

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_alloctrait_key_t;

typedef enum omp_alloctrait_value_t {
  omp_atv_false = 0,
  omp_atv_true = 1,
  omp_atv_default = 2,
```

can manage traces associated with the device. One allocates a buffer in which the device can deposit trace events. The second callback processes a buffer of trace events from the device.

- If the device requires a trace buffer, the OpenMP implementation invokes the tool-supplied callback function on the host device to request a new buffer.

- The OpenMP implementation 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 operation that the host initiated to cause these activities. To correlate activities on the host with activities on a device, a tool can register a **ompt_callback_target_submit** callback. Before and after the host initiates each distinct activity creation of an initial task on a device associated with a structured block for a **target** construct on a device, the OpenMP implementation dispatches the **ompt_callback_target_submit** callback on the host in the thread that is executing the task that encounters the **target** construct. Examples of activities that could cause an **ompt_callback_target_submit** callback to be dispatched include an explicit data copy between a host and target device or execution of a computation. This callback provides the tool with a pair of identifiers: one that identifies the target region and a second that uniquely identifies an activity 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 completion callback to process a non-empty sequence of records in a trace buffer that is associated with the device.

- The tool-supplied buffer completion callback may return immediately, ignoring records in the trace buffer, or it may iterate through them using the **ompt_advance_buffer_cursor** entry point to inspect each record. A tool may use the **ompt_get_record_type** runtime entry point to inspect the type of the record at the current cursor position. Three runtime entry points (**ompt_get_record_ompt**, **ompt_get_record_native**, and **ompt_get_record_abstract**) allow tools to inspect the contents of some or all records in a trace buffer. The **ompt_get_record_native** runtime entry point uses the native trace format of the device. The **ompt_get_record_abstract** runtime entry point decodes the contents of a native trace record and summarizes them as an **ompt_record_abstract_t** record. The **ompt_get_record_ompt** runtime entry point can only be used to retrieve records in OMPT format.

- Once tracing has been started on a device, a tool may pause or resume tracing on the device at any time by invoking **ompt_pause_trace** with an appropriate flag value as an argument.

- A tool may invoke the **ompt_flush_trace** runtime entry point for a device at any time between device initialization and finalization to cause the device to flush pending trace records.

- At any time, a tool may use the **ompt_start_trace** runtime entry point to start tracing or the **ompt_stop_trace** runtime entry point to stop tracing on a device. When tracing is stopped on a device, the OpenMP implementation eventually gathers all trace records already collected on the device and presents them to the tool using the buffer completion callback.

Description of Arguments

The *device_num* argument indicates the device which the buffer contains events.

The *buffer* argument is the address of a buffer that was previously allocated by a *buffer request* callback.

The *bytes* argument indicates the full size of the buffer.

The *begin* argument is an opaque cursor that indicates the position of the beginning of the first record in the buffer.

The *buffer_owned* argument is 1 if the data to which the buffer points can be deleted by the callback and 0 otherwise. If multiple devices accumulate trace 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* = 0. In this case, the callback may not delete the buffer.

Cross References

- **ompt_buffer_t** type, see Section 4.4.4.7 on page 444.
- **ompt_buffer_cursor_t** type, see Section 4.4.4.8 on page 445.

4.5.2.25 ompt_callback_target_data_op_t

Summary

The **ompt_callback_target_data_op_t** type is used for callbacks that are dispatched when a thread maps data to a device.

Format

C / C++

```
typedef void (*ompt_callback_target_data_op_t) (
  ompt_scope_endpoint_t endpoint,
  ompt_id_t target_id,
  ompt_id_t host_op_id,
  ompt_target_data_op_t optype,
  void *src_addr,
  int src_device_num,
  void *dest_addr,
  int dest_device_num,
  size_t bytes,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_target_data_op_t {
  ompt_id_t host_op_id;
  ompt_target_data_op_t optype;
  void *src_addr;
  int src_device_num;
  void *dest_addr;
  int dest_device_num;
  size_t bytes;
  ompt_device_time_t end_time;
  const void *codeptr_ra;
} ompt_record_target_data_op_t;
```

C / C++

Description

A registered **ompt_callback_target_data_op** callback is dispatched when device memory is allocated or freed, as well as when data is copied to or from a device.

Note – An OpenMP implementation may aggregate program variables and data operations upon them. For instance, an OpenMP implementation may synthesize a composite to represent multiple scalars and then allocate, free, or copy this composite as a whole rather than performing data operations on each scalar individually. Thus, callbacks may not be dispatched as separate data operations on each variable.

Description of Arguments

The *endpoint* argument indicates that the callback signals the beginning of a scope or the end of a scope.

The *host_op_id* argument is a unique identifier for a data ~~operations~~ operation on a target device.

The *optype* argument indicates the kind of data ~~mapping~~operation.

The *src_addr* argument indicates the data address before the operation, where applicable.

The *src_device_num* argument indicates the source device number for the data operation, where applicable.

The *dest_addr* argument indicates the data address after the operation.

The *dest_device_num* argument indicates the destination device number for the data operation.

4.5.2.28 `ompt_callback_target_submit_t`

Summary

The **`ompt_callback_target_submit_t`** type is used for callbacks that are dispatched ~~when~~ before and after the host initiates creation of an initial task ~~is created~~ on a device.

Format

C / C++

```
typedef void (*ompt_callback_target_submit_t) (
  ompt_scope_endpoint_t endpoint,
  ompt_id_t target_id,
  ompt_id_t host_op_id,
  unsigned int requested_num_teams
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_target_kernel_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_kernel_t;
```

C / C++

Description

A thread dispatches a registered **`ompt_callback_target_submit`** callback on the host ~~when~~ before and after a target task ~~creates~~ initiates creation of an initial task on a ~~target~~ device.

Description of Arguments

The *endpoint* argument indicates that the callback signals the beginning of a scope or the end of a scope.

The *target_id* argument is a unique identifier for the associated target region.

The *host_op_id* argument is a unique identifier for the initial task on the target device.

The *requested_num_teams* argument is the number of teams that the host requested to execute the kernel. The actual number of teams that execute the kernel may be smaller and generally will not be known until the kernel begins to execute on the device.

If **ompt_set_trace_ompt** has configured the device to trace kernel execution then the device will log a **ompt_record_target_kernel_t** record in a trace. The fields in the record are as follows:

- The *host_op_id* field contains a unique identifier that can be used to correlate a **ompt_record_target_kernel_t** record with its associated **ompt_callback_target_submit** callback on the host;
- The *requested_num_teams* field contains the number of teams that the host requested to execute the kernel;
- The *granted_num_teams* field contains the number of teams that the device actually used to execute the kernel;
- The time when the initial task began execution on the device is recorded in the *time* field of an enclosing **ompt_record_t** structure; and
- The time when the initial task completed execution on the device is recorded in the *end_time* field.

Cross References

- **target** construct, see Section 2.13.5 on page 170.
- **ompt_id_t** type, see Section 4.4.4.3 on page 442.
- **ompt_scope_endpoint_t** type, see Section 4.4.4.11 on page 446.

4.5.2.29 ompt_callback_control_tool_t

Summary

The **ompt_callback_control_tool_t** type is used for callbacks that dispatch *tool-control* events.