OMPD: An Application Programming Interface for a Debugger Support Library for OpenMP

Ariel Burton[1], John DelSignore[1], Alexandre Eichenberger[2], Ignacio Laguna[3], John Mellor-Crummey[4], Martin Schulz[3], Joachim Protze[5]

[1]Rogue Wave Software
[2]IBM T.J. Watson Research Center
[3]Lawrence Livermore National Laboratory
[4]Rice University
[5]RWTH Aachen University

Version 2.001

July 22, 2016

Contents

Acknowledgments

A previous technical report laid down the foundation for OMPD [4] and it was used as a guide and inspiration to develop this document. We acknowledge and appreciate the contribution from the authors of that document: Alexandre Eichenberger, John Mellor-Crummey, Martin Schulz, Nawal Copty, John DelSignore, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz.

We acknowledge and appreciate the input from Andreas Hindborg, Lai Wei, and from members of the OpenMP Tools Working Group.

The work to write this document has been performed partially under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344.

1 Introduction

Today, it is difficult to produce high quality tools that support debugging of OpenMP programs without tightly integrating them with a specific OpenMP runtime implementation. To address this problem, this document defines OMPD, an application programming interface (API) for a shared-library plugin that will enable debuggers to inspect the internal execution state of OpenMP programs. OMPD provides third-party variants of OMPT[3], an emerging OpenMP performance tools application programming interface. Extending the OpenMP standard with this API will make it possible to contruct powerful debugging tools that will support any standard-compliant OpenMP implementation. OMPD will portably enable debuggers to provide OpenMP-aware stack traces, single-stepping in and out of parallel regions, and allow the debugger to operate on the members of a thread team.

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 before 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]. An earlier version of the material presented here appeared in [4].

1.1 Design Objectives

The design for OMPD attempts to satisfy several objectives for a debugging tool interface for OpenMP. These objectives are as follows:

- 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 know where to 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 implementers.
- The API should not impose an unreasonable development burden on tool implementers.

An OpenMP runtime system will provide a shared library that a debugger can dynamically 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 support for OpenMP state, call frame, task and parallel region information.

1.2 Structural Overview

Figure 1 shows how the different components fit together. The debugger loads the OMPD plugin that matches the OpenMP runtime being used by the target. The plugin exports the API defined in this document, which the debugger uses to get OpenMP information about the target. The OMPD plugin will need to look up the symbols, or read data out of the target. It does not do this directly, but instead asks the debugger to perform these operations for it using a callback interface exported by the debugger. This callback interface is also defined in this document (see Section 16).

This architectural layout insulates the debugger from the details of the internal structure of the OpenMP runtime. Similarly, the OMPD plugin does not need to be concerned about how to access the target. Decoupling the plugin and debugger in this ways allows for great flexibility in how the target and tool are deployed, so that, for example, there is no requirement that debugger and target execute on the same machine.

Generally the debugger does not interact directly with the OpenMP runtime in the target, and instead uses the OMPD plugin for this purpose. However, there are a few instances where the debugger does need to access the OpenMP runtime directly. These cases fall into two broad categories. The first is during initialization, where the debugger needs to be able to look up symbols and read variables in OpenMP runtime in order to identify the OPMP plugin it should use. This is discussed in Section 2.

The second category relates to arranging for the debugger to be notified when certain events occur during the execution of the OpenMP program. The model used for this purpose is that the OpenMP implementation is required to define certain symbols in the runtime code. Each of these symbols corresponds to an event type. The runtime must ensure that control passes through the appropriate named location when events occur. If the debugger wants to get notification of an event, it can plant a breakpoint at the matching location.

The code locations can, but do not need to, be functions. They can, for example, simply be labels. However, the names must have external `C` linkage.

1.3 Design Scope

The following OMPD API design is limited in scope to support OpenMP 3.1 (or earlier) programs, and it cannot necessarily be applied to OpenMP 4.0 (or later) programs due to the addition of *target regions* in OpenMP 4.0, which may include accelerator devices such as GPUs.

However, the current OMPD API design allows for future expansion of the OMPD API to support OpenMP 4.0, without breaking compatibility or unnecessarily expanding its size or complexity. To this end, Section 3.1 and Figure 2 include OMPD concepts that will be required to support OpenMP 4.0 target regions in the future.

2 OpenMP Runtime Interface

As part of the OpenMP interface, OMPD requires that the OpenMP runtime system provides a public variable `ompd_dll_locations`, which is an `argv`-style vector of filename string pointers that provides the pathnames(s) of any compatible OMPD plugin implementations (if any). `ompd_dll_locations` must have `C` linkage. The debugger uses the name verbatim, and in particular, will not apply any name mangling before performing the look up. The pathnames may be relative or absolute. The variable declaration is as follows:

```
const char **ompd_dll_locations;
```

`ompd_dll_locations` shall point to a NULL-terminated vector of zero or more NULL-terminated pathname strings. There are no filename conventions for pathname strings. The last entry in the vector shall be NULL. The vector of string pointers must be fully initialized *before* `ompd_dll_locations` is set to a non-NULL value, such that if the debugger stops execution at any point where `ompd_dll_locations` is non-NULL, then the vector of strings it points to is valid and complete.

The programming model or architecture of the debugger (and hence that of the required OMPD) might not match that of the target OpenMP program. It is the responsibility of the debugger to interpret the contents of `ompd_dll_locations` to find a suitable OMPD that matches its own architectural characteristics. On platforms that support different programming models (*e.g.*, 32- v. 64-bit), OpenMP implementers are encouraged to provide OMPD implementations for all models,

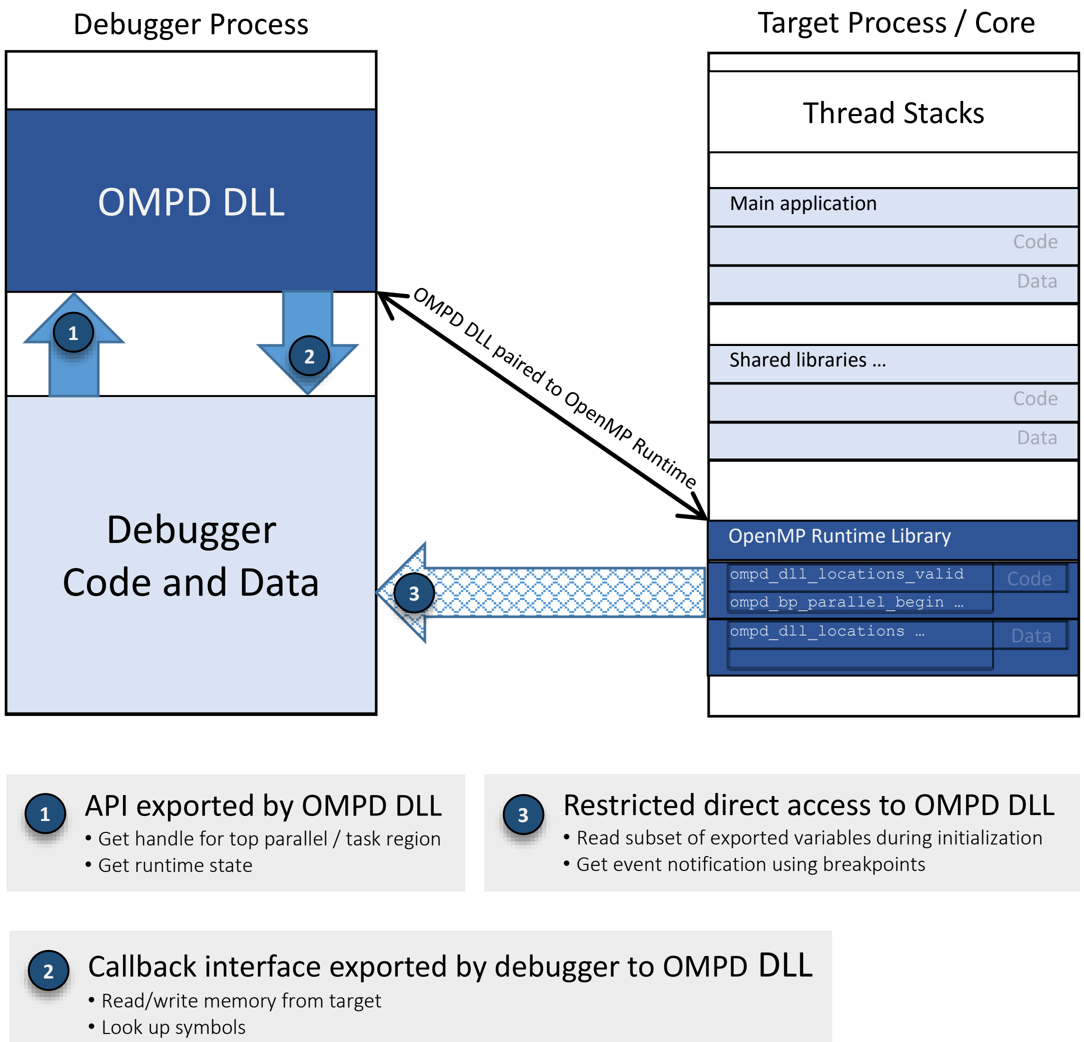

Figure 1: OMPD: Structural overview

and which can handle targets of any model. Thus, for example, a 32-bit debugger should be able to debug a 64-bit target by loading a 32-bit OMPD that can manage a 64-bit OpenMP runtime.

The OpenMP runtime shall notify the debugger that `ompd_dll_locations` is valid by allowing execution to pass through a location identified by the symbol `ompd_dll_locations_valid`. This symbol must have external, `C`, linkage.

Conceptually, `ompd_dll_locations_valid` has the following signature:

```
void ompd_dll_locations_valid ( void );
```

However, as noted in Section 1.2, the event notification location does not need to be a function, and can instead be a labeled location in the code.

The debugger can receive notification of this event by planting a breakpoint at this location. `ompd_dll_locations_valid` has `C` linkage, and the debugger will not apply name mangling before searching for this routine. In order to support debugging, the OpenMP runtime may need to collect and maintain information that it might otherwise not do, perhaps for performance reasons, or because it is not otherwise needed. The OpenMP runtime will collect whatever information is necessary to support OMPD debugging if:

1. the environment variable `OMP_DEBUG` is set to `on`
2. the target calls the `void omp_debug_enable ( void )` function defined in the OpenMP runtime. This function may be called by the main executable, or any of the shared libraries the executable loads, and may be made in an initializer executed when a shared library is loaded (*e.g.*, those in the `.init` section of an ELF DLL). It should be called before the target executes its first OMP construct.

   Rationale: In some cases it may not be possible to control a target's environment. `omp_debug_enable` allows a target itself to turn on data collection for OMPD. Allowing the function to be called from an initializer allows the call to be positioned in an otherwise empty DLL that the programmer can link with the target. This leaves the target code unmodified.

3 Terminology

We refer to the Glossary in the OpenMP standard document [5] for the terms defined there.

This document refers to *contexts* and *handles*. Contexts are entities that are defined by the debugger, and are opaque to the OMPD implementation. Handles are entities that are defined by the OMPD implementation, and are opaque to the debugger. The OMPD API contains opaque definitions of debugger contexts (see Section 15.4) and OMPD handles (see Section 15.3).

Data passed across the interface between the debugger and the OMPD implementation must be managed to prevent memory leakage. Space for data may be allocated on the stack, static data areas, thread local storage, or the heap. In all cases, the data will be said to have an *owner* which is responsible for deallocating them when they are no longer needed. The owner need not be—in fact in many cases is not—the same component that allocated the memory. Where the creating component and owner are different, memory will usually be allocated on the heap. The OMPD implementation must not access the heap directly, but instead it must use the callbacks supplied to it by the debugger. The specific mechanism that must be used by an owner to deallocate memory will depend on the entity involved. Memory management is covered in more detail in Section 5.

All OMPD-related symbols needed by the debugger must have `C` linkage.

3.1 OMPD Concepts

Figure 2 depicts the OMPD concepts of *process*, *address space*, *thread*, *image file*, and *target architecture*, which are defined as follows:

Process A process is a collection of one or more threads and address spaces. The collection may be homogeneous or heterogeneous, containing, for example, threads or address spaces from host programs or accelerator devices. A process may be a "live" operating system process, or a core file.

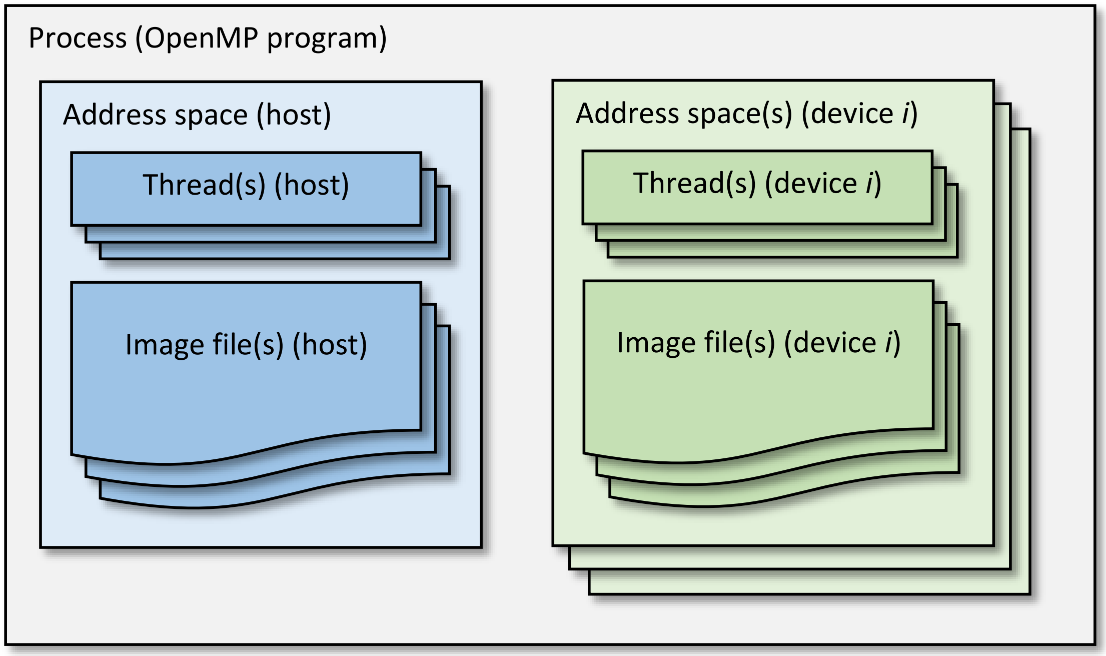

Figure 2: Key concepts of OMPD

Thread A thread is an execution entity running within a specific address space within a process.

Address Space An address space is a collection of logical, virtual, or physical memory address ranges containing code, stack, and data. The memory address ranges within an address space need not be contiguous. An address space may be segmented, where a segmented address consists of a segment identifier and an address in that segment. An address space has associated with it a collection of image files that have been loaded into it. For example, an OpenMP program running on a system with GPUs may consist of multiple address spaces: one for the host program and one for each GPU device. In practical terms, on such systems an OpenMP *device* may be implemented as a CUDA context, which *is* an address space into which CUDA image files are loaded and CUDA kernels are launched.

Image File An image file is an executable or shared library file that is loaded into a target address space. The image file provides symbolic debug information to the debugger.

Target Architecture A target architecture is defined by the processor (CPU or GPU) and the Application Binary Interface (ABI) used by threads and address spaces. A process may contain threads and address spaces for multiple target architectures.

For example, a process may contain a host address space and threads for an x86_64, 64-bit CPU architecture, along with accelerator address spaces and threads for an NVIDIA® GPU architecture or for an Intel® Xeon Phi™ architecture.

3.2 OMPD Handles

OMPD handles identify OpenMP entities during the execution of an OpenMP program. Handles are opaque to the debugger, and defined internally by the OMPD implementation. Below we define these handles and the conditions under which they are guaranteed to be valid.

Address Space Handle The *address space handle* identifies a portion of an instance of an *OpenMP program* that is running on a host device or a target device. The host address space handle is allocated and initialized with the per process or core file initialization call to ompd_process_initialize. A process or core file is initialized by passing the host address space context to that function to obtain an address space handle for the process or core file. The handle remains valid until it is released by the debugger. §4.2, p8

The handle is created by the OMPD implementation, which passes ownership to the debugger which is responsible for indicating when it no longer needs the handle. The debugger releases the handle when it calls ompd_release_address_space_handle. The OMPD implementation can use the handle to cache invariant address-space-specific data (e.g., symbol addresses), and to retain a copy of the debugger's address space context pointer. The handle is passed into subsequent API function calls. In the OMPD API, an address space handle is represented by the opaque type `ompd_address_space_handle_t`. *Future versions of this API will support address space handles for target devices, which will be allocated and initialized by various OMPD API calls.* §7.1, p9

Thread Handle The *thread handle* identifies an *OpenMP thread*. Thread handles are allocated and initialized by various OMPD API calls. A handle is valid for the life time of the corresponding system thread. Thread handles are represented by `ompd_thread_handle_t`, and created by the OMPD implementation which passes ownership to the debugger which is responsible for indicating when it no longer needs the handle. The debugger releases the thread handle by calling ompd_release_thread_handle. §7.2, p11

Parallel Handle The *parallel handle* identifies an *OpenMP parallel region*. It is allocated and initialized by various OMPD API calls. The handle is valid for the life time of the parallel region. The handle is guaranteed to be valid if at least one thread in the parallel region is paused, or if a thread in a nested parallel region is paused. Parallel handles are represented by the opaque type `ompd_parallel_handle_t`, and created by the OMPD implementation which passes ownership to the debugger which is responsible for indicating when it no longer needs the handle. The debugger releases the parallel handle by calling ompd_release_parallel_handle. §7.3, p12

Task Handle The *task handle* identifies an *OpenMP task region*. It is allocated and initialized by various OMPD API calls. The handle is valid for the life time of the task region. The handle is guaranteed to be valid if all threads in the task team are paused. Task handles are represented by the opaque type `ompd_task_handle_t`, and created by the OMPD implementation which passes ownership to the debugger which is responsible for indicating when it no longer needs the handle. The debugger releases the task handle by calling ompd_release_task_handle. §7.4, p14

3.3 Debugger Contexts

Debugger contexts are used to identify a process, address space, or thread object in the debugger. Contexts are passed from the debugger into various OMPD API calls, and then from the OMPD implementation back to the debugger's callback functions. For example, symbol lookup and memory accesses are done in the "context" of a particular address space and possibly thread in the debugger. Contexts are opaque to the OMPD implementation, and defined by the debugger.

Address Space Context The *address space context* identifies the debugger object for a portion of an instance of an *OpenMP program* that is running on a host or target device. An address space is contained within a process, and has an associated target architecture. The address space context must be valid for the life time of its associated address space handle. The host address space context is passed into the process initialization call ompd_process_initialize to associate the host address space context with the address space handle. The OMPD implementation can assume that the address space context is valid until ompd_release_address_space_handle is called for the address space context passed into the initialization routine. §4.2, p8 §7.1, p9

Thread Context The *thread context* identifies the debugger object for a thread. The debugger owns and initializes the thread context. The OMPD implementation obtains a thread context using the `get_thread_context` callback. This callback allows the OMPD implementation to map an operating system thread ID to a debugger thread context. The OMPD implementation can assume that the thread context is valid for as long as the debugger is holding any references to thread handles that may contain the thread context.

3.4 Operating System Thread Identifiers

An operating system thread ID, is the object that allows the debugger and OMPD implementation to map a thread handle to and from a thread context. That is, the OS thread ID is the common identifier for a thread that is visible to both the debugger and the OMPD implementation. The operating system-specific information is platform dependent, and therefore is not defined explicitly in this API. Thus the interface defines `ompd_osthread_kind_t` which identifies what "kind" of information an §15.2, p23 operating system thread ID represents, such as `pthread_t`, lightweight process ID, or accelerator-specific ID. When an operating system thread ID needs to be passed across the interface, the caller passes the "kind" of the ID, the size of the ID in bytes, and a pointer to the operating system-specific information. The format of the information, such as byte ordering, is that of the target. The ID is owned by the caller, which is responsible for its allocation and deallocation.

4 Initialization and Finalization

As described in the following sections, the OMPD DLL must be initialized exactly once after it is leaded, and finalized exactly once before it is unloaded. Per target process or core file initialization and finalization are also required.

4.1 Per DLL Initialization

The debugger starts the initialization by calling `ompd_initialize`, which is defined by the OMPD DLL implementation. Typically this will happen after the debugger has loaded the OMPD DLL. Once loaded, the debugger can determine the version of the OMPD API supported by the DLL by calling the following function in the DLL:

```
ompd_rc_t ompd_get_version ( int *version );
```

On success this should return `ompd_rc_ok`; `ompd_rc_bad_input` indicates that the argument is invalid. Other errors could be reported by `ompd_rc_error`. A descriptive string describing the OMPD implementation is returned by this function:

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

The return values are the same as `ompd_get_version`. The string returned by the OMPD DLL is 'owned' by the DLL, and it must not be modified or released by the debugger. It is guaranteed to remain valid for as long as the DLL is loaded. `ompd_get_version_string` may be called before `ompd_initialize` (see below). Accordingly, the OMPD DLL must not use heap or stack memory for the string it returns to the debugger.

The signatures of `ompd_get_version` and `ompd_get_version_string` are guaranteed not to change in future version of the API. In contrast, the type definitions and prototypes in the rest of the API do not carry the same guarantee. Therefore the debugger should check the version of the API of a loaded OMPD DLL before calling any other function of the API.

The debugger must provide the OMPD library with a set of callback functions that enable OMPD to allocate and deallocate memory in the debugger's address space, to lookup the sizes of basic primitive types in the target, to lookup symbols in the target, as well as to read and write memory in the target. These callback functions are provided to the OMPD library via a table—a list of function pointers—of type `ompd_callbacks_t`.

The signature of the function is shown below:

```
ompd_rc_t ompd_initialize ( const ompd_callbacks_t *callbacks );
```

The type ompd_callbacks_t is defined in Section 16. The argument is guaranteed to be valid for the duration of the call. The OMPD library cannot assume that callbacks will remain valid after the call returns back to the debugger. §16, p29

On success, ompd_initialize returns ompd_rc_ok. If the data argument is invalid, ompd_rc_bad_input should be returned. All other errors will be reported by ompd_rc_error.

The above initialization is performed for each OMPD DLL that is loaded by the debugger; there may more than one DLL present in the debugger because it may be controlling a number of targets that may be using different runtimes which require different OMPD DLLs. This initialization must be performed exactly once before the debugger can begin operating on a target process or core file.

4.2 Per Target Initialization

The debugger initializes a session working on a target process or core file by calling:

```
ompd_rc_t ompd_process_initialize (
  ompd_address_space_context_t  *context,                /* IN */
  ompd_address_space_handle_t  **handle                 /* OUT */
);
```

The context argument is the pointer to the debugger's host address space context object for the target process or core file. The OMPD implementation returns a pointer to the address space handle in *handle, which the debugger is responsible for releasing when it is no longer needed. This function must be called before any OMPD operations are performed on the target. ompd_process_initialize gives the OMPD DLL an opportunity to confirm that it is capable of handling the target process or core file identified by the context. Incompatibility is signaled by a return value of ompd_rc_incompatible.

On return, the handle is owned by the debugger, which must release it using ompd_release_address_space_handle. §7.1, p9

4.3 Per Target Finalization

When the debugger is finished working on the target address space for a process or core file, it calls ompd_release_address_space_handle to tell the OMPD implementation that it not longer needs the address space, and to give the OMPD implementation an opportunity to release any resources it may have related to the handle. §7.1, p9

4.4 Per DLL Finalization

When the debugger is finished with the OMPD DLL it should call:

```
ompd_rc_t ompd_finalize ( void );
```

before unloading the DLL. This should be the last call the debugger makes to the DLL before unloading it. The call to ompd_finalize gives the OMPD DLL a chance to free up any remaining resources it may be holding.

The OMPD DLL may implement a *finalizer* section. This will execute as the DLL is unloaded, and therefore after the debugger's call to ompd_finalize. The OMPD DLL is allowed to use the callbacks (provided to it earlier by the debugger after the call to ompd_initialize) during finalization.

5 Memory Management

The OMPD DLL must not access the heap manager directly. Instead if it needs heap memory it should use the memory allocation and deallocation callback functions provided by the debugger to obtain and release heap memory. This will ensure that the DLL does not interfere with any custom memory management scheme the debugger may use.

If the OMPD DLL is implemented in `C++`, memory management operators like `new` and `delete` in all their variants, *must all* be overloaded and implemented in terms of the callbacks provided by the debugger. The OMPD DLL must be coded so that any of its definitions of `new` or `delete` do not interfere with any defined by the debugger.

In some cases the OMPD DLL will need to allocate memory to return results to the debugger. This memory will then be 'owned' by the debugger, which will be responsible for releasing it. It is therefore vital that the OMPD DLL and the debugger use the same memory manager.

Handles are created by the OMPD implementation. These are opaque to the debugger, and depending on the specific implementation of OMPD may have complex internal structure. The debugger cannot know whether the handle pointers returned by the API correspond to discrete heap allocations. Consequently, the debugger must not simply deallocate a handle by passing an address it receives from the OMPD DLL to its own memory manager. Instead, the API includes functions that the debugger must use when it no longer needs a handle.

Contexts are created by the debugger and passed to the OMPD implementation. The OMPD DLL does not need to release contexts; instead this will be done by the debugger after it releases any handles that may be referencing the contexts.

6 Thread and Signal Safety

The OMPD implementation does not need to be reentrant. It is the responsibility of the debugger to ensure that only one thread enters the OMPD DLL at a time.

The OMPD implementation must not install signal handlers or otherwise interfere with the debugger's signal configuration.

7 Handle Management

Each OMPD call that is dependent on some context must provide this context via a handle. There are handles for address spaces, 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.

7.1 Address Space Handles

The debugger obtains an address space handle when it initializes a session on a live process or core file by calling `ompd_process_initialize`. On return from `ompd_process_initialize` the address space handle is owned by the debugger. §4.2, p8

When the debugger is finished with the target address space handle it should call `ompd_release_address_space_handle` to release the handle and give the OMPD implementation the opportunity to release any resources it may have related to the target.

```
ompd_rc_t ompd_release_address_space_handle (
  ompd_address_space_handle_t  *handle                    /* IN */
);
```

7.2 Thread Handles

Retrieve handles for all OpenMP threads. The `ompd_get_threads` operation enables the debugger to obtain pointers to handles for all OpenMP threads associated with an address space handle. A successful invocation of `ompd_get_threads` returns a pointer to a vector of pointers to handles in `*thread_handle_vector` and returns the number of handle pointers in `*num_handles`. This call yields meaningful results only if all OpenMP threads in the target process 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.

```
ompd_rc_t ompd_get_threads (
  ompd_address_space_handle_t  *handle,                     /* IN */
  ompd_thread_handle_t      ***thread_handle_vector,      /* OUT */
  int                          *num_handles               /* OUT */
);
```

The `num_handles` pointer argument must be valid. The `thread_handle_vector` pointer argument may be NULL, in which case the number of handles that would have been returned had the argument not been NULL is returned in `*num_handles`. This allows the debugger to find out how many OpenMP threads are running in the address space when it is not interested in the handles themselves.

The OMPD DLL gets the memory required for the vector of pointers to thread handles using the memory allocation routine in the callbacks it received during the call to `ompd_initialize`. §4.1, p8 If the OMPD implementation needs to allocate heap memory for the thread handles, it must use the callbacks to acquire this memory. On return, the vector and the thread handles are ‘owned’ by the debugger, and the debugger is responsible for releasing them when they are no longer required.

The thread handles must be released by calling `ompd_release_thread_handle`. §7.2, p11 The vector was allocated by the OMPD implementation using the allocation routine in the callbacks it received during initialization (see `ompt_initialize`); §4.1, p8 the debugger must deallocate the vector in a compatible manner.

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 pointers to thread handles in `*thread_handle_vector`, 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.

```
ompd_rc_t ompd_get_thread_in_parallel (
  ompd_parallel_handle_t   *parallel_handle,                        /* IN */
  ompd_thread_handle_t   ***thread_handle_vector,                 /* OUT */
  int                       *num_handles                          /* OUT */
);
```

The `num_handles` pointer argument must be valid. The `thread_handle_vector` pointer argument may be NULL, in which case the number of handles that would have been returned had the argument not been NULL is returned in `*num_handles`.

The OMPD must obtain the memory for the vector of pointers to thread handles using the memory allocation callback function that was passed to it during `ompd_initialize`. §4.1, p8 If the OMPD implementation needs to allocate heap memory for the thread handles it must use the callbacks to acquire this memory. After the call the vector and the thread handles are ‘owned’ by the debugger, which is responsible for releasing them. The thread handles must be released by calling `ompd_thread_handle`. §7.2, p11 The vector was allocated by the OMPD implementation using the allocation routine in the callbacks; the debugger must deallocate the vector in a compatible manner.

Retrieve the handle for the OpenMP master thread in a parallel region. The `ompd_get_master_thread_in_parallel` operation enables the debugger to obtain a handle for the OpenMP master thread in a parallel region. A successful invocation of `ompd_get_master_thread_in_parallel` returns a handle for the thread that encountered the parallel construct. This call yields meaningful results only if an OpenMP thread in the parallel region is stopped; otherwise, the parallel region is not guaranteed to be alive.

```
ompd_rc_t ompd_get_master_thread_in_parallel (
   ompd_parallel_handle_t  *parallel_handle,                          /* IN */
   ompd_thread_handle_t   **thread_handle                            /* OUT */
);
```

On success `ompd_get_master_thread_in_parallel` returns `ompd_rc_ok`. A pointer to the thread handle is returned in `*thread_handle`. After the call the thread handle is owned by the debugger, which must release it when it is no longer required by calling `ompd_release_thread_handle`. §7.2, p11

Release a thread handle. Thread handles are opaque to the debugger, which therefore cannot release them directly. Instead, when the debugger is finished with a thread handle it must pass it to the OMPD `ompd_release_thread_handle` routine for disposal.

```
ompd_rc_t ompd_release_thread_handle (
  ompd_thread_handle_t  *thread_handle                                /* IN */
);
```

Compare thread handles. The internal structure of thread handles is opaque to the debugger. While the debugger can easily compare pointers to thread handles, it cannot determine whether handles of two different addresses refer to the same underlying thread. The following function can be used to compare thread handles.

```
ompd_rc_t ompd_thread_handle_compare (
  ompd_thread_handle_t   *thread_handle_1,                            /* IN */
  ompd_thread_handle_t   *thread_handle_2,                            /* IN */
  int                    *cmp_value                                  /* OUT */
);
```

On success, `ompd_thread_handle_compare` returns in `*cmp_value` a signed integer value that indicates how the underlying threads compare: a value less than, equal to, or greater than 0 indicates that the thread corresponding to `thread_handle_1` is, respectively, less than, equal to, or greater than that corresponding to `thread_handle_2`.

For OMPD implementations that always have a single, unique, underlying thread handle for a given thread, this operation reduces to a simple comparison of the pointers. However, other implementations may take a different approach, and therefore the only reliable way of determining whether two different pointers to thread handles refer the same or distinct threads is to use `ompd_thread_handle_compare`.

Allowing thread handles to be compared allows the debugger to hold them in ordered collections. The means by which thread handles are ordered is implementation-defined.

String id. The `ompd_get_thread_handle_string_id` function returns a string that contains a unique printable value that identifies the thread. The string should be a single sequence of alphanumeric or underscore characters, and NULL terminated.

```
ompd_rc_t ompd_get_thread_handle_string_id (
  ompd_thread_handle_t   *thread_handle,                              /* IN */
  char                  **string_id                                  /* OUT */
);
```

The OMPD implementation allocates the string returned in `*string_id` using the allocation routine in the callbacks passed to it during initialization. On return the string is owned by the debugger, which is responsible for deallocating it.

The contents of the strings returned for thread handles which compare as equal with `ompd_thread_handle_compare` must be the same. §7.2, p11

7.3 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 a pointer to the parallel handle for the innermost, or topmost, parallel region associated with an OpenMP thread. This call is meaningful only if the thread whose handle is provided is stopped.

```
ompd_rc_t ompd_get_top_parallel_region (
  ompd_thread_handle_t        *thread_handle,                       /* IN */
  ompd_parallel_handle_t     **parallel_handle                      /* OUT */
);
```

The parallel handle must be released by calling `ompd_release_parallel_handle`. §7.3, p12

Retrieve the handle for an enclosing parallel region. The `ompd_get_enclosing_parallel_handle` operation enables the debugger to obtain a pointer to the parallel 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.

```
ompd_rc_t ompd_get_enclosing_parallel_handle (
  ompd_parallel_handle_t        *parallel_handle,                   /* IN */
  ompd_parallel_handle_t       **enclosing_parallel_handle         /* OUT */
);
```

On success `ompd_get_enclosing_parallel_handle` returns `ompd_rc_ok`. A pointer to the parallel handle for the enclosing region is returned in `*enclosing_parallel_handle`. After the call the handle is owned by the debugger, which must release it when it is no longer required by calling `ompd_release_parallel_handle`. §7.3, p12

Retrieve the handle for the parallel region enclosing a task. The `ompd_get_task_parallel_handle` operation enables the debugger to obtain a pointer to the parallel handle for the parallel region enclosing the task region specified by `task_handle`. This call is meaningful only if at least one thread in the parallel region is stopped.

```
ompd_rc_t ompd_get_task_parallel_handle (
  ompd_task_handle_t            *task_handle,                       /* IN */
  ompd_parallel_handle_t       **task_parallel_handle              /* OUT */
);
```

On success `ompd_get_task_parallel_handle` returns `ompd_rc_ok`. A pointer to the parallel regions handle is returned in `*task_parallel_handle`. The parallel handle is owned by the debugger, which must release it by calling `ompd_release_parallel_handle`. §7.3, p12

Release a parallel region handle. Parallel region handles are opaque to the debugger, which therefore cannot release them directly. Instead, when the debugger is finished with a parallel region handle it must must pass it to the OMPD `ompd_release_parallel_handle` routine for disposal.

```
ompd_rc_t ompd_release_parallel_handle (
  ompd_parallel_handle_t  *parallel_handle                          /* IN */
);
```

Compare parallel region handles. The internal structure of parallel region handles is opaque to the debugger. While the debugger can easily compare pointers to parallel region handles, it cannot determine whether handles at two different addresses refer to the same underlying parallel region.

```
ompd_rc_t ompd_parallel_handle_compare (
  ompd_parallel_handle_t  *parallel_handle_1,                  /* IN */
  ompd_parallel_handle_t  *parallel_handle_2,                  /* IN */
  int                     *cmp_value                          /* OUT */
);
```

On success, `ompd_parallel_handle_compare` returns in `*cmp_value` a signed integer value that indicates how the underlying parallel regions compare: a value less than, equal to, or greater than 0 indicates that the region corresponding to `parallel_handle_1` is, respectively, less than, equal to, or greater than that corresponding to `parallel_handle_2`.

For OMPD implementations that always have a single, unique, underlying parallel region handle for a given parallel region, this operation reduces to a simple comparison of the pointers. However, other implementations may take a different approach, and therefore the only reliable way of determining whether two different pointers to parallel regions handles refer the same or distinct parallel regions is to use `ompd_parallel_handle_compare`.

Allowing parallel region handles to be compared allows the debugger to hold them in ordered collections. The means by which parallel region handles are ordered is implementation-defined.

String id. The `ompd_get_parallel_handle_string_id` function returns a string that contains a unique printable value that identifies the parallel region. The string should be a single sequence of alphanumeric or underscore characters, and NULL terminated.

```
ompd_rc_t ompd_get_parallel_handle_string_id (
  ompd_parallel_handle_t   *parallel_handle,                   /* IN */
  char                    **string_id                         /* OUT */
);
```

The OMPD implementation allocates the string returned in `*string_id` using the allocation routine in the callbacks passed to it during initialization. On return the string is owned by the debugger, which is responsible for deallocating it.

The contents of the strings returned for parallel regions handles which compare as equal with `ompd_parallel_handle_compare` must be the same. §7.3, p13

7.4 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 a pointer to the task handle for the innermost, or topmost, task region associated with an OpenMP thread. This call is meaningful only if the thread whose handle is provided is stopped.

```
ompd_rc_t ompd_get_top_task_region (
  ompd_thread_handle_t       *thread_handle,                   /* IN */
  ompd_task_handle_t        **task_handle                     /* OUT */
);
```

The task handle must be released by calling `ompd_release_task_handle`. §7.4, p14

Retrieve the handle for an enclosing task. The OMPD API includes operations to obtain the handle of the parent of a task represented by a given task handle. There are two notions of parenthood. The *scheduling* parent task is the task that was active when the child task was scheduled to run. The *generating* is the task that encountered the OpenMP that caused the child task to be created.

The generating and scheduling parents need not be the same. This might happen if the thread executing a task encounters an OpenMP construct. When this happens, the thread will enter the runtime. The runtime will set up the tasks to implement the OpenMP program construct, and then call its schedular to choose a task to execute. If the scheduler chooses a task other than one of these newly created tasks to run, the scheduling parent of the selected task will not be the same as its generating parent. The former is the task that the thread was executing most recently, and from which it entered the runtime. The later is the task which encountered the OpenMP construct it is executing.

The debugger uses `ompd_get_generating_ancestor_task_region` to obtain a pointer to the task handle for the task that encountered the OpenMP construct which caused the task represented by `task_handle` to be created. This call is meaningful only if the thread executing the task specified by `task_handle` is stopped.

```
ompd_rc_t  ompd_get_generating_ancestor_task_region (
  ompd_task_handle_t         *task_handle,                          /* IN */
  ompd_task_handle_t        **parent_task_handle                   /* OUT */
);
```

The `ompd_get_scheduling_ancestor_task_region` returns the scheduling parent of the task represented `task_handle`. The scheduling parent task is the OpenMP task that was active when the child task was scheduled.

This call is meaningful only if the thread executing the task specified by `task_handle` is stopped.

```
ompd_rc_t  ompd_get_scheduling_ancestor_task_region (
  ompd_task_handle_t         *task_handle,                          /* IN */
  ompd_task_handle_t        **parent_task_handle                   /* OUT */
);
```

The parent task handle must be released by calling ompd_release_task_handle. §7.4, p14

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

```
ompd_rc_t ompd_get_implicit_task_in_parallel (
  ompd_parallel_handle_t          *parallel_handle,                 /* IN */
  ompd_task_handle_t            ***task_handle_vector,             /* OUT */
  int                              *num_handles                    /* OUT */
);
```

The OMPD must use the memory allocation callback to obtain the memory for the vector of pointers to task handles returned by the operation. If the OMPD implementation needs to allocate heap memory for the task handles it returns, it must use the callbacks to acquire this memory. After the call the vector and the task handles are 'owned' by the debugger, which is responsible for deallocating them. The task handles must be released calling ompd_release_task_handle. §7.4, p14 The vector was allocated by the OMPD implementation using the allocation routine passed to it during the call to ompd_initialize. The debugger itself must deallocate the vector in a compatible manner. §4.1, p8

Release a task handle. Task handles are opaque to the debugger, which therefore cannot release them directly. Instead, when the debugger is finished with a task handle it must pass it to the OMPD `ompd_release_task_handle` routine for disposal.

```
ompd_rc_t ompd_release_task_handle (
  ompd_task_handle_t  *task_handle                                  /* IN */
);
```

Compare task handles. The internal structure of task handles is opaque to the debugger. While the debugger can easily compare pointers to task handles, it cannot determine whether handles at two different addresses refer to the same underlying task.

```
ompd_rc_t ompd_task_handle_compare (
  ompd_task_handle_t  *task_handle_1,                        /* IN */
  ompd_task_handle_t  *task_handle_2,                        /* IN */
  int                 *cmp_value                            /* OUT */
);
```

On success, `ompd_task_handle_compare` returns in `*cmp_value` a signed integer value that indicates how the underlying tasks compare: a value less than, equal to, or greater than 0 indicates that the task corresponding to `task_handle_1` is, respectively, less than, equal to, or greater than that corresponding to `task_handle_2`.

For OMPD implementations that always have a single, unique, underlying task handle for a given task, this operation reduces to a simple comparison of the pointers. However, other implementations may take a different approach, and therefore the only reliable way of determining whether two different pointers to task handles refer the same or distinct task is to use `ompd_task_handle_compare`.

Allowing task handles to be compared allows the debugger to hold them in ordered collections. The means by which task handles are ordered is implementation-defined.

String id. The `ompd_get_task_handle_string_id` function returns a string that contains a unique printable value that identifies the task. The string should be a single sequence of alphanumeric or underscore characters, and NULL terminated.

```
ompd_rc_t ompd_get_task_handle_string_id (
  ompd_task_handle_t   *task_handle,                         /* IN */
  char                **string_id                           /* OUT */
);
```

The OMPD implementation allocates the string returned in `*string_id` using the allocation routine in the callbacks passed to it during initialization. On return the string is owned by the debugger, which is responsible for deallocating it.

The contents of the strings returned for task handles which compare as equal with `ompd_task_handle_compare` must be the same. §7.4, p15

8 Address Space 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`.

```
ompd_rc_t ompd_get_num_procs (
  ompd_address_space_handle_t  *handle,                      /* IN */
  ompd_tword_t                 *val                         /* OUT */
);

ompd_rc_t ompd_get_thread_limit (
  ompd_address_space_handle_t  *handle,                      /* IN */
  ompd_tword_t                 *val                         /* OUT */
);
```

The `ompd_get_num_procs` function returns the number of processors available to the device associated with the address space `handle` in `*val`.

The `ompd_get_thread_limit` function returns the maximum number of OpenMP threads available on the device associated with the address space `handle` in `*val`.

9 Parallel Region Inquiries

We describe OMPD functions to perform inquiries about parallel regions.

9.1 Parallel Region Settings

Determine the number of threads associated with a parallel region.

```
ompd_rc_t ompd_get_num_threads (
  ompd_parallel_handle_t  *parallel_handle,                          /* IN */
  ompd_tword_t            *val                                      /* OUT */
);
```

Determine the nesting depth of a particular parallel region.

```
ompd_rc_t ompd_get_level (
  ompd_task_handle_t  *task_handle,                                  /* IN */
  ompd_tword_t        *val                                          /* OUT */
);
```

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

```
ompd_rc_t ompd_get_active_level (
  ompd_task_handle_t  *task_handle,                                  /* IN */
  ompd_tword_t        *val                                          /* OUT */
);
```

9.2 Parallel Function Entry Point

The `ompd_get_parallel_function` returns the entry point of the code that corresponds to the body of the parallel construct.

```
ompd_rd_t ompd_get_parallel_function (
  ompd_parallel_handle_t  *parallel_handle,                          /* IN */
  ompd_address_t          *entry_point                              /* OUT */
);
```

10 Thread Inquiries

We describe OMPD functions to perform inquiries about threads.

10.1 Operating System Thread Inquiry

Mapping an operating system thread to an OMPD thread handle. 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 to a pointer to the thread handle for the OpenMP thread.

```
ompd_rc_t ompd_get_thread_handle (
  ompd_address_space_handle_t   *handle,                       /* IN */
  ompd_osthread_kind_t           kind,                         /* IN */
  ompd_size_t                    sizeof_osthread,              /* IN */
  const void                    *osthread,                     /* IN */
  ompd_thread_handle_t         **thread_handle                /* OUT */
);
```

The operating system ID `*osthread` is guaranteed to be valid for the duration of the call. If the OMPD implementation needs to retain the operating system-specific thread identifier it must copy it.

The thread handle `*thread_handle` returned by the OMP implementation is 'owned' by the debugger, which must release it by calling `ompd_release_thread_handle`. If `os_thread` does not refer to an OpenMP thread, `ompd_get_thread_handle` returns `ompd_rc_bad_input` and `*thread_handle` is also set to NULL. §7.2, p11

Mapping an OMPD thread handle to an operating system thread. `ompd_get_osthread` performs the mapping between an OMPD thread handle and an operating system-specific thread identifier.

```
ompd_rc_t ompd_get_osthread (
  ompd_thread_handle_t  *thread_handle,                        /* IN */
  ompd_osthread_kind_t   kind,                                 /* IN */
  ompd_size_t            sizeof_osthread,                      /* IN */
  void                  *osthread                             /* OUT */
);
```

The caller indicates what *kind* of operating system-specific thread identifier it wants by setting the `kind` 'in' parameter. It also passes a pointer to the buffer into which the OMPD implementation writes the operating system-specific thread identifier, and the size of the buffer, to the OMPD implementation. The buffer is owned by the debugger. §15.2, p23

On success `ompd_get_osthread` returns `rc_ok`, and returns the operating system-specific thread identifier in `*osthread`. If the operation fails, the OMPD implementation returns the appropriate value from `ompd_rc_t`. Note that the operation should fail if the OMPD implementation is unable to return an operating system-specific identifier of the requested 'kind' or size. §15.5, p24

10.2 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.

```
ompd_rc_t ompd_get_state (
  ompd_thread_handle_t  *thread_handle,                        /* IN */
  ompd_state_t          *state,                               /* OUT */
  ompd_wait_id_t        *wait_id                              /* OUT */
);
```

The states are represented by values of the enumeration type `ompd_state_t`. The symbolic names of the members of `ompd_state_t` should match those of the OMPT enumeration type `omp_state_e`. However, there is no guarantee that the numeric values of the corresponding symbolic constants are identical. §15.9, p26

11 Task Inquiries

We describe OMPD functions to perform inquiries about tasks.

11.1 Task Function Entry Point

The `ompd_get_task_function` returns the entry point of the code that corresponds to the body of code executed by the task:

```
ompd_rc_t ompd_get_task_function (
  ompd_task_handle_t  *task_handle,                                /* IN */
  ompd_address_t      *entry_point                                /* OUT */
);
```

11.2 Task Settings

Here we describe functions to retrieve information from OpenMP tasks, including the values of some *Internal Control Variables (ICVs)*. A target is able to get the information defined here directly from the runtime. For this reason, these inquiry functions have no counterparts in the OMPT interface. 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. Values are returned through the 'out' parameter `val`.

The `ompd_get_max_threads` function returns the value of the target's *nthreads-var* ICV (§2.3.1 of [5]), and corresponds to the `omp_get_max_threads` function in the OpenMP runtime API. This returns an upper bound on the number threads that could be used to form a new team if a `parallel` construct without a `num_threads` clause were encountered.

```
ompd_rc_t ompd_get_max_threads (
  const ompd_task_handle_t  *task_handle,                          /* IN */
  ompd_tword_t              *val                                  /* OUT */
);
```

The *nthreads-var* ICV is defined in OpenMP as a list (§2.3.2 of [5]). Like `omp_get_max_threads`, `ompd_get_max_threads` returns the first element of the list.

```
ompd_rc_t ompd_get_thread_num (
  const ompd_thread_handle_t  *thread_handle,                      /* IN */
  ompd_tword_t                *val                                /* OUT */
);
```

`ompd_get_thread_num` corresponds to the `omp_get_thread_num` routine in the OpenMP runtime, and returns the thread's logical thread number in the team.

`ompd_in_parallel` returns logical true (*i.e.*, `*val != 0`) if *active-levels-var* ICV (§2.3.1 of [5]) is greater than 0, and false (0) otherwise. The routine corresponds to `omp_in_parallel` in the OpenMP runtime.

```
ompd_rc_t ompd_in_parallel (
  const ompd_task_handle_t  *task_handle,                          /* IN */
  ompd_tword_t              *val                                  /* OUT */
);
```

`ompd_in_final` corresponds to `omp_in_final` and returns logical true if the task is a final task.

```
ompd_rc_t ompd_in_final (
  const ompd_task_handle_t  *task_handle,                          /* IN */
  ompd_tword_t              *val                                  /* OUT */
);
```

`ompd_get_dynamic` returns the value of the *dyn-var* ICV (§2.3.1 of [5]), and corresponds to the `omp_get_dynamic` member of the OpenMPI API.

```
ompd_rc_t ompd_get_dynamic (
  const ompd_task_handle_t  *task_handle,                            /* IN */
  ompd_tword_t              *val                                    /* OUT */
);
```

dyn-var determines whether dynamic adjustment of the number of threads is enabled or disabled.

`ompd_get_nested` corresponds to `omp_get_nested`, and returns the value of the *nest-var* ICV (§2.3.1 of [5]).

```
ompd_rc_t ompd_get_nested (
  const ompd_task_handle_t  *task_handle,                            /* IN */
  ompd_tword_t              *val                                    /* OUT */
 );
```

nest-var determines if nested parallelism is enabled; a logical true value indicates that it is, false that it is not.

The maximum number of nested levels parallelism is returned by `get_max_active_levels`.

```
ompd_rc_t ompd_get_max_active_levels (
  const ompd_thread_handle_t  *thread_handle,                        /* IN */
  ompd_tword_t                *val                                  /* OUT */
);
```

This operation corresponds to the OpenMP routine `omp_get_max_active_levels` and the ICV *max-active-levels-var* (§2.3.1 of [5]).

`ompd_get_schedule` returns information about the schedule that is applied when `runtime` scheduling is used. This information is represented in the target by the *run-sched-var* ICV (§3.2.1 of [5]).

```
ompd_rc_t ompd_get_schedule (
  ompd_task_handle_t        *task_handle,                            /* IN */
  ompd_sched_t              *kind,                                  /* OUT */
  ompd_tword_t              *modifier                               /* OUT */
);
```

OpenMP defines a minimum set of values in the enumeration type `omp_sched_t` (§3.2.12 of [5]). The OMPD API defines `ompd_sched_t`, which contains the corresponding OpenMP enumeration values and "lo" and "hi" values for the range of implementation-specific scheduling values that can be represented by the OMPD API. The scheduling kind is returned in `*kind`. The interpretation of `*modifier` depends on the value of `*kind`. See §3.2.12 and §3.2.13 of [5] for further details. §15.6, p24

`ompd_get_proc_bind` returns the value of the task's *bind-var* ICV (§2.3.1 of [5]), which "controls the binding of the OpenMP threads to places," or "default thread affinity policies."

```
ompd_rc_t ompd_get_proc_bind (
  ompd_task_handle_t        *task_handle,                            /* IN */
  ompd_proc_bind_t          *bind                                   /* OUT */
);
```

The OMPD API defines `ompd_proc_bind_t`, which contains the corresponding OpenMP enumeration values. The binding is returned in `*bind`. See §3.2.22 of [5] for further details. §15.7, p25

`ompd_is_implicit` returns logical true (*i.e.*, `*val != 0`) if a task is implicit, and false (0) otherwise. The routine has no corresponding call in the OpenMP runtime.

```
ompd_rc_t ompd_is_implicit (
  ompd_task_handle_t  *task_handle,                                  /* IN */
  ompd_tword_t        *val                                          /* OUT */
);
```

11.3 OMPT Task Inquiry Analogues

The function ompd_get_task_frame ia a third-party versions of ompt_get_task_frame. ompd_get_task_frame is discussed under Stack Unwinding in Section 11.4.

```
ompd_rc_t ompd_get_task_frame (
  ompd_task_handle_t        *task_handle,                         /* IN */
  ompd_address_t            *exit_runtime_addr,                  /* OUT */
  ompd_address_t            *reenter_runtime_addr                /* OUT */
);
```

11.4 Stack Unwinding

The ompd_get_task_frame function returns stack frame information about the target thread associated with the task. This routine corresponds to ompt_get_task_frame in the OMPT API, and the approach for stack inspection is similar to that described in Appendix B of [3]. The exit_runtime_addr gives the address of the frame at which the thread *left* the OpenMP runtime to execute the user code associated with the task. The reenter_runtime_addr is the address of the frame that called the OpenMP runtime. The debugger can unwind a thread's stack by getting the thread's current task using ompd_get_top_task_region. Using the task handle, the debugger §7.4, p13 can find the thread's exit and reentry stack frame addresses using ompd_get_task_frame. It can then use ompd_get_scheduling_ancestor_task_region to find the next task in the thread's stack, §7.4, p14 and then call ompd_get_task_frame for the parent task. The frames between the scheduling parent task's reenter address and the top task's exit address are frames in which the thread is executing OpenMP runtime code. This process can be repeated to allow all frames in the thread's backtrace that correspond to execution in the OpenMP runtime to be identified.

Using the generating ancestor operation ompd_get_generating_ancestor_task_region, the §7.4, p14 same process can be used to determine the chain logical ancestors. Note that in this case the logical backtrace may traverse across the stacks of different threads.

The position within the stack frame where the runtime addresses point is implementation defined.

12 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, the OpenMP runtime must define a number of symbols at which the debugger may plant breakpoints to receive notification of particular events. The runtime is expected to execute through these locations when these events occur *and* data collection for OMPD is enabled (see §2). These locations may, but do not have to, be subroutines (see §1.2).

Advice to implementors The debugger needs to be able to detect the beginning of OpenMP runtime code. Especially inline generated runtime code should be built without source line information.

12.1 Parallel Regions

The OpenMP runtime must execute through ompd_bp_parallel_begin when a new parallel region is launched. This should occur after a task encounters a parallel construct, but before any implicit task starts to execute the parallel region's work. Conceptually, the type signature for ompd_bp_parallel_begin is:

```
void ompd_bp_parallel_begin ( void );
```

When the debugger gains control when the breakpoint is triggered, the debugger can map the the operating system thread to an OpenMP thread handle using ompd_get_thread_handle. At this point the handle returned by ompd_get_top_parallel_region is that of the new parallel region. The debugger can find the entry point of the user code that the new parallel region will execute by passing the parallel handle region to ompd_get_parallel_function. The actual number of threads, rather than the requested number of threads, in the team is returned by ompd_get_num_threads. The task handle returned by ompd_get_top_task_region will be that of the task encountering the parallel construct. The 'reenter runtime' address in the information returned by ompd_get_task_frame will be that of the stack frame where the thread called the OpenMP runtime to handle the parallel construct. The 'exit runtime' address will be for the stack frame where the thread left the OpenMP runtime to execute the user code that encountered the parallel construct.

§10.1, p17
§7.3, p12
§9.2, p16
§9.1, p16
§7.4, p13
§11.3, p20

When a parallel region finishes, the OpenMP runtime will cause control to flow through the location ompd_bp_parallel_end. Conceptually, ompd_bp_parallel_end has this type signature, but as with other event notification locations does not need to be a function:

```
void ompd_bp_parallel_end ( void );
```

At this point the debugger can map the operating system thread that hit the breakpoint to an OpenMP thread handle using ompd_get_thread_handle. ompd_get_top_parallel_region returns the handle of the terminating parallel region. ompd_get_top_task_region returns the handle of the task that encountered the parallel construct that initiated the parallel region just terminating. The 'reenter runtime' address in the frame information returned by ompd_get_task_frame will be that for the stack frame in which the thread called the OpenMP runtime to start the parallel construct just terminating. The 'exit runtime' address will refer to the stack frame where the thread left the OpenMP runtime to execute the user code that invoked the parallel construct just terminating.

§10.1, p17
§7.3, p12
§7.4, p13
§11.3, p20

Both the begin and end events are raised once per region, and not once for each thread per region.

12.2 Task Regions

When starting a new task region, the OpenMP runtime system must allow control to pass through ompd_bp_task_begin. Conceptually, ompd_bp_task_end has this type signature, but as with other event notification locations does not need to be a function:

```
void ompd_bp_task_begin ( void );
```

The OpenMP runtime system will execute through this location after the task construct is encountered, but before the new explicit task starts. When the breakpoint is triggered the debugger can map the operating thread to an OpenMP handle using ompd_get_thread_handle. ompd_get_top_task_region returns the handle of the new task region. The entry point of the user code to be executed by the new task from returned from ompd_get_task_function.

§10.1, p17
§7.4, p13
§11.1, p18

When a task region completes, the OpenMP runtime system executes through the location ompd_bp_task_end. If it is implemented as a subroutine, ompd_bp_task_end has this signature:

```
void ompd_bp_task_end ( void );
```

As above, when the breakpoint is hit the debugger can use ompd_get_thread_handle to map the triggering operating system thread to the corresponding OpenMP thread handle. At this point ompd_get_top_task_region returns the handle for the terminating task.

§10.1, p17
§7.4, p13

13 Display Control Variables

Using the ompd_get_display_control_vars function, the debugger can extract a NULL-terminated vector of strings 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).

```
ompd_rc_t ompd_get_display_control_vars (
  ompd_address_space_handle_t   *handle,                          /* IN */
  const char * const *          *control_var_values              /* OUT */
);
```

The format of the strings is:

`name=a string`

The debugger must not modify the vector or strings (*i.e.*, they are both `const`). The strings are NULL terminated. The vector is NULL terminated.

After returning from the call, the vector and strings are 'owned' by the debugger. Providing the termination constraints are satisfied, the OMPD implementation is free to use static or dynamic memory for the vector and/or the strings, and to arrange them in memory as it pleases. If dynamic memory is used, then the OMPD implementation must use the allocate callback it received in the call to `ompd_initialize`. As the debugger cannot make any assumptions about how the memory used for the vector and strings, it cannot release the display control variables directly when they are no longer needed, and instead it must use the `ompd_release_display_control_vars` function: §4.1, p8

```
ompd_rc_t ompd_release_display_control_vars (
  const char * const *           control_var_values               /* IN */
);
```

14 OpenMP Runtime Requirements

Most of the debugger's OpenMP-related activities on a target will be performed through the OMPD interface. However, supporting OMPD introduces some requirements of the OpenMP runtime. Some of these have been discussed earlier. Here we summarize these requirements and collect them together for easy reference.

1. The OpenMP must define `ompd_dll_locations`; §2, p2
2. The OpenMP must define `ompd_dll_locations_valid` and ensure that control flows through it once `ompd_dll_locations` is ready to be read by the debugger; §2, p4
3. In order to support debugging, the OpenMP may need to collect and maintain information about a target's execution that, perhaps for performance reasons, it would not otherwise not do. The OpenMP runtime must support the following mechanisms for indicating that it should collect whatever information is necessary to support OMPD:
   (a) the environment variable `OMP_DEBUG` is set to `on`;
   (b) the *target* calls `omp_debug_enable ()` §2, p4
4. The OpenMP must define the following code symbols, and execute through them at the times described in Section 12:

   `ompd_bp_parallel_begin`
   `ompd_bp_parallel_end`
   `ompd_bp_task_begin`
   `ompd_bp_task_end`
5. Any OMPD-related symbols needed by the debugger must have `C` linkage.

15 OMPD Interface Type Definitions

The ompd.h file contains declarations and definitions for OMPD API types, structures, and functions.

15.1 Basic Types

```
typedef uint64_t ompd_taddr_t;          /* unsigned integer large enough */
                                        /* to hold a target address or a */
                                        /* target segment value          */
typedef int64_t  ompd_tword_t;          /* signed version of ompd_addr_t */
typedef uint64_t ompd_wait_id_t;        /* identifies what a thread is   */
                                        /* waiting for                   */
typedef struct {
  ompd_taddr_t segment;                 /* target architecture specific  */
                                        /* segment value                 */
  ompd_taddr_t address;                 /* target address in the segment */
} ompd_address_t;

#define OMPD_SEGMENT_UNSPECIFIED  ((ompd_taddr_t) 0)
#define OMPD_SEGMENT_TEXT         ((ompd_taddr_t) 1)
#define OMPD_SEGMENT_DATA         ((ompd_taddr_t) 2)
```

An `ompd_address_t` is a structure that OMPD uses to specify target addresses, which may or may not be segmented. The following rules apply:

- If the target architecture is not segmented, the OMPD implementation should use `OMPD_SEGMENT_UNSPECIFIED` for the segment value.
- If the target architecture uses simple “text” and “data” segments, which is common on some systems, the OMPD implementation should use `OMPD_SEGMENT_TEXT` for the text segment value, and `OMPD_SEGMENT_DATA` for the data segment value.
- The segment value for the NVIDIA® GPU target architecture should use a `ptxStorageKind` enumeration value as defined by the CUDA Debugger API. This enumeration is defined by the `cudadebugger.h` header file contained within a CUDA SDK package.
- Otherwise, the segment value is target architecture specific.

15.2 Operating System Thread Information

An OpenMP runtime may be implemented on different threading substrates. OMPD uses the `ompd_osthread_kind_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_winthread
} ompd_osthread_kind_t;
```

The operating system-specific information can vary in size and format, and therefore is not explicitly represented in this API. Operating system-specific thread identifiers are passed across the interface by reference, that is, by a pointer to where the information can be found. In addition, the ‘kind’ and size of the information are also passed.

When operating system-specific thread identifiers are passed as either ‘in’ or ‘out’ parameters, they are allocated and owned by the caller, which is responsible for their eventual disposal.

15.3 OMPD Handles

Each OMPD interface operation that applies to a particular address space, thread, parallel region, or task must explicitly specify the target entity for the operation using a *handle*. OMPD employs handles for address spaces (for a host or target device), threads, parallel regions, and tasks. A handle for an entity is constant while the entity itself is live. Handles are defined by the OMPD implementation, and are opaque to the debugger. This is how the `ompd.h` header file defines these types:

```
typedef struct _ompd_address_space_handle_s  ompd_address_space_handle_t;
typedef struct _ompd_thread_handle_s         ompd_thread_handle_t;
typedef struct _ompd_parallel_handle_s       ompd_parallel_handle_t;
typedef struct _ompd_task_handle_s           ompd_task_handle_t;
```

Defining the externally visible type names in this way introduces an element of type safety to the interface, and will help to catch instances where incorrect handles are passed by the debugger to the OMPD implementation. The `structs` do not need to be defined at all. The OMPD implementation would need to cast incoming (pointers to) handles to the appropriate internal, private types.

15.4 Debugger Contexts

The debugger contexts are opaque to the OMPD, and are defined in the `ompd.h` header file as follows:

```
typedef struct _ompd_address_space_context_s  ompd_address_space_context_t;
typedef struct _ompd_thread_context_s         ompd_thread_context_t;
```

15.5 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                */
  ompd_rc_bad_input    =  3, /* bad input parameters (other than handle) */
  ompd_rc_error        =  4, /* error                                    */
  ompd_rc_unsupported  =  5, /* operation is not supported               */
  ompd_rc_needs_state_tracking =  6,
                             /* needs runtime state tracking enabled     */
  ompd_rc_incompatible =  7, /* target is not compatible with this OMPD  */
  ompd_rc_target_read_error =  8,
                             /* error reading from the target            */
  ompd_rc_target_write_error =  9,
                             /* error writing from the target            */
  ompd_rc_nomem        = 10, /* unable to allocate memory                */
} ompd_rc_t;
```

15.6 OpenMP Scheduling

This enumeration defines `ompd_sched_t`, which is the OMPD API definition corresponding to the OpenMP enumeration type `omp_sched_t` (§3.2.12 of [5]). `ompd_sched_t` also defines `ompd_sched_vendor_lo` and `ompd_sched_vendor_hi` to define the range of implementation-specific `omp_sched_t` values than can be handle by the OMPD API.

```
typedef enum {
  ompd_sched_static = 1,
  ompd_sched_dynamic = 2,
  ompd_sched_guided = 3,
  ompd_sched_auto = 4,
  ompd_sched_vendor_lo = 5,
  ompd_sched_vendor_hi = 0x7fffffff
} ompd_sched_t;
```

15.7 OpenMP Proc Binding

This enumeration defines `ompd_proc_bind_t`, which is the OMPD API definition corresponding to the OpenMP enumeration type `omp_proc_bind_t` (§3.2.22 of [5]).

```
typedef enum {
  ompd_proc_bind_false = 0,
  ompd_proc_bind_true = 1,
  ompd_proc_bind_master = 2,
  ompd_proc_bind_close = 3,
  ompd_proc_bind_spread = 4
} ompd_proc_bind_t;
```

15.8 Primitive Types

This structure contains members that the OMPD implementation can use to interrogate the debugger about the “sizeof” of primitive types in the target address space.

```
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;
```

This enumeration of primitive types is used by OMPD to express the primitive type of data for target to host conversion.

```
typedef enum {
  ompd_type_char        = 0,
  ompd_type_short       = 1,
  ompd_type_int         = 2,
  ompd_type_long        = 3,
  ompd_type_long_long   = 4,
  ompd_type_pointer     = 5
} ompd_target_prim_types_t;
```

15.9 Runtime States

The OMPD runtime states mirror those in OMPT (see Appendix A of [3]). Note that there is no guarantee that the numeric values of the corresponding members of the enumerations are identical.

```
typedef enum {
  /* work states (0..15) */
```

```
    ompd_state_work_serial          = 0x00,   /* working outside parallel */
    ompd_state_work_parallel        = 0x01,   /* working within parallel  */
    ompd_state_work_reduction       = 0x02,   /* performing a reduction   */

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

    /* overhead states (32..63) */
    ompd_state_overhead             = 0x20,   /* non-wait overhead        */

    /* barrier wait states (64..79) */
    ompd_state_wait_barrier         = 0x40,   /* generic barrier          */
    ompd_state_wait_barrier_implicit = 0x41,   /* implicit barrier         */
    ompd_state_wait_barrier_explicit = 0x42,   /* explicit barrier         */

    /* task wait states (80..95) */
    ompd_state_wait_taskwait        = 0x50,   /* waiting at a taskwait    */
    ompd_state_wait_taskgroup       = 0x51,   /* waiting at a taskgroup   */

    /* mutex wait states (96..111) */
    ompd_state_wait_mutex           = 0x60,   /* waiting for any mutex kind
                                                                          */
    ompd_state_wait_lock            = 0x61,   /* waiting for lock         */
    ompd_state_wait_critical        = 0x62,   /* waiting for critical     */
    ompd_state_wait_atomic          = 0x63,   /* waiting for atomic       */
    ompd_state_wait_ordered         = 0x64,   /* waiting for ordered      */

    /* misc (112..127) */
    ompd_state_undefined            = 0x70,   /* undefined thread state    */
    ompd_state_first                = 0x71,   /* initial enumeration state */
  } ompd_state_t;
```

15.10 Type Signatures for Debugger Callbacks

For OMPD to provide information about the internal state of the OpenMP runtime system in a target process or core file, it must have a means to extract information from the target. A target "process" may be a "live" process or a core file. A target thread may be a "live" thread in a process, or a thread in a core file. To enable OMPD to extract state information from a target process or core file, a debugger supplies OMPD with callback functions to inquire about the size of primitive types in the target, look up the addresses of symbols, 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. The OMPD DLL *must* obtain and release heap memory *only* using the callbacks provided to it by the debugger. It must *not* call the heap manager directly using `malloc`. For C++ implementations this means the OMPD implementation *must* overload the functions `new`, `new(throw)`, `new[]`, `delete`, `delete(throw)`, and `delete[]` in *all* their variants and use the debugger-provided callback functions to implement them.

```
  typedef ompd_rc_t (*ompd_dmemory_alloc_fn_t) (
    ompd_size_t bytes,              /* IN: the number of bytes to allocate */
    void   **ptr /* OUT: on success, a pointer to the allocated memory here */
  );
```

```
typedef ompd_rc_t (*ompd_dmemory_free_fn_t) (
  void *ptr           /* IN: the address of the memory to be deallocated */
);
```

Context management. The callback signature below is used to map an operating system thread handle to a debugger thread context. The OMPD implementation can use this thread context to access thread local storage (TLS).

```
typedef ompd_rc_t (*ompd_get_thread_context_for_osthread_fn_t) (
  ompd_address_space_context_t  *address_space_context,            /* IN */
  ompd_osthread_kind_t           kind,                             /* IN */
  ompd_size_t                    sizeof_osthread,                  /* IN */
  const void                    *osthread,                         /* IN */
  ompd_thread_context_t        **thread_context                   /* OUT */
);
```

On success, the `ompd_thread_context_t` corresponding to the operating system thread identifier `*osthread` of type `kind` and size `sizeof_osthread` is returned in `*thread_context`. The thread context is created, and remains owned, by the debugger. The OMPD implementation can assume that the thread context is valid for as long as the debugger is holding any references to thread handles that may contain the thread context.

Context navigation. The following callback signature is used to "navigate" address space and thread object relationships.

Thread context to address space context. Given a thread context, get the address space context for the thread and return it in `*address_space_context`. If `thread_context` refers to a host device thread, this function returns the context for the host address space. If `thread_context` refers to a target device thread, this function returns the context for the target device's address space.

```
typedef ompd_rc_t (*ompd_get_address_space_context_for_thread_fn_t) (
  ompd_thread_context_t         *thread_context,                 /* IN */
  ompd_address_space_context_t **address_space_context          /* OUT */
);
```

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

```
typedef ompd_rc_t (*ompd_tsizeof_prim_fn_t) (
  ompd_address_space_context_t  *context,                          /* IN */
  ompd_target_type_sizes_t *sizes             /* OUT: returned type sizes */
);
```

Symbol lookup. The callback signature below is used to look up the address of a global symbol in the target. The argument `thread_context` is optional for global memory access and is NULL in this case. If the `thread_context` argument is not NULL, this will give the thread specific context for the symbol lookup, for the purpose of calculating thread local storage (TLS) addresses.

```
typedef ompd_rc_t (*ompd_tsymbol_addr_fn_t) (
  ompd_address_space_context_t *address_space_context,             /* IN */
  ompd_thread_context_t        *thread_context, /* IN: TLS thread or NULL */
  const char                   *symbol_name,    /* IN: global symbol name */
  ompd_address_t               *symbol_addr           /* OUT: on success, */
                                                    /* the symbol address */
);
```

The symbol name supplied by the OMPD implementation is used verbatim by the debugger, and in particular, no name mangling is performed prior to the lookup.

Memory access. The callback signatures below are used to read or write memory in the target. Data transfers are of unstructured bytes; it is the responsibility of the OMPD implementation to arrange for any byte swapping as necessary. The argument `thread_context` is optional for global memory access and is NULL in this case. If the argument is not NULL, it identifies the thread specific context for the memory access, for the purpose of accessing thread local storage (TLS) memory. The buffer is allocated and owned by the OMPD implementation.

```
typedef ompd_rc_t (*ompd_tmemory_read_fn_t) (
  ompd_address_space_context_t *address_space_context,             /* IN */
  ompd_thread_context_t        *thread_context, /* IN: TLS thread or NULL */
  const ompd_address_t         *addr,        /* IN: address in the target */
  ompd_tword_t                  nbytes,      /* IN: number of bytes to be */
                                                          /* transferred */
  void                         *buffer  /* OUT: buffer for data read from */
                                                           /* the target */
);

typedef ompd_rc_t (*ompd_tmemory_write_fn_t) (
  ompd_address_space_context_t *address_space_context,             /* IN */
  ompd_thread_context_t        *thread_context, /* IN: TLS thread or NULL */
  const ompd_address_t         *addr,        /* IN: address in the target */
  ompd_tword_t                  nbytes,      /* IN: number of bytes to be */
                                                         /* transferred  */
  const void                   *buffer  /* IN: buffer for date written to */
                                                           /* the target */
);
```

Data format conversion. The callback signature below is used to convert data from the target address space byte ordering to the host (OMPD implementation) byte ordering, and vice versa.

```
typedef ompd_rc_t (*ompd_target_host_fn_t) (
  ompd_address_space_context_t *address_space_context,             /* IN */
  const void                   *input,                             /* IN */
  int                           unit_size,                         /* IN */
  int                           count,    /* IN: number of primitive type */
                                                     /* items to process */
  void                         *output                            /* OUT */
);
```

The input and output buffers are allocated and owned by the OMPD implementation, and it is its responsibility to ensure that the buffers are the correct size.

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) (
  const char              *string
);
```

16 Debugger Callback Interface

OMPD must interact with both the debugger and an OpenMP target process or core file. 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, 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             */

  /* 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 in the target  */
  ompd_tsymbol_addr_fn_t   t_symbol_addr_lookup;  /* look up the address of a symbol       */


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

  /* convert byte ordering */
  ompd_target_host_fn_t target_to_host;
  ompd_target_host_fn_t host_to_target;

  /*--------------------------------------------------------------------------------------*/
  /* context management                                                                   */
  /*--------------------------------------------------------------------------------------*/

  ompd_get_thread_context_for_osthread_fn_t get_thread_context_for_osthread;

  /*--------------------------------------------------------------------------------------*/
  /* context navigation                                                                   */
  /*--------------------------------------------------------------------------------------*/

  ompd_get_address_space_context_for_thread_fn_t
  get_address_space_context_for_thread;

} ompd_callbacks_t;
```

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] A. Eichenberger, J. Mellor-Crummey, M. Schulz, N. Copty, J. Cownie, R. Dietrich, X. Liu, E. Loh, D. Lorenz, and other members of the OpenMP Tools Working Group. OMPT: An OpenMP tools application programming interface for performance analysis. Technical Report TR2, The OpenMP Architecture Review Board, March 2014.

[4] A. Eichenberger, J. Mellor-Crummey, M. Schulz, N. Copty, J. DelSignore, R. Dietrich, X. Liu, E. Loh, D. Lorenz, and other members of the OpenMP Tools Working Group. OMPT and OMPD: OpenMP tools application programming interfaces for performance analysis and debugging. Technical report, The OpenMP Architecture Review Board, April 2013. (This document was superseded by TR2 [3]).

[5] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, version 4.0 edition, July 2013.

[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.