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

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

Nawal Copty[§], Jim Cownie[¶], Robert Dietrich[‖], John Del Signore[**], Eugene Loh[§], Daniel Lorenz[††]
and other members of the OpenMP Tools Working Group

March 21, 2014

1 Introduction

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 [**?**] and MPI [**?**]. A 2003 paper describes a previous effort to define a debugging support library for OpenMP [**?**].

Here, we define OMPD, a shared library companion to an OpenMP runtime system that a debugger can load to help interpret the state of the runtime in a live process or a core file.

1.1 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.

[*]IBM T.J. Watson Research Center
[†]Rice University
[‡]Lawrence Livermore National Laboratory
[§]Oracle
[¶]Intel
[‖]TU Dresden, ZIH
[**]Rogue Wave
[††]Jülich Supercomputer Center

- 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 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`, `ompt_task_id_t`, and `ompt_parallel_id_t` data structures.

2 OpenMP Runtime Interface

As part of the OpenMP interface, we simply require that the OpenMP runtime system provides a public variable `ompd_dll_locations`, which is an argv-style array of filename strings that provides the location(s) of any compatible OMPD plugin implementations (if any).

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

The value of `ompd_dll_locations` may be NULL initially, but must be filled in before `ompt_initialize` is called. After that, `ompd_dll_locations` will point to a vector of zero or more NULL-terminated pathname strings. There are no filename conventions for pathname strings. The last entry in the vector will be NULL.

If OMPT runtime state tracking has been enabled for a runtime [**?**], the OpenMP runtime system will maintain information about the state of each OpenMP thread that will be available to an OMPD plugin, including the following types of information: `ompt_state_t`, `ompt_wait_id_t`, `ompt_frame_t`, `ompt_task_id_t`, and `ompt_parallel_id_t` data structures.

3 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 ompd_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 **??**.

4 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.

4.1 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
);
```

4.2 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
 );
```

4.3 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
);
```

5 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
);
```

6 Parallel Region Inquiries

6.1 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.

```
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 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
);
```

6.2 OMPT Parallel Region Inquiry Analogues

The function `ompd_get_parallel_id` is a third-party variant of `ompt_get_parallel_id` . The only difference between the OMPD and OMPT version 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
);
```

7 Thread Inquiries

7.1 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.*

7.2 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 Task Inquiries

8.1 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
);
```

8.2 OMPT Task Inquiry Analogues

The functions defined here are third-party versions of `ompt_get_task_frame` and `ompt_get_task_id`. 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_id(
  ompd_context_t *context,
  ompd_task_handle_t task_handle,
  ompd_task_id_t *task_id
);
```

9 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
);
```

10 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
);
```

11 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.

12 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
```

13 OMPD 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 **??**) and support all OMPT tool-facing inquiry functions (as described in Section **??**).

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.

14 OMPD Interface Type Definitions

14.1 Basic Types

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

14.2 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_winthread;
} ompd_osthread_kind_t;

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

14.3 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;
```

14.4 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;
```

14.5 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;
```

14.6 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
```

15 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;
```