OMPT and OMPD: OpenMP Tools Application Programming Interfaces for Performance Analysis and Debugging

Alexandre Eichenberger[*], John Mellor-Crummey[†], Martin Schulz[‡],

Nawal Copty[§], John DelSignore[¶], Robert Dietrich[‖], Xu Liu[**], Eugene Loh[††], Daniel Lorenz[‡‡]
and other members of the OpenMP Tools Working Group

April 24, 2013

1 Introduction

To enable portable tools for performance analysis and debugging of OpenMP programs, we define an application programming interface (API) for tools that we propose for adoption as part of the OpenMP standard and supported by all OpenMP compliant implementation.

There are two parts to the proposed interface: OMPT—a first-party API for performance tools, and OMPD—a shared-library plugin for debuggers that enables a debugger to inspect and control execution of an OpenMP program.

1.1 OMPT

The design of OMPT is based on experience with two prior efforts to define a standard OpenMP tools API: the POMP API [5] and the Sun/Oracle Collector API [3, 4]. POMP is geared toward trace-based measurements, which has the shortcoming that its overhead can be significant because operations to be traced, e.g., an iteration of an OpenMP work-sharing loop, can take less time than recording an event in a trace. As an alternative to POMP's trace-based approach, the Sun/Oracle Collector API was designed primarily to support measurement and attribution of performance information using asynchronous sampling of call stacks. This sampling-based design enables construction of tools that attribute costs to full calling contexts without the drawbacks of tracing; namely, tools can record compact profiles with low runtime overhead. However, not all events can be traced prohibiting the implementation of subset of tools, such as trace analyzers or verification tools. OMPT builds on ideas of both POMP and the Sun/Oracle collector API to support asynchronous sampling and optional trace event generation and extends them with support for *blame shifting* [7, 8] which shifts attribution of costs from symptoms to causes. The OMPT interface can be implemented either in entirely in the compiler or entirely the OpenMP runtime system, as well as using a hybrid compiler/runtime option.

Most routines described in the OMPT API are intended only for use by tools rather than for direct use by applications. As a result, all OMPT API functions have a C binding only. A Fortran binding is provided only for a few application-facing inquiry and control functions, described in Section 6.

[*]IBM T.J. Watson Research Center
[†]Rice University
[‡]LLNL
[§]Oracle
[¶]Rogue Wave
[‖]TU Dresden, ZIH
[**]Rice University
[††]Oracle
[‡‡]Juelich Supercomputer Center

1.1.1 OMPT Design Objectives

OMPT tries to satisfy several design objectives for a performance tool interface for OpenMP. These objectives are listed in decreasing order of importance.

- The API should enable tools to gather sufficient information about an an OpenMP program executing under control of an OpenMP runtime system to associate costs with the program and the runtime system.
  - The API should provide an interface sufficient to construct low-overhead performance tools based on asynchronous sampling.
  - The API should enable a profiler that uses call stack unwinding to identify which frames in its call stack correspond to routines in the OpenMP runtime system.
  - An OpenMP runtime system should associate the activity of a thread at any point in time with a *state*, e.g., idle, which will enable a performance tool to interpret program execution behavior.
  - Certain API routines must be defined as thread-safe so that they can be invoked in a signal handler by a profiler as part of processing asynchronous events, e.g., handling sample events.
- Incorporating support for the API in an OpenMP runtime system should add negligible overhead to an OpenMP runtime system if the interface is not in use by a tool.
- The API should define support for trace based performance tools.
- Adding the API to an OpenMP implementation must not impose an unreasonable development burden on implementer.
- The API should not impose an unreasonable development burden on tool implementers.

1.1.2 OMPT Interface

To support the OMPT interface for tools, an OpenMP runtime system has two responsibilities: maintain information about the state of each OpenMP thread and provide a set of API calls that tools can use to interrogate the OpenMP runtime. Maintaining information about the state of each thread in the runtime system is not free and thus an OpenMP runtime system need not maintain state information unless a tool has registered itself, an environment variable directed the tool to track runtime state, or a debugger has demanded that runtime state information be maintained. Without any explicit request for tool support to be enabled, an OpenMP runtime need not maintain any information to support tools and may provide trivial (and thus, perhaps useless) answers to any API queries.

1.2 OMPD

A common idiom has emerged to support the manipulation of a programming abstraction by debuggers: the programming abstraction provides a plugin library that the debugger loads into its own address space. The debugger then uses an API provided by the plugin library to inspect and manipulate state associated with the programming abstraction in a target. The target may be a live process or a core file. Such plugin libraries have been defined to support debugging of threads [6] and MPI [2]. A 2003 paper describes a previous effort to define a debugging support library for OpenMP [1].

1.2.1 OMPD Design Objectives

The design for OMPD tries to satisfy several design objectives for a performance tool interface for OpenMP. These objectives are listed in decreasing order of importance.

- The API should enable a debugger to inspect the state of a live process or a core file.
  - The API should provide the debugger with third-party versions of the OpenMP runtime inquiry functions.

  - The API should provide the debugger with third-party versions of the OMPT inquiry functions.
- The API should facilitate interactive control of a live process in the following ways:
  - Help a debugger place breakpoints to intercept the beginning and end of parallel regions and task regions.
  - Help a debugger identify the first program instruction that the OpenMP runtime will execute in a parallel region or a task region so that it can set breakpoints inside the regions.
- Adding the API to an OpenMP implementation must not impose an unreasonable development burden on implementer.
- The API should not impose an unreasonable development burden on tool implementers.

1.3 Document Roadmap

The document first outlines aspects of the OMPT tools API. Section 2 describes state information maintained by the OpenMP runtime system for use by tools. Section 3 describes callback events for tools supported by the OpenMP runtime system. Section 4 describes tool data structures. Section 5 describes runtime system inquiry operations for tools. Section 6 describes runtime system inquiry and control operations available to applications. Section 7 describes interfaces for tool initialization. Section 8 describes the OMPD interface, which provides a superset of the first-party OMPT support, in the form of a debugger plugin that supports third-party inspection and control of a target process. Section 9 describes global variables provided by the OpenMP runtime to support OMPD.

2 Runtime State

To enable a tool to understand what an OpenMP thread is doing, when tools support has been turned on, an OpenMP runtime will maintain state information for each OpenMP (master, worker, or idle) thread that can be queried by a tool. The state maintained for each thread by the OpenMP runtime is an approximation of the thread's instantaneous state. When a thread not associated with the OpenMP runtime queries its state, the runtime returns `ompt_state_undefined`.

To enable low overhead implementations, an OpenMP runtime has some flexibility as to if and when it must report thread state transitions. For example, consider when a thread acquires a lock. One compliant runtime may transition the thread state to `ompt_state_wait_lock` early before attempting to acquire a lock. Another compliant runtime may transition a thread state to `ompt_state_wait_lock` late only if the thread begins to spin or block to wait for an unavailable lock. A third compliant runtime may transition the state to `ompt_state_wait_lock` even later - only after a thread waits for a significant amount of time.

Each thread maintains not only a state but also an `ompt_wait_id_t` identifier. When a thread is waiting for a lock, critical region, ordered, or atomic, and the thread is in the corresponding wait state, then the thread's `wait_id` field must point to the lock, critical region identifier, or atomic variable upon which the thread is waiting. A thread's `wait_id` is meaningless if the thread is not in a wait state.

State values 0 to 127 are reserved for current OMPT states and future extensions.

Idle State

`ompt_state_idle`

A thread is idle while waiting to work on an OpenMP parallel region.

Work States

`ompt_state_work_serial`

A thread executing "useful" work outside all parallel regions. Any thread in existence prior to OpenMP initialization is initially reported in this state.

`ompt_state_work_parallel`

A thread executing "useful" work inside a parallel region.

`ompt_state_work_reduction`

A thread working to combine reduction results. A compliant runtime might never have a thread enter this state; a thread performing a reduction is allowed to be in state `ompt_state_work_parallel` or `ompt_state_overhead`.

Wait States (Non Mutex)

`ompt_state_wait_barrier`

A thread waiting at an (implicit/explicit) barrier construct. A compliant implementation may have a thread enter this state early, when the thread encounters a barrier, or late, when the thread begins to wait at the barrier.

`ompt_state_wait_taskwait`

A thread waiting at a taskwait construct. A compliant implementation may have a thread enter this state early, when the thread encounters a taskwait construct, or late, when the thread begins to wait for an uncompleted task.

`ompt_state_wait_taskgroup`

A thread waiting at a taskgroup construct. A compliant implementation may have a thread enter this state early, when the thread encounters the end of a taskgroup construct, or late, when the thread begins to wait for an uncompleted task.

Wait States (Mutex)

`ompt_state_wait_lock` (`ompt_state_wait_nest_lock`)

A thread waiting for a (nest) lock. A compliant implementation may have a thread enter this state early, when a thread encounters a (nest) lock set routine, or late, when the thread begins to wait for a (nest) lock.

Before a thread enters this state, the OpenMP runtime system will update the thread's `ompt_wait_id_t` field to contain the address of the (nest) lock being acquired.

`ompt_state_wait_critical`

A thread waiting to enter a critical construct. A compliant implementation may have a thread enter this state early, when a thread encounters a critical construct, or late, when the thread begins to wait to enter the critical construct. A compliant implementation may report a thread waiting to enter a critical region as waiting for a lock associated with the region.

Before a thread enters this state, the OpenMP runtime system will update the thread's `ompt_wait_id_t` field to contain an address (e.g., a lock) associated with the critical region.

`ompt_state_wait_atomic`

A thread waiting to enter an atomic construct. A compliant implementation may have a thread enter this state early, when encountering an atomic construct, or late, when the thread begins to wait to enter the atomic construct. A compliant implementation may report waiting at an atomic region as waiting for a corresponding lock. A compliant implementation may opt to not report this state, for example, when using atomic hardware instructions.

Before a thread enters this state, the OpenMP runtime system will update the thread's `ompt_wait_id_t` field to contain the address of the atomic variable `ompt_state_wait_ordered`

A thread waiting to enter an ordered construct. A compliant implementation may have a thread enter this state early, when encountering an ordered construct, or late, when the thread begins to wait at the ordered construct.

Before a thread enters this state, the OpenMP runtime system will update the thread's `ompt_wait_id_t` field to contain the address of a variable associated with the ordered construct. The variable may be the parallel loop index variable or may be a distinct runtime internal variable.

Overhead State

`ompt_state_overhead`

A thread may report the overhead state while preparing for a parallel region, preparing for a new explicit task, preparing for a worksharing region, preparing for computing loop iterations, or performing some other work inside a parallel region. It is compliant to report some or all OpenMP runtime overhead as work.

Miscellaneous States

`ompt_state_undefined`

This state is reserved for threads that are not user threads, initial threads, threads currently in an OpenMP team, or threads waiting to become part of an OpenMP team.

`ompt_state_first`

This state is a placeholder exclusively reserved for use by the OMPT runtime call `ompt_enumerate_state` (see Section 5.1), which can be used to query all available runtime states. A thread will never be in this state as an OpenMP program executes.

`ompt_state_last`

This state is a placeholder exclusively reserved for use by the OMPT runtime call `ompt_enumerate_state` (see Section 5.1), which can be used to query all available runtime states. A thread will never be in this state as an OpenMP program executes.

3 Events

This section describes callback events that an OpenMP runtime may provide for use by a tool. Each callback has a particular type signatures defined for it.

There are two classes of events: mandatory events and optional events. Mandatory events must be implemented in any compliant runtimes implementations. Optional events are grouped in sets of related events. While each event can be individually included or omitted, we encourage tools to consider implementing all or none of the events in a given set.

A callback need not be registered for an event. An OpenMP runtime system will not make any callback unless a tool has registered to receive it.

3.1 Mandatory Events

The following events are mandatory and must be supported by a compliant OpenMP runtime system.

Threads

`ompt_event_thread_create`

The OpenMP runtime invokes this callback after a new thread is created and fully initialized but before the thread is used by any OpenMP tasks. The callback executes in the execution environment of the thread. This callback has type signature `ompt_thread_callback_t`.

`ompt_event_thread_exit`

If the thread's `ompt_thread_t` value field is non-zero, the OpenMP runtime invokes this callback after it completes of all its tasks and before the thread is destroyed. The callback executes in the execution environment of the thread. This callback has type signature `ompt_thread_callback_t`.

Parallel Regions

`ompt_event_parallel_create`

The OpenMP runtime invokes this callback after the parallel region is fully initialized and before team threads execute the parallel region work. The callback executes in the context of the parent thread. This callback has type signature `ompt_new_parallel_callback_t`,

`ompt_event_parallel_exit`

The OpenMP runtime invokes this callback after the parallel region executes its closing synchronization barrier and before resuming execution of the parent task. The callback executes in the context of the parent thread. This callback has type signature `ompt_new_parallel_callback_t`.

Tasks

`ompt_event_task_create`

The OpenMP runtime invokes this callback after the parent task creates a new explicit task and before the new task or executes. The callback executes in the execution environment of the parent task. This callback has type signature `ompt_new_task_callback_t`.

`ompt_event_task_exit`

If the task's `ompt_data_t` value field is non-zero, the OpenMP runtime invokes this callback after the explicit task completes and before the thread resumes execution of another task. The callback executes in the execution environment of an unspecified ancestor task; its `task_data` parameter points to the `ompt_data_t` structure of the exited task. This callback has type signature `ompt_task_callback_t`.

Application Tool Control

`ompt_event_control`

If the user program calls `ompt_control`, the OpenMP runtime invokes this callback. The callback executes in the environment of the user control call; the parameters for the callback are the ones passed by the user to `ompt_control`. This callback has type signature `ompt_control_callback_t`.

Termination

`ompt_event_runtime_shutdown`

The OpenMP runtime system invokes this callback before it shut down the runtime system. This callback allows the tool to clean up its state and report its data as needed. It is possible for a runtime to restart at some later time, in which case it may call the initializer callback again. This callback has type signature `ompt_callback_t`.

3.2 Optional Events

This section describes two sets of events. One set of events is used by sampling-based performance tools that employ a strategy known as blame shifting to attribute waiting to activity in contexts that cause other threads to wait rather than the contexts in which the waiting is observed.

Supporting any of the events in this section is optional for a compliant OpenMP runtime system.

3.2.1 Events for Blame Shifting

This section describes synchronous events used by sampling-based performance tools that employ 'blame shifting' to transfer blame for waiting from contexts where waiting is observed to code responsible for the waiting.[1] Using these callbacks, a tool employing blame shifting can attribute time a thread spends waiting for a lock to the context of the lock holder. Similarly, time threads spend waiting at a barrier can be attributed back to to code being executed by working threads while other threads wait.

Idle State Entry/Exit

`ompt_event_idle_begin`

The OpenMP runtime invokes this callback when starting to idle outside a parallel region. The callback executes in the environment of the thread. If this callback is registered, the callback for `ompt_event_idle_end` must also be registered. This callback has type signature `ompt_thread_callback_t`.

`ompt_event_idle_end`

The OpenMP runtime invokes this callback when a thread finishes idling outside a parallel region. The callback executes in the environment of the thread. If this callback is registered, the callback for `ompt_event_idle_begin` must also be registered. This callback has type signature `ompt_thread_callback_t`.

Barrier Idling

`ompt_event_wait_barrier_begin`

The OpenMP runtime invokes this callback when a thread starts to wait at a barrier. One barrier may generate multiple pairs of barrier begin and end callbacks, e.g., if waiting at the barrier occurs in multiple stages. The callback executes in the environment of the task. If this callback is registered, the callback for `ompt_event_wait_barrier_end` must also be registered. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_wait_barrier_end`

The OpenMP runtime invokes this callback when a thread finishes waiting at a barrier. One barrier may generate multiple pairs of barrier begin and end callbacks, e.g., if waiting at the barrier occurs in multiple stages. The callback executes in the environment of the task. If this callback is registered, the callback for `ompt_event_wait_barrier_begin` must also be registered. This callback has type signature `ompt_parallel_callback_t`.

Taskwait Idling

`ompt_event_wait_taskwait_begin`

The OpenMP runtime invokes this callback when a thread starts to wait at a taskwait. One taskwait may generate multiple pairs of taskwait begin and end callbacks. This callback executes in the environment of the task. If this callback is defined, the callback for `ompt_event_wait_taskwait_end` must also be defined. This callback has type signature `ompt_parallel_callback_t`.

[1] Blame shifting has previously been demonstrated to be effective for attributing costs associated with threads idling while waiting to steal work in a work-stealing runtime [7], and spin waiting to acquire a lock [8]

`ompt_event_wait_taskwait_end`

The OpenMP runtime invokes this callback when a task finishes waiting at a taskwait. One taskwait may generate multiple pairs of taskwait begin and end callbacks. This callback executes in the environment of the task. If this callback is defined, the callback for `ompt_event_wait_taskwait_begin` must also be defined. This callback has type signature `ompt_parallel_callback_t`.

Taskgroup Idling

`ompt_event_wait_taskgroup_begin` The OpenMP runtime invokes this callback when a task starts to wait at a taskgroup. One taskgroup may generate multiple pairs of taskgroup begin and end callbacks. This callback executes in the environment of the task. If this callback is defined, the callback for `ompt_event_wait_taskgroup_end` must also be defined. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_wait_taskgroup_end`

The OpenMP runtime invokes this callback when a task finishes waiting as a taskgroup ends. One taskgroup may generate multiple pairs of taskgroup begin and end callbacks. This callback executes in the environment of the task. If this callback is registered, the callback for `ompt_event_wait_taskgroup_begin` must also be registered. This callback has type signature `ompt_parallel_callback_t`.

Lock Release

`ompt_event_release_lock`

The OpenMP runtime system invokes this callback after a task releases a lock. This callback executes in the environment of the task; its `wait_id` parameter identifies the released lock. This callback has type signature `ompt_wait_callback_t`.

`ompt_event_release_nest_lock_last`

The OpenMP runtime invokes this callback for certain releases of a nest lock. If a task acquires a nest lock n times, this callback occurs only after the nth release. The inner n-1 releases are handled by `ompt_event_release_nest_lock_prev` events. This callback executes in the environment of the task; its `wait_id` parameter identifies the nest lock released. This callback has type signature `ompt_wait_callback_t`.

Critical Release

`ompt_event_release_critical`

The OpenMP runtime system invokes this callback after a task exits a critical region. This callback executes in the environment of the task; its `wait_id` parameter identifies the critical region being exited. This callback has type signature `ompt_wait_callback_t`.

Ordered Release

`ompt_event_release_ordered`

The OpenMP runtime system invokes this callback after a task completes an ordered region. This callback executes in the environment of the task; its `wait_id` parameter identifies a variable associated with the ordered construct. This callback has type signature `ompt_wait_callback_t`.

Atomic Release

`ompt_event_release_atomic`

The OpenMP runtime system invokes this callback after a task completes an atomic region. This callback executes in the environment of the task; its `wait_id` parameter identifies the atomic data being computed upon.

If an atomic block is implemented using a hardware instruction, then an OpenMP runtime may choose to never report this event. However, if an atomic region is implemented using any mechanism that might involve spinning in software, then an OpenMP runtime developer should consider reporting this event if the time or effort a thread invests in waiting or retries exceeds a constant threshold defined by the developer. Examples of spinning in software include spin waiting on a critical section used to implement atomics, or retrying atomic operations implemented using hardware primitives that may fail. Examples of hardware primitives that could fail with explicit retries in software include transactional instructions, load-linked/store-conditional, and compare-and-swap.

This callback has type signature `ompt_wait_callback_t`.

3.2.2 Events for Trace-based Measurement Tools

The following are synchronous events to support trace-based measurement of tasks.

Task Creation and Destruction

`ompt_event_implicit_task_create`

The OpenMP runtime system invokes this callback, after an implicit task is fully initialized and before the task executes its work. This callback executes in the context of the implicit task. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_implicit_task_exit`

If the implicit task's `ompt_data_t` value field is non-zero, the OpenMP runtime system invokes this callback, which has type signature `ompt_parallel_callback_t`, after the implicit task executes its closing synchronization barrier, and before returning to idle or the task is destroyed. The callback executes in the context of the implicit task.

`ompt_event_task_switch`

The OpenMP runtime system invokes this callback after it suspends one task and before it resumes another task. This callback has type signature `ompt_task_switch_callback_t`. This callback executes in the environment of the resumed task. If the suspended task actually completed and its data structure was deallocated, the `suspended_task_data` parameter is NULL.

Lock Creation and Destruction

`ompt_event_init_lock` (`ompt_event_init_nest_lock`)

The OpenMP runtime system invokes this callback just after this task initializes the (nest) lock. This callback executes in the environment of the task; its `wait_id` parameter identifies the lock. This callback has type signature `ompt_wait_callback_t`.

`ompt_event_destroy_lock` (`ompt_event_destroy_nest_lock`)

The OpenMP runtime system invokes this callback just before this task destroys the (nest) lock. This callback executes in the environment of the task; its `wait_id` parameter identifies the lock. This callback has type signature `ompt_wait_callback_t`.

Loops

`ompt_event_loop_begin`

The OpenMP runtime system invokes this callback after the parallel loop is initialized for this thread and before this thread executes a first loop iteration. This callback executes in the context of the task. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_loop_end`

The OpenMP runtime system invokes this callback after the last loop iteration for this thread executes and before this thread executes the loop barrier (wait) or the statement following the loop (nowait). This callback executes in the context of the task. This callback has type signature `ompt_parallel_callback_t`.

Sections

`ompt_event_section_begin`

The OpenMP runtime system invokes this callback after a parallel section is initialized for this thread and before this thread executes a first section. This callback executes in the context of the task. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_section_end`

The OpenMP runtime system invokes this callback after the last section for this thread is executed and before this thread executes the section barrier (wait) or the statement following the section construct (nowait). This callback executes in the context of the task. This callback has type signature `ompt_parallel_callback_t`.

Single Blocks

`ompt_event_single_in_block_begin`

The OpenMP runtime system invokes this callback after the single construct is initialized for this thread and before this thread executes the code block of the single region. This callback executes in the context of the task. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_single_in_block_end`

The OpenMP runtime system invokes this callback after this thread executes the code code block of the single region and before this thread executes the single barrier (wait) or the statement following the single construct (nowait). This callback executes in the context of the task. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_single_others_begin`

The OpenMP runtime system invokes this callback after the single construct is initialized for this thread and before this thread would have executed the code block of the single region if this thread had been selected. This callback executes in the context of the task. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_single_others_end`

The OpenMP runtime system invokes this callback after this thread would have executed the code block of the single region if this thread had been selected and before this thread executes the single barrier (wait) or the statement following the single construct (nowait). This callback executes in the context of the task. This callback has type signature `ompt_parallel_callback_t`.

Master Blocks

`ompt_event_master_begin`

The OpenMP runtime system invokes this callback after the master section is initialized for this thread and before this thread executes the master code. This callback executes in the context of the master task. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_master_end`

The OpenMP runtime system invokes this callback after the master code is executed and before this thread executes the statement following the master construct. This callback executes in the context of the task. This callback has type signature `ompt_parallel_callback_t`.

Barriers

`ompt_event_barrier_begin`

The OpenMP runtime system invokes this callback before this thread starts executing the barrier construct. This callback executes in the context of the task. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_barrier_end`

The OpenMP runtime system invokes this callback after this thread completes executing the barrier construct. This callback executes in the context of the task. This callback has type signature `ompt_parallel_callback_t`.

Taskwait

`ompt_event_taskwait_begin`

The OpenMP runtime system invokes this callback before this thread starts executing the taskwait construct. This callback executes in the context of the task. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_taskwait_end`

The OpenMP runtime system invokes this callback after this thread completes executing the taskwait construct. This callback executes in the context of the task. This callback has type signature `ompt_parallel_callback_t`.

Taskgroup

`ompt_event_taskgroup_begin`

The OpenMP runtime system invokes this callback before this thread starts executing the taskgroup construct. This callback executes in the context of the task. This callback has type signature `ompt_parallel_callback_t`.

`ompt_event_taskgroup_end`

The OpenMP runtime system invokes this callback after this thread completes executing the taskgroup construct. This callback executes in the context of the task. This callback has type signature `ompt_parallel_callback_t`.

Locks

`ompt_event_wait_lock`

The OpenMP runtime system invokes this callback if this task enters the `ompt_state_wait_lock` (`ompt_state_wait_nest_lock`) state. This callback executes in the environment of the task; its `wait_id` parameter identifies the (nest) lock. This callback has type signature `ompt_wait_callback_t`.

`ompt_event_acquired_lock`

The OpenMP runtime system invokes this callback just after this task acquires the lock. This callback executes in the environment of the task; its `wait_id` parameter identifies the lock. This callback has type signature `ompt_wait_callback_t`.

Nest Locks

`ompt_event_wait_nest_lock`

The OpenMP runtime system invokes this callback if this task enters the `ompt_state_wait_lock` (`ompt_state_wait_nest_lock`) state. This callback executes in the environment of the task; its `wait_id` parameter identifies the (nest) lock. This callback has type signature `ompt_wait_callback_t`.

`ompt_event_acquired_nest_lock_first`

The OpenMP runtime system invokes this callback just after this task acquires a nest lock for the first time. This callback executes in the environment of the task; its `wait_id` parameter identifies the nest lock. This callback has type signature `ompt_wait_callback_t`.

`ompt_event_release_nest_lock_prev`

The OpenMP runtime system invokes this callback after a nest lock has been released but is still owned by this task. If a nest lock was acquired n times by the same task, this callback occurs for the inner n-1 releases. The nth release is handled by the `ompt_event_release_nest_lock_last` event. This callback executes in the environment of the task; its `wait_id` parameter identifies the released nest lock. This callback has type signature `ompt_wait_callback_t`.

`ompt_event_acquired_nest_lock_next`

The OpenMP runtime system invokes this callback just after this task acquires the nest lock that was already owed by this task. This callback executes in the environment of the task; its `wait_id` parameter identifies the nest lock. This callback has type signature `ompt_wait_callback_t`.

Critical Sections

`ompt_event_wait_critical`

The OpenMP runtime system invokes this callback if this task enters the `ompt_state_wait_critical` state. This callback executes in the environment of the task; its `wait_id` parameter identifies the critical region being entered. This callback has type signature `ompt_wait_callback_t`.

`ompt_event_acquired_critical`

The OpenMP runtime system invokes this callback just after this task enters the critical region. This callback executes in the environment of the task; its `wait_id` parameter identifies the critical region being entered. This callback has type signature `ompt_wait_callback_t`.

Ordered Sections

`ompt_event_wait_ordered`

The OpenMP runtime system invokes this callback if this task enters the `ompt_state_wait_ordered` state. This callback executes in the environment of the task; its `wait_id` parameter identifies a variable associated with the ordered construct. This callback has type signature `ompt_wait_callback_t`.

`ompt_event_acquired_ordered`

The OpenMP runtime system invokes this callback just after this task enters the ordered region. This callback executes in the environment of the task; its `wait_id` parameter identifies a variable associated with the ordered construct. This callback has type signature `ompt_wait_callback_t`.

Atomic Blocks

`ompt_event_wait_atomic`

The OpenMP runtime system invokes this callback if this task enters the `ompt_state_wait_atomic` state. This callback executes in the environment of the task; its `wait_id` parameter identifies the atomic data being computed upon. This callback has type signature `ompt_wait_callback_t`.

`ompt_event_acquired_atomic`

The OpenMP runtime system invokes this callback just after this task enters the atomic region. This callback executes in the environment of the task; its `wait_id` parameter identifies the atomic data being computed upon. This callback has type signature `ompt_wait_callback_t`.

Miscellaneous

`ompt_event_flush`

The OpenMP runtime system invokes this callback just after performing a flush operation. This callback executes in the environment of the task. This callback has type signature `ompt_thread_callback_t`.

4 Tool Data Structures

4.1 Thread and Task Data

Each OpenMP thread and task instance provides an `ompt_data_t` data structure, which is a union of an integer and a pointer.

```
typedef union ompt_data_u {
    uint64_t value;    /* data under tool control    */
    void *ptr;          /* pointer under tool control */
} ompt_data_t;
```

The lifetime of the structure begins when a thread/task instance is created and ends when the instance is destroyed. While the value of a structure is preserved over the lifetime of the thread or task with which it is associated, tools should not assume that the address of an `ompt_data_t` structure remains constant over its lifetime.

When a thread/task instance is created, the callback associated with event creation must initialize the `ompt_data_t` structure. If there is no callback associated with this event, the OpenMP runtime initializes the structure value field to 0. The address of the `ompt_data_t` structure is passed to callbacks associated with the creation/destruction of threads/tasks. The address of the structure can also be retrieved on demand, e.g., by invoking an inquiry function in a signal handler.

If the `ompt_data_t` value field is 0 for a thread or task at the point that an exit callback would be made, the exit callback is not invoked. The tool is responsible for coordinating any concurrent accesses to `ompt_data_t` structures.

4.2 Parallel Region Identifier

Each OpenMP parallel region instance has an associated `ompt_parallel_id_t` that uniquely identifies the region instance.

```
typedef uint64_t ompt_parallel_id_t;
```

The `ompt_parallel_id_t` for a parallel region instance is unique across all instances of all parallel regions. The value of this structure is defined when a parallel region instance is created and passed to callbacks associated with creation/destruction of the parallel region instance. A parallel region's ID can be retrieved on demand, e.g., by invoking an inquiry function in a signal handler. Tools should not assume that `ompt_parallel_id_t` values for adjacent region instances are consecutive.

4.3 Wait Identifier

Each thread instance provides a `ompt_wait_id_t` data structure, which identifies what caused a thread to wait.

```
typedef uint64_t ompt_wait_id_t;
```

For example, when a thread is waiting for a lock, this structure identifies the address of the lock. This structure is undefined when a thread is not in a wait state. The value of the `ompt_wait_id_t` structure is passed to callbacks associated with wait events, and also can be retrieved on demand, e.g., by invoking an inquiry function in a signal handler.

4.4 Pointers to Support Classification of Stack Frames

Each implicit or explicit task instance provides an `ompt_frame_t` data structure which contains pointers to OpenMP runtime procedure frames that appear above and below procedure frames associated with user task code.

```
typedef struct ompt_frame_s {
    void *exit_runtime_frame;    /* next frame is user code     */
    void *reenter_runtime_frame; /* previous frame is user code */
} ompt_frame_t;
```

The structure's lifetime begins when a task instance is created and ends when the task instance is destroyed. While the value of the structure is preserved over the lifetime of the task, tools should not assume that the address of a structure remains constant over its lifetime. Frame data is passed to some callbacks; it can also be retrieved for a task (e.g. by a signal handler). Frame data contains two components:

`exit_runtime_frame` This value is set once, the first time that a task exits the runtime to begin executing user code. This field points to the stack frame of the runtime procedure that called the user code. This value is NULL until just before the task exits the runtime.

`reenter_runtime_frame` This value is set each time that current task re-enters the runtime to create new (implicit or explicit) tasks. This field points to the stack frame of the runtime procedure called by a task to re-enter the runtime. This value is NULL until just after the task re-enters the runtime.

Advice to tool implementers: A monitoring tool using asynchronous sampling can observe values of `exit_runtime_frame` and `reenter_runtime_frame` before they are set to non-NULL values while in the runtime. Tools must be prepared to handle samples that occur in this brief window.

5 Inquiry Functions for Tools

Inquiry functions retrieve data from the execution environment for the tools. All inquiry functions are async signal safe.

exit / reentry	reentry = null	reentry = defined
exit = null	case 1) initial task in user code case 2) explicit task that is created but not yet scheduled	initial task in runtime because of a parallel region or a task creation
exit = defined	non-initial task in user code	non-initial task in runtime because of a parallel region or a task creation

Table 1: Meaning of various values for `exit_runtime_frame` and `reenter_runtime_frame`.

5.1 Enumerate States Supported by an OpenMP Runtime

An OpenMP runtime system is allowed to support other states in addition to those described herein. For instance, a particular runtime system may want to provide finer-grain information about the nature of runtime overhead, e.g., to differentiate between the overhead associated with setting up a parallel region and the overhead associated with setting up a task. Further, a tool may not report all states defined herein, e.g., if state tracking for a particular state would be too expensive. To enable a tool to identify all states that an OpenMP runtime system implements, OMPT provides the following interface for enumerating all possibly reported runtime states.

```
_OMP_EXTERN int ompt_enumerate_state(
  int current_state,
  int *next_state,
  const char **next_state_name
);
```

When this interface is invoked for the first time, the value `ompt_state_first` should be supplied for `current_state`. The argument `next_state` is a pointer to an integer that will be set to the code for the next state in the enumeration. The argument `next_state_name` is a pointer to a location that will be filled in with a pointer to the name associated with `next_state`. Subsequent invocations of `ompt_enumerate_state` should pass the code returned in `next_state` by the prior call. The enumeration is complete when `ompt_state_last` is returned in `next_state`. The canonical way to enumerate the states supported by an OpenMP runtime system is shown below:

```
int state;
const char *state_name;
for (int ok = ompt_enumerate_state(ompt_state_first, &state, &state_name);
     ok && state != ompt_state_last;
     ompt_enumerate_state(state, &state, &state_name)) {
        // tool notes that the runtime supports ompt_state_t "state"
        // associated with "state_name"
}
```

5.2 Thread Data Inquiry

Function `ompt_get_thread_data` is an inquiry function to access data stored by the OpenMP runtime system for the current thread for use by a tool.

```
_OMP_EXTERN ompt_data_t *ompt_get_thread_data(void);
```

This inquiry function returns NULL prior to OpenMP initialization or when no tool is attached to the runtime. This function is async signal safe.

ancestor level value	meaning
0	current parallel region
1	parallel region directly enclosing region at ancestor level 0
2	parallel region directly enclosing region at ancestor level 1
...	

Table 2: Meaning of different values for the `ancestor_level` argument to `ompt_get_parallel_function`.

5.3 Thread State Inquiry

Function `ompt_get_state` is the inquiry function to determine the state of the current thread.

```
_OMP_EXTERN ompt_state_t ompt_get_state(
  ompt_wait_id_t *wait_id
);
```

The location specified by `wait_id` is updated point to the wait identifier associated with the current state, if any, or NULL otherwise. This function returns `ompt_state_undefined` prior to OpenMP initialization or when no tool is attached to the runtime. This function is async signal safe.

5.4 Parallel Region Inquiry

The OMPT interface defines two inquiry functions to access data stored by the OpenMP runtime for parallel regions. The first, `ompt_get_parallel_id`, returns the unique parallel id associated with this instance of the parallel region:

```
_OMP_EXTERN ompt_parallel_id_t ompt_get_parallel_id(
  int ancestor_level
);
```

Outside a parallel region, `ompt_get_parallel_id` should return 0. If a thread is in the idle state, then `ompt_get_parallel_id` should return 0. In all other cases, the thread should return the state of the enclosing parallel region, even if the thread is waiting at a barrier.

The second, `ompt_get_parallel_function`, returns a pointer to the compiler generated function used by the OpenMP runtime to encapsulate the code of the parallel region, if any, and NULL otherwise.

```
_OMP_EXTERN void *ompt_get_parallel_function(
  int ancestor_level
);
```

Both of the functions take an ancestor level as an argument. By specifying different values for ancestor level, one can access information about each parallel region, even if parallel regions are nested. The meaning of different values for the `ancestor_level` argument to `ompt_get_parallel_function` is given in Table 2.

These functions return the value 0 when requesting higher levels of ancestry than available, prior to OpenMP initialization, or when no tool is attached to the OpenMP runtime. These functions are async signal safe.

5.5 Task Region Inquiry

The OMPT interface defines three inquiry functions to access data stored by the OpenMP runtime for task regions. Function `ompt_get_task_data` returns the tool data associated with a given task. Function `ompt_get_task_frame` returns the tool frame associated with a given task.

```
_OMP_EXTERN ompt_data_t *ompt_get_task_data(
  int ancestor_level
);
```

ancestor level value	meaning
0	current task
1	direct parent of task at ancestor level 0
2	direct parent of task at ancestor level 1
...	

Table 3: Meaning of different values for the `ancestor_level` argument to `ompt_get_task_function`.

```
_OMP_EXTERN ompt_frame_t *ompt_get_task_frame(
  int ancestor_level
);
```

The value returned by the `ompt_get_task_function` indicates the compiler-generated function used by the OpenMP runtime to encapsulate the code of the task construct, if any, and NULL otherwise.

```
_OMP_EXTERN void *ompt_get_task_function(
  int ancestor_level
);
```

The meaning of different values for the `ancestor_level` argument to `ompt_get_task_function` is given in Table **??**.

These functions return the value 0 when requesting higher levels of ancestry than available, prior to OpenMP initialization, or when no tool is attached to the OpenMP runtime. These functions are async signal safe.

5.6 Tool Support Version Inquiry

The function `ompt_get_ompt_version` returns the version of the OMPT interface supported by the runtime.

```
_OMP_EXTERN int ompt_get_ompt_version(void);
```

The version of OMPT described by this document is known as version 1.

6 Inquiry and Control Functions for Applications

The functions described in this section are the only ones with a Fortran interface in addition to a C/C++ interface.

6.1 Runtime Version Inquiry

The function `ompt_get_runtime_version`, with the type signature shown below

```
_OMP_EXTERN int ompt_get_runtime_version(char *buffer, int length);
```

fills `buffer` with a version-specific string of at most `length` characters. The suggested format is

```
<vendor>-<major version number>.<minor version number>[-<optional feature]*
```

Namely, a vendor name, major and minor version numbers, and, optionally, a list of zero or more features, separated by dashes. As an example, IBM's OpenMP runtime might return the following version string " IBM-1.1-core=1-blame=1-trace=0", indicating that IBM's OpenMP runtime supports the OMPT tools API core augmented with support for blame shifting, but not support for detailed tracing.

6.2 Tool Control

The function `ompt_control` can be called by an application to pass control information to a tool. The signature for this function is shown below:

```
_OMP_EXTERN void ompt_control(uint64_t command, uint64_t modifier);
```

A classic use case for the `ompt_control` routine might be for an application to start and stop data collection by a tool.

7 Initializing OMPT Support for Tools

An OpenMP runtime need not maintain information to support tools and may provide trivial (and thus, perhaps useless) answers in response to invocations of any API inquiry functions. Section 7.1 describes normal initialization for a tool. Section 7.2 describes environment variable control over tool initialization. Section 8.11 describes a tool initialization API for a debugger.

7.1 Initialization of a Tool

A tool must register itself with an OpenMP runtime system and then specify callbacks for events of interest. Section 7.1.1 describes the initializer for a tool. Section 7.1.2 describes registration of callbacks for OMPT events.

7.1.1 Initializer for a Full-featured Tool

A tool must register itself with an OpenMP runtime by overriding the following weak symbol:

```
_OMP_EXTERN int ompt_initialize(void);
```

The role of `ompt_initialize` is to register callbacks for specific events, e.g., creating a parallel region. A tool must register a callback for every event of interest using `ompt_set_callback`, as described in Section 7.1.2. The OpenMP runtime system defines a weak symbol version of `ompt_initialize` that returns 0; a tool-provided version must return 1.

Since only one tool-provided definition of `ompt_initialize` will be seen by an OpenMP runtime, only one tool can be registered. Ordinarily, `ompt_initialize` will be invoked by an OpenMP runtime immediately after the runtime initializes itself.

An OpenMP runtime system *may* allow registration of a tool after initialization of the OpenMP runtime at a *clean point*. An OpenMP runtime is said to be at a clean point when no pthread is inside a parallel region. An OpenMP runtime system will not necessarily attempt to register a tool at a clean point unless a debugger has previously called `ompd_enable(true)` as described in Section 8.11.

After a process fork, if OpenMP is re-initialized in the child process, the OpenMP runtime system in the child process will call `ompt_initialize` under the same conditions as it would for any process.

7.1.2 Callback Registration for a Full-featured Tool

Full-featured tools register callbacks to receive notification of various events that occur as an OpenMP program executes. A tool uses `ompt_set_callback` to register callback functions.

```
_OMP_EXTERN int ompt_set_callback(
  ompt_event_t event,
  ompt_callback_t callback
);
```

The function `ompt_set_callback` may only be called within the implementation of `ompt_initialize` provided by a tool, as described in Section 7.1.1 The possible return codes for `ompt_set_callback` and their meaning is shown in Table 3. Registration of supported callbacks may fail if this function is called outside

return code	meaning
0	event may occur; no callback is possible
1	event will never occur in runtime
2	event may occur; callback invoked when convenient
3	event may occur; callback always invoked when event occurs

Table 4: Meaning of return codes for `ompt_set_callback`.

`OMPT_INITIALIZE` value	action
undefined	`ompt_initialize` is called after the OpenMP runtime initializes itself. If the return value from `ompt_initialize` is non-zero, the OpenMP runtime must maintain runtime state information for each OpenMP thread and appropriately respond to any invocations of the inquiry API.
disable	OMPT is disabled regardless of whether tools are present or not. The OpenMP runtime is not required to maintain any information about thread state and the runtime may supply trivial answers to any invocations of the inquiry API.
false	OMPT is disabled unless explicitly turned on by call to `ompd_enable`.
true	`ompt_initialize` is called after the OpenMP runtime initializes itself. Regardless of the return value from `ompt_initialize`, the OpenMP runtime must maintain runtime state information for each OpenMP thread and appropriately respond to any invocations of the inquiry API.

Table 5: OpenMP runtime responses to settings of the `OMPT_INITIALIZE` environment variable.

`ompt_initialize`. The `ompt_callback_t` type for a callback does not reflect the actual signature of the callback; OMPT uses this generic type to avoid the need to declare a separate registration function for each actual callback type.

The function `ompt_get_callback`, as shown below, may be called at any time to inspect whether a callback has been registered or not. If a callback has been registered, `ompt_set_callback` will return 1 and set `callback` to the address of the callback function; otherwise, `ompt_set_callback` will return 0.

```
_OMP_EXTERN int ompt_get_callback(
  ompt_event_t event,
  ompt_callback_t *callback
);
```

7.2 An Environment Variable for Tool Initialization

The environment variable `OMPT_INITIALIZE` is used to control tool initialization. Table 4 describes actions an OpenMP runtime system takes in response to various values of `OMPT_INITIALIZE`. Regardless of whether a tool is present or not, setting `OMPT_INITIALIZE=disable` directs an OpenMP runtime to disable all support for tools. Any full-featured tool present will not be initialized and the OpenMP runtime is neither required to maintain any runtime thread state information nor respond to invocations of the OMPT inquiry API with anything but trivial answers.

An OpenMP runtime will attempt to initialize a tool if `OMPT_INITIALIZE` is undefined or set to `true`. In this case, the OpenMP runtime will maintain thread state information. If the OpenMP runtime calls `ompt_initialize`, but no tool-provided version of `ompt_initialize` is present, a weak version of `ompt_initialize` provided by the OpenMP runtime will return 0. If a tool-provided version of `ompt_initialize` is present, it must return 1. Only if `ompt_initialize` returns 1 is the OpenMP runtime obligated to invoke any event callbacks registered by `ompt_initialize` when appropriate.

If `OMPT_INITIALIZE` is set to `disable` or `false`, the OpenMP runtime will not call `ompt_initialize` and attempt to initialize a tool.

If OMPT_INITIALIZE is set to false, all OMPT tool support for state tracking or callbacks will be disabled unless a call to ompd_enable, described in Section 8.11, directs it to do otherwise. Behavior for any other values of OMPT_INITIALIZE is unspecified.

8 OMPD: A Debugger Support Library

An OpenMP runtime system will provide a shared library that a debugger can load to help interpret the state of the runtime in a live process or a core file.

If tool support has been enabled, the OpenMP runtime system will maintain information about the state of each OpenMP thread. This includes ompt_state_t, ompt_wait_id_t, ompt_frame_t, and ompt_parallel_id_t data structures.

8.1 Initialization

The OMPD debugger support library needs the debugger to provide a set of callback functions that enable OMPD to manage memory in the debugger address space, look up sizes for primitive types in the target, to look up symbols in the target, query information about structures in the target, as well as read/write memory in the target. The OMPD library invokes the function ompd_initialize, passing a pointer to a ompd_callbacks_t structure that the debugger will initialize for OMPD. The signature for the function is shown below.

```
EXTERN ompt_rc_t ompd_initialize(
  ompd_callbacks_t *data
);
```

The OMPD library may call ompd_initialize in a library initialization constructor. The type ompd_target_t is defined in Appendix B.7.

8.2 Handle Management

Each OMPD call that is dependent on some context must provide this context via a handle. There are handles for threads, parallel regions, and tasks. Handles are guaranteed to be constant for the duration of the construct they represent. This section describes function interfaces for extracting handle information from the OpenMP runtime system.

8.2.1 Thread Handles

Retrieve handles for all OpenMP threads. The ompd_get_threads operation enables the debugger to obtain handles for all OpenMP threads. A successful invocation of ompd_get_threads returns a pointer to a vector of handles in thread_handle_array and returns the number of handles in num_handles. This call yields meaningful results only if all OpenMP threads are stopped; otherwise, the OpenMP runtime may be creating and/or destroying threads during or after the call, rendering useless the vector of handles returned.

```
EXTERN ompd_rc_t ompd_get_threads(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_thread_handle_t **thread_handle_array,
  int *num_handles
);
```

Retrieve handles for OpenMP threads in a parallel region. The ompd_get_thread_in_parallel operation enables the debugger to obtain handles for all OpenMP threads associated with a parallel region. A successful invocation of ompd_get_thread_in_parallel returns a pointer to a vector of handles in thread_handle_array and returns the number of handles in num_handles. This call yields meaningful results only if all OpenMP threads in the parallel region are stopped; otherwise, the OpenMP runtime may be creating and/or destroying threads during or after the call, rendering useless the vector of handles returned.

```
EXTERN ompd_rc_t ompd_get_thread_in_parallel(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_parallel_handle_t parallel_handle,
  ompd_thread_handle_t **thread_handle_array,
  int *num_handles
);
```

8.2.2 Parallel Region Handles

Retrieve the handle for the innermost parallel region for an OpenMP thread. The operation `ompd_get_top_parallel_region` enables the debugger to obtain the handle for the innermost parallel region associated with an OpenMP thread. This call is meaningful only if the thread whose handle is provided is stopped.

```
EXTERN ompd_rc_t ompd_get_innermost_parallel_region(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_thread_handle_t thread_handle,
  ompd_parallel_handle_t *parallel_handle
 );
```

Retrieve the handle for an enclosing parallel region. The `ompd_get_ancestor_parallel_handle` operation enables the debugger to obtain the handle for the parallel region enclosing the parallel region specified by `parallel_handle`. This call is meaningful only if at least one thread in the parallel region is stopped.

```
EXTERN ompd_rc_t ompd_get_enclosing_parallel_handle(
  ompd_context_t *context,   /* debugger handle for the target */
   ompd_parallel_handle_t parallel_handle,
   ompd_parallel_handle_t *enclosing_parallel_handle
 );
```

8.2.3 Task Handles

Retrieve the handle for the innermost task for an OpenMP thread. The debugger uses the operation `ompd_get_top_task_region` to obtain the handle for the innermost task region associated with an OpenMP thread. This call is meaningful only if the thread whose handle is provided is stopped.

```
EXTERN ompd_rc_t ompd_get_top_task_region(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_thread_handle_t thread_handle,
  ompd_task_handle_t *task_handle
);
```

Retrieve the handle for an enclosing task. The debugger uses `ompd_get_ancestor_task_handle` to obtain the handle for the task region enclosing the task region specified by `task_handle`. This call is meaningful only if the thread executing the task specified by `task_handle` is stopped.

```
EXTERN ompd_rc_t  ompd_get_ancestor_task_handle(
  ompd_context_t *context,   /* debugger handle for the target */
   ompd_task_handle_t task_handle,
   ompd_task_handle_t *parent_task_handle
 );
```

Retrieve implicit task handle for a parallel region. The `ompd_get_implicit_task_in_parallel` operation enables the debugger to obtain handles for implicit tasks associated with a parallel region. This call is meaningful only if all threads associated with the parallel region are stopped.

```
EXTERN ompd_rc_t ompd_get_implicit_task_in_parallel(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_parallel_handle_t parallel_handle,
  ompd_task_handle_t **task_handle_array,
  int *num_handles
);
```

8.3 Process and Thread Settings

The functions `ompd_get_num_procs` and `ompd_get_thread_limit` are third-party versions of the OpenMP runtime functions `omp_get_num_procs` and `omp_get_thread_limit`.

```
EXTERN ompd_rc_t ompd_get_num_procs(
  ompd_tword_t *val
);

EXTERN ompd_rc_t ompd_get_thread_limit(
  ompd_tword_t *val
);
```

8.4 Parallel Region Inquiries

8.4.1 Settings

Determine the number of threads associated with a parallel region.

```
EXTERN ompd_rc_t ompd_get_num_threads(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_parallel_handle_t parallel_handle,
  ompd_tword_t *val
);
```

Determine the nesting depth of a particular parallel region instance.

```
EXTERN ompd_rc_t ompd_get_level(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_parallel_handle_t parallel_handle,
  ompd_tword_t *val
);
```

Determine the number of enclosing active parallel regions. `ompd_get_active_level` returns the number of nested, active parallel regions enclosing the parallel region specified by its handle.

```
EXTERN ompd_rc_t ompd_get_active_level(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_parallel_handle_t parallel_handle,
  ompd_tword_t *val
);
```

8.4.2 OMPT Parallel Region Inquiry Analogues

The functions `ompt_get_parallel_id` and `ompt_get_parallel_function` are third-party variants of their OMPT counterparts. The only difference between the OMPD and OMPT versions is that the OMPD must supply a parallel region handle to provide a context for these inquiries.

```
EXTERN ompd_rc_t ompd_get_parallel_id(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_parallel_handle_t parallel_handle,
  ompd_parallel_id_t *id
);

EXTERN ompd_rc_t ompd_get_parallel_function(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_parallel_handle_t parallel_handle,
  ompd_taddr_t *parallel_addr   /* first instruction in the parallel region */
);
```

8.5 Thread Inquiries

8.5.1 Operating System Thread Inquiry

OMPD provides the function `ompd_get_thread_handle` to inquire whether an operating system thread is an OpenMP thread or not. If the function returns `ompd_rc_ok`, then the operating system thread is an OpenMP thread and `thread_handle` will be initialized with the value of a handle for this thread that is meaningful to the OpenMP runtime system.

```
EXTERN ompd_rc_t ompd_get_thread_handle(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_osthread_t *os_thread,
  ompd_thread_handle_t *thread_handle
);

EXTERN ompd_rc_t ompd_get_osthread(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_thread_handle_t thread_handle,
  ompd_osthread_t *os_thread
);
```

Note: This function does not take a `pthread_t` *as an argument because OMPD should not assume that operating system threads are pthreads.*

8.5.2 OMPT Thread State Inquiry Analogue

The function `ompd_get_state` is a third-party version of `ompt_get_state`. The only difference between the OMPD and OMPT counterparts is that the OMPD version must supply a thread handle to provide a context for this inquiry.

```
EXTERN ompd_rc_t ompd_get_state(
  ompd_context_t *context,
  ompd_thread_handle_t thread_handle,
  ompt_state_t *state,
  ompt_wait_id_t *wait_id
);
```

8.6 Task Inquiries

8.6.1 Task Settings

Retrieve information from OpenMP tasks. These inquiry functions have no counterparts in the OMPT interface as a first-party tool can call OpenMP runtime inquiry functions directly. The only difference between the OMPD inquiry operations and their counterparts in the OpenMP runtime is that the OMPD version must supply a task handle to provide a context for each inquiry.

```
EXTERN ompd_rc_t ompd_get_max_threads(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_task_handle_t task_handle,
  int *val
);

EXTERN ompd_rc_t ompd_get_thread_num(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_task_handle_t task_handle,
  int *val
 );

EXTERN ompd_rc_t ompd_in_parallel(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_task_handle_t task_handle,
  int *val
);

EXTERN ompd_rc_t ompd_in_final(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_task_handle_t task_handle,
  int *val
 );

EXTERN ompd_rc_t ompd_get_dynamic(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_task_handle_t task_handle,
  int *val
);

EXTERN ompd_rc_t ompd_get_nested(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_task_handle_t task_handle,
  int *val
 );

EXTERN ompd_rc_t ompd_get_max_active_levels(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_task_handle_t task_handle,
  int *val
);

EXTERN ompd_rc_t ompd_get_schedule(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_task_handle_t task_handle,
  omp_sched_t *kind,
  int *modifier
);

EXTERN ompd_rc_t ompd_get_proc_bind(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_task_handle_t task_handle,
  omp_proc_bind_t *bind
);
```

8.6.2 OMPT Task Inquiry Analogues

The functions defined here are third-party versions of `ompt_get_task_frame` and `ompt_get_task_function`. The only difference between the OMPD and OMPT counterparts is that the OMPD version must supply a task handle to provide a context for these inquiries.

```
EXTERN ompd_rc_t ompd_get_task_frame(
  ompd_context_t *context,   /* debugger handle for the target */
  ompd_task_handle_t task_handle,
  void *sp_exit,
  void *sp_reentry
);

EXTERN ompd_rc_t ompd_get_task_function(
  ompd_context_t *context,
  ompd_task_handle_t task_handle,
  ompd_taddr_t *task_addr /* address of the first instruction in the task region */
);
```

8.7 OMPD Version and Compatibility Information

The OMPD function `ompd_get_version_string` returns a descriptive string describing an implementation of the OMPD library. The function `ompd_get_version_compatibility` returns an integer code used to indicate the revision of the OMPD specification supported by an implementation of OMPD.

```
EXTERN ompd_rc_t ompd_get_version_string(
  const char **string
);

EXTERN ompd_rc_t ompd_get_version_compatibility(
  int *val
);
```

8.8 OMPD Error String

The OMPD function `ompd_get_error_string` returns a descriptive string to the debugger for a specified error code.

```
EXTERN ompd_rc_t ompd_get_error_string(
  int errcode,
  const char **string
);
```

8.9 Breakpoint Locations for Managing Parallel Regions and Tasks

Neither a debugger nor an OpenMP runtime system know what application code a program will launch as parallel regions or tasks until the program invokes the runtime system and provides a code address as an argument. To help a debugger control the execution of an OpenMP program launching parallel regions or tasks, OMPD provides a routine that the debugger can invoke to determine where to place breakpoints.

The `ompd_get_breakpoints` routine will fill in an `ompd_breakpoints_t` structure with pointers to code locations where the debugger can place breakpoints to intercept execution just before the OpenMP runtime launches a task or parallel region, and just after execution of a parallel region or task completes.

```
typedef struct ompd_breakpoints_s {
  ompd_taddr_t parallel_pre_execute;
  ompd_taddr_t parallel_post_execute;
  ompd_taddr_t task_pre_execute;
  ompd_taddr_t task_post_execute;
} ompd_breakpoints_t;

EXTERN ompd_rc_t ompd_get_breakpoints(
  ompd_context_t *context,  /* debugger handle for the target */
  ompd_breakpoints_t *bkpt_locations
);
```

When the debugger gains control as the `parallel_pre_execute` code location breakpoint triggers, the debugger can determine what user code the parallel region will execute by mapping the operating system thread that triggered the breakpoint to an OpenMP thread handle using `ompd_get_thread_handle`, mapping the thread handle to a parallel region handle using `ompd_get_top_parallel_region`, and then using `ompd_get_parallel_function` to determine the entry point for the user code that the parallel region will execute.

Similarly, when the debugger gains control as a breakpoint at the `task_pre_execute` code location triggers, the debugger can determine what user task code will execute by mapping a native thread to an OpenMP thread handle using `ompd_get_thread_handle`, mapping the thread handle to a parallel region handle using `ompd_get_top_task_region`, and then using `ompt_get_task_function` to determine the entry point for the user tasking code.

Each of these breakpoints is triggered only once per parallel region, not once per thread in a parallel region. The `task_pre_execute` and `task_post_execute` breakpoints may be triggered in different threads if a task executes on a different thread then where it was launched.

8.10 Display Control Variables

Using the `ompd_display_control_vars` function, the debugger can extract a string that contains a sequence of name/value pairs of control variables whose settings are (a) user controllable, and (b) important to the operation or performance of an OpenMP runtime system. The control variables exposed through this interface will include all of the OMP environment variables, settings that may come from vendor or platform-specific environment variables (e.g., the IBM XL compiler has an environment variable that controls spinning vs. blocking behavior), and other settings that affect the operation or functioning of an OpenMP runtime system (e.g., `numactl` settings that cause threads to be bound to cores).

```
EXTERN ompd_rc_t ompd_display_control_vars(
   const char **control_var_values
);
```

The format of the string returned by `ompd_display_control_vars` is a sequence of newline separated name/value pairs of the following form:

```
name=valuestring_that_can_contain_any_char_but_newline
anothername=another value string
```

8.11 OMPT Tool Initialization Control

A debugger can control the level at which OpenMP runtime support for tools is activated by invoking

```
EXTERN int ompd_enable(
  ompd_enable_setting_t setting
);
```

To disable all OMPT support for tools, a debugger calls `ompd_enable(false)`. To enable support for tools, a debugger calls `ompd_enable(true)`. With this setting specified, an OpenMP runtime will maintain runtime state (as described in 2) and support all OMPT tool-facing inquiry functions (as described in Section 5).

When `ompd_enable` is called, its effect is not necessarily instantaneous. A call to enable or disable tool support will take effect at a clean point.

Upon a call to `ompd_enable(true)`, if has not already been enabled, an OpenMP runtime may invoke a tool's `ompt_initialize` callback at the next clean point. Upon a call to `ompd_enable` with `false` as an argument, if a tool has already been initialized and the tool has registered a callback for `ompt_event_runtime_shutdown`, the shutdown callback may occur no earlier than the next clean point.

If a tool is already enabled before a call to `ompd_enable(true)`, a call to `ompt_enable_complete` occurs before the call to `ompd_enable` returns. If no tool is present or it has already been disabled, the call to `ompt_enable_complete` occurs before the call to `ompd_enable` returns. A debugger can set a breakpoint in `ompt_enable_complete` to observe when a tool has been enabled or disabled.

9 OpenMP Runtime Library Global Variables

Section 8 describes the OMPD dynamic library that will help a debugger interact with the state of an OpenMP target process. One difficulty that a debugger faces is determining what plug-in library should be used to interact with a target process. To address this problem, the OpenMP runtime system provides the base name of the family of matching plugins in a public variable. This variable will be available in both live OpenMP processes or a core files.

```
_OMP_EXTERN const char *ompt_debugger_plugin;
```

A debugger will search for a suitable matching library (with the base name given in the variable) in directories in `LD_LIBRARY_PATH`.

References

[1] J. Cownie, J. DelSignore, B. R. de Supinski, and K. Warren. DMPL: an OpenMP DLL debugging interface. In *Proceedings of the OpenMP applications and tools 2003 international conference on OpenMP shared memory parallel programming*, WOMPAT'03, pages 137–146, Berlin, Heidelberg, 2003. Springer-Verlag.

[2] J. Cownie and W. Gropp. A standard interface for debugger access to message queue information in MPI. In *Proceedings of PVMMPI'99*, pages 51–58, 1999. `http://www.mcs.anl.gov/research/projects/mpi/mpi-debug/eurompi-paper.ps.gz`.

[3] M. Itzkowitz, O. Mazurov, N. Copty, and Y. Lin. An OpenMP runtime API for profiling. Sun Microsystems, Inc.. OpenMP ARB White Paper. Available online at `http://www.compunity.org/futures/omp-api.html`.

[4] G. Jost, O. Mazurov, and D. An Mey. Adding new dimensions to performance analysis through user-defined objects. In *Proceedings of the 2005 and 2006 International Conference on OpenMP shared memory parallel programming*, IWOMP'05/IWOMP'06, pages 255–266, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] B. Mohr, A. D. Malony, S. Shende, and F. Wolf. Design and prototype of a performance tool interface for OpenMP. *The Journal of Supercomputing*, 23:105–128, 2002.

[6] Sun Microsystems. Man pages section 3: Threads and realtime library functions: libthread_db(3THR), 1998. `http://docs.oracle.com/cd/E19455-01/806-0630/6j9vkb8dk/index.html`.

[7] N. R. Tallent and J. M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 229–240, New York, NY, USA, 2009. ACM.

[8] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 269–280, New York, NY, USA, 2010. ACM.

A OMPT Interface Type Definitions

A.1 Runtime States

When OMPT is enabled, an OpenMP runtime system will maintain information about the state of each OpenMP thread. Below we define an enumeration type that specifies the set of runtime states. The purpose of these states is described in Section 2.

```
typedef enum {
  /* work states (0..15) */
  ompt_state_work_serial      = 0x00, /* working outside parallel   */
  ompt_state_work_parallel    = 0x01, /* working within parallel    */
  ompt_state_work_reduction   = 0x02, /* performing a reduction     */

  /* idle (16..31) */
  ompt_state_idle             = 0x10, /* waiting for work           */

  /* overhead states (32..63) */
  ompt_state_overhead         = 0x20, /* overhead excluding wait
                                       * states                     */

  /* wait states non-mutex (64..79) */
  ompt_state_wait_barrier     = 0x40, /* waiting at a barrier       */
  ompt_state_wait_taskwait    = 0x41, /* waiting at a taskwait      */
  ompt_state_wait_taskgroup   = 0x42, /* waiting at a taskgroup     */

  /* wait states mutex (80..95) */
  ompt_state_wait_lock        = 0x50, /* waiting for lock           */
  ompt_state_wait_nest_lock   = 0x51, /* waiting for nest lock      */
  ompt_state_wait_critical    = 0x52, /* waiting for critical       */
  ompt_state_wait_atomic      = 0x53, /* waiting for atomic         */
  ompt_state_wait_ordered     = 0x54, /* waiting for ordered        */

  /* misc (96..127) */
  ompt_state_undefined        = 0x60, /* undefined thread state     */
  ompt_state_first            = 0x61, /* initial enumeration state  */
  ompt_state_last             = 0x62, /* final enumeration state    */
} ompt_state_t;
```

A.2 Runtime Event Callbacks

When OMPT support for a tool is enabled, OMPT enables a tool to indicate interest in receiving notification about certain OpenMP runtime events by registering callbacks. When those events occur during execution, OMPT will invoke the registered callback in the appropriate thread context Below we define an enumeration type that specifies the set of event callbacks that may be supported by an OpenMP runtime system. The purpose of these callbacks is described in Section 3.

```
typedef enum {
  /*--- Mandatory Events ---*/
  ompt_event_parallel_create          = 1,  /* parallel create        */
  ompt_event_parallel_exit            = 2,  /* parallel exit          */

  ompt_event_task_create              = 3,  /* task create            */
  ompt_event_task_exit                = 4,  /* task destroy           */

  ompt_event_thread_create            = 5,  /* thread create          */
  ompt_event_thread_exit              = 6,  /* thread exit            */

  ompt_event_control                  = 7,  /* support control calls  */

  ompt_event_runtime_shutdown         = 8, /* runtime shutdown       */

  /*--- Optional Events (blame shifting) ---*/
  ompt_event_idle_begin               = 9,  /* begin idle state       */
  ompt_event_idle_end                 = 10,  /* end idle state         */

  ompt_event_wait_barrier_begin       = 11, /* begin wait at barrier  */
  ompt_event_wait_barrier_end         = 12, /* end wait at barrier    */
  ompt_event_wait_taskwait_begin      = 13, /* begin wait at taskwait */
  ompt_event_wait_taskwait_end        = 14, /* end wait at taskwait   */
  ompt_event_wait_taskgroup_begin     = 15, /* begin wait at taskgroup*/
  ompt_event_wait_taskgroup_end       = 16, /* end wait at taskgroup  */

  ompt_event_release_lock             = 17, /* lock release           */
  ompt_event_release_nest_lock_last   = 18, /* last nest lock release */
  ompt_event_release_critical         = 19, /* critical release       */
  ompt_event_release_atomic           = 20, /* atomic release         */
  ompt_event_release_ordered          = 21, /* ordered release        */

  /*--- Optional Events (synchronous events) --- */
  ompt_event_implicit_task_create     = 22, /* implicit task create   */
  ompt_event_implicit_task_exit       = 23, /* implicit task destroy  */

  ompt_event_task_switch              = 24, /* task switch            */

  ompt_event_loop_begin               = 25, /* task at loop begin     */
  ompt_event_loop_end                 = 26, /* task at loop end       */
  ompt_event_section_begin            = 27, /* task at section begin  */
  ompt_event_section_end              = 28, /* task at section end    */
  ompt_event_single_in_block_begin    = 29, /* task at single begin   */
  ompt_event_single_in_block_end      = 30, /* task at single end     */
  ompt_event_single_others_begin      = 31, /* task at single begin   */
  ompt_event_single_others_end        = 32, /* task at single end     */
  ompt_event_master_begin            = 33, /* task at master begin   */
  ompt_event_master_end              = 34, /* task at master end     */
  ompt_event_barrier_begin           = 35, /* task at barrier begin  */
  ompt_event_barrier_end             = 36, /* task at barrier end    */
  ompt_event_taskwait_begin          = 37, /* task at taskwait begin */
  ompt_event_taskwait_end            = 38, /* task at task wait end  */
  ompt_event_taskgroup_begin         = 39, /* task at taskgroup begin*/
  ompt_event_taskgroup_end           = 40, /* task at taskgroup end  */

  ompt_event_release_nest_lock_prev   = 41, /* prev nest lock release */

  ompt_event_wait_lock               = 42, /* lock wait              */
  ompt_event_wait_nest_lock          = 43, /* nest lock wait         */
  ompt_event_wait_critical           = 44, /* critical wait          */
  ompt_event_wait_atomic             = 45, /* atomic wait            */
  ompt_event_wait_ordered            = 46, /* ordered wait           */

  ompt_event_acquired_lock           = 47, /* lock acquired          */
  ompt_event_acquired_nest_lock_first = 48, /* 1st nest lock acquired */
  ompt_event_acquired_nest_lock_next  = 49, /* next nest lock acquired*/
  ompt_event_acquired_critical       = 50, /* critical acquired      */
  ompt_event_acquired_atomic         = 51, /* atomic acquired        */
  ompt_event_acquired_ordered        = 52, /* ordered acquired       */

  ompt_event_init_lock               = 53, /* lock init              */
  ompt_event_init_nest_lock          = 54, /* nest lock init         */
  ompt_event_destroy_lock            = 55, /* lock destruction       */
  ompt_event_destroy_nest_lock       = 56, /* nest lock destruction  */

  ompt_event_flush                   = 57, /* after executing flush  */

} ompt_event_t;
```

A.3 Type Signatures for Tool Callbacks

The tool callback type signature associated with each event is specified in the description of the `ompt_event_t` enum. Below are definitions for all of the tool callback function signatures.

```
typedef void (*ompt_thread_callback_t) (
  ompt_data_t *thread_data           /* tool data for thread       */
  );

typedef void (*ompt_parallel_callback_t) (
  ompt_data_t *task_data,           /* tool data for a task        */
  ompt_parallel_id_t parallel_id    /* id of parallel region       */
  );

typedef void (*ompt_new_parallel_callback_t) (
  ompt_data_t  *parent_task_data,   /* tool data for parent task   */
  ompt_frame_t *parent_task_frame,  /* frame data of parent task   */
  ompt_parallel_id_t parallel_id    /* id of parallel region       */
  );

typedef void (*ompt_task_callback_t) (
  ompt_data_t *task_data            /* tool data for task          */
  );

typedef void (*ompt_task_switch_callback_t) (
  ompt_data_t *suspended_task_data, /* tool data for suspended task */
  ompt_data_t *resumed_task_data    /* tool data for resumed task   */
  );

typedef void (*ompt_new_task_callback_t) (
  ompt_data_t  *parent_task_data,   /* tool data for parent task    */
  ompt_frame_t *parent_task_frame,  /* frame data for parent task   */
  ompt_data_t  *new_task_data       /* tool data for created task   */
  );

typedef void (*ompt_wait_callback_t) (
  ompt_wait_id_t wait_id            /* wait id                      */
  );

typedef void (*ompt_control_callback_t) (
  uint64_t command,               /* command of control call      */
  uint64_t modifier               /* modifier of control call     */
  );
```

Placeholder callback signature. The type `ompt_callback_t` is a placeholder signature used only by the tool callback registration interface. Only one callback registration function is defined and it expects that the callback supplied will be cast into type `ompt_callback_t`, regardless of its actual type signature. This approach avoids the need for a separate registration routine for each unique tool callback signature.

```
typedef void (*ompt_callback_t)(void);
```

B OMPD Interface Type Definitions

B.1 Basic Types

```
typedef uint64_t ompd_taddr_t;
typedef int64_t  ompd_tword_t;
```

B.2 OS Thread Handle

An OpenMP runtime may be implemented on different threading substrates. OMPD uses the `ompd_osthread_t` type to describe an operating system thread upon which an OpenMP thread is overlaid.

```
typedef enum {
  ompd_osthread_pthread,
  ompd_osthread_lwp
} ompd_osthread_kind_t;

typedef struct  {
  ompd_osthread_kind_t kind;
  union {
    int64_t pthread;
    int64_t lwp;
  } data;
} ompd_osthread_t;
```

B.3 Context Handles

Each OMPD interface operation that applies to a particular thread, parallel region, or task must explicitly specify the context for the operation using a handle. OMPD employs context handles for threads, parallel regions, and tasks. A handle for an entity is constant while the entity itself is live.

```
typedef uint64_t ompd_thread_handle_t;
typedef uint64_t ompd_parallel_handle_t;
typedef uint64_t ompd_task_handle_t;
typedef uint64_t ompd_type_handle_t;
```

B.4 Return Codes

Each OMPD interface operation has a return code. The purpose of the each return code is explained by the comments in the definition below.

```
typedef enum {
  ompd_rc_ok            = 0,  /* operation was successful */
  ompd_rc_unavailable   = 1,  /* info is not available (in this context) */
  ompd_rc_stale_handle  = 2,  /* handle is no longer valid */
  ompt_rc_bad_input     = 3,  /* bad input parameters (other than handle) */
  ompt_rc_error         = 4,  /* error */
  ompt_rc_unsupported   = 5   /* operation is not supported */
  ompd_rc_needs_state_tracking = 6   /* needs runtime state tracking enabled */
} ompd_rc_t;
```

B.5 Primitive Types

This enumeration of primitive types is used by OMPD to interrogate the debugger about the size of primitive types in the target.

```
typedef struct {
  int sizeof_char;
  int sizeof_short;
  int sizeof_int;
  int sizeof_long;
  int sizeof_long_long;
  int sizeof_pointer;
} ompd_target_type_sizes_t;
```

B.6 Type Signatures for Debugger Callbacks

For OMPD to provide information about the internal state of the OpenMP runtime system in a target process, it must have a means to extract information from the target process. The target process may be a live process or core file. To enable OMPD to extract state information from a target process, a debugger supplies OMPD with callback functions to inquire about the size of primitive types in the target, look up symbols, look up the offset of a field in a type, as well as read and write memory in the target. OMPD then uses these callbacks to implement its interface operations. Signatures for the debugger callbacks used by OMPD are given below.

Memory management. The callback signatures below are used to allocate and free memory in the debugger's address space.

```
typedef ompd_rc_t (*ompd_dmemory_alloc_fn_t) (
  ompd_context_t *context,  /* debugger handle for the target */
  size_t bytes,             /* the primitive type of interest */
  void **ptr                /* a successful call returns a pointer to the memory here */
);

typedef ompd_rc_t (*ompd_dmemory_free_fn_t) (
  ompd_context_t *context,  /* debugger handle for the target */
  void *ptr                 /* a successful call deallocates the memory here */
);
```

Primitive type size. The callback signature below is used to look up the sizes of primitive types in the target.

```
typedef ompd_rc_t (*ompd_tmemory_access_fn_t) (
  ompd_context_t *context,  /* debugger handle for the target */
  ompd_target_type_sizes_t *sizes,    /* a successful call returns the  type sizes here */
);
```

Symbol lookup. The callback signature below is used to look up the address of a global symbol in the target.

```
typedef ompd_rc_t (*ompd_tsymbol_addr_fn_t) (
  ompd_context_t *context,  /* debugger handle for the target */
  const char *symbol_name,  /* global symbol name */
  ompd_taddr_t *symbol_addr        /* a successful call returns the symbol address here */
);
```

Type lookup. The callback signature below is used to look up a type in the target.

```
typedef ompd_rc_t (*ompd_ttype_fn_t) (
  ompd_context_t *context,  /* debugger handle for the target */
  const char *type_name,    /* name of the type/structure */
  ompd_ttype_handle_t *ttype_handle         /* a successful call returns the type handle here */
);
```

Type size lookup. The callback signature below is used to look up the size of a type in the target.

```
typedef ompd_rc_t (*ompd_ttype_sizeof_fn_t) (
  ompd_context_t *context,  /* debugger handle for the target */
  ompd_ttype_handle_t *ttype_handle,    /* handle of the type/structure */
  ompd_tword_t *type_size         /* a successful call returns the type size here */
);
```

Type field offset lookup. The callback signature below is used to look up the offset of a field in a type in the target.

```
typedef ompd_rc_t (*ompd_ttype_offset_fn_t) (
  ompd_context_t *context,  /* debugger handle for the target */
  ompd_ttype_handle_t *ttype_handle,    /* handle of the type/structure */
  const char *field_name,   /* field of interest in the type/structure */
  ompd_tword_t *field_offset         /* a successful call returns the field offset here */
);
```

Memory access. The callback signature below is used to read or write memory in the target.

```
typedef ompd_rc_t (*ompd_tmemory_access_fn_t) (
  ompd_context_t *context,  /* debugger handle for the target */
  ompd_taddr_t *addr,               /* address in the process or core file */
  void *buffer,             /* input buffer for write; output buffer for read  */
  ompd_tword_t bufsize            /* number of bytes to be transferred  */
);
```

Data format conversion. The callback signature below is used to convert data from the target byte ordering to the host byte ordering

```
typedef ompd_rc_t (*ompd_target_host_fn_t) (
  ompd_context_t *context,
  const void *input,
  void *output,
  int nbytes);
```

Get error string. The callback signature below is used by OMPD to retrieve an error string from the debugger given an error code.

```
typedef ompd_rc_t (*ompd_error_string_fn_t) (
  ompd_context_t *context,
  int error_code,
  const char **string
```

Print string. The callback signature below is used by OMPD to have the debugger print a string. OMPD should not print directly.

```
typedef ompd_rc_t (*ompd_print_string_fn_t) (
  ompd_context_t *context,
  const char *string
```

B.7 Debugger Callback Interface

OMPD must interact with both the debugger and an OpenMP target process or address space. OMPD must interact with the debugger to allocate or free memory in address space that OMPD shares with the debugger. OMPD needs the debugger to access the target on its behalf to inquire about the sizes of primitive types in the target, look up the address of symbols in the target, look up the offset of fields in structures in the target, as well as read and write memory in the target.

OMPD interacts with the debugger and the target through a callback interface. The callback interface is defined by the `ompd_callbacks_t` structure. The debugger supplies `ompd_callbacks_t` to OMPD by filling it out in the `ompd_initialize` callback.

```
typedef struct {
  /*---------------------------------------------------------------------------------------*/
  /* debugger interface
  /*---------------------------------------------------------------------------------------*/

  /* interface for ompd to allocate/free memory in the debugger's address space */
  ompd_dmemory_alloc_fn_t  d_alloc_memory;      /* allocate memory in the debugger         */
  ompd_dmemory_free_fn_t   d_free_memory;       /* free memory in the debugger             */

  /* errors */
  ompd_error_string_fn_t get_error_string;  /* retrieve an error string for an error code  */

  /* printing */
  ompd_print_string_fn_t print_string;    /* have the debugger print a string for OMPD     */

  /*---------------------------------------------------------------------------------------*/
  /* target interface
  /*---------------------------------------------------------------------------------------*/

  /* obtain information about the size of primitive types in the target   */
  ompd_tsizeof_prim_fn_t   t_sizeof_prim_type;    /* return the size of a primitive type   */

  /* obtain information about symbols and structure offsets in the target  */
  ompd_tsymbol_addr_fn_t   t_symbol_addr_lookup;  /* look up the address of a symbol       */

  ompd_ttype_fn_t   t_type_lookup;                /* look up a type in the target          */
  ompd_ttype_sizeof_fn_t   t_type_sizeof;         /* look up the size of of a type         */
  ompd_ttype_offset_fn_t   t_type_field_offset;   /* look up a field offset in a type      */

  /* access data in the target   */
  ompd_tmemory_access_fn_t t_read_memory;       /* read from target address into buffer    */
  ompd_tmemory_access_fn_t t_write_memory;      /* write from buffer to target address     */

  /* convert byte ordering */
  ompd_target_host_fn_t target_to_host;

} ompd_callbacks_t;
```

Outstanding Issues

General Issues

- Is it preferable to replace all enum definitions with integer types and use `#define` to define values?

OMPT Issues

- The OpenMP runtime currently defines a global variable `ompt_debugger_plugin` to identify a compatible OMPD implementation. We should probably follow the lead of MPIR for how we specify this.
- We need to distinguish which functions require OMPT state tracking to be enabled. Functions that require OMPT state tracking are not guaranteed to be available unless state tracking is explicitly enabled using `OMPT_INITIALIZE=true` or using `ompd_enable(true)`.
- A description of the Fortran binding for the two application-facing inquiry functions is needed.

OMPD Issues

- Are there any other OS thread types that should be covered by `ompd_thread_kind_t`?
- Do we need to distinguish between process and thread context in OMPD?
- Is there a need for a mechanism that will allow OMPD to inquire about values of thread-local variables in used in the runtime implementation of OpenMP threads? If so, we need to design this mechanism.
- Do we want name demangling support from OMPD? Is that something too compiler revision specific to support in OMPD?