D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

November 17, 2012

This work was supported in part by NSF and ARPA under NSF contract CDA-9115428 and Esprit under project HPC Standards (21111).

This is the result of a LaTeX run of a draft of a single chapter of the MPIF Final Report document.

13.1 MPI Handle Debugging

Random notes:

- George questions the need for the `mpidbg_process_t` structure (and the mapping that it provides to MPI_COMM_WORLD. What does Allinea think of this?
- Some additional open issues throughout the text are marked with "JMS".

13.2 Premise

Debuggers can display the value of intrinsic datatypes (e.g., int, double, etc.). Even composite struct instances can be displayed. A logical extension to this concept is that the debugger should be able to show information about MPI opaque handles in a running application (and possibly cache such values for playback when the application is not running, such as during corefile analysis). Similar in spirit to the API for obtaining the message passing queues, a simple API can be used between the debugger and the MPI application to obtain information about specific MPI handle instances.

13.3 Background

MPI defines several types of opaque handles that are used in applications (e.g., `MPI_Comm`, `MPI_Request`, etc.). The opaque handles are references to underlying MPI objects that are private to the MPI implementation. These objects contain a wealth of information that can be valuable to display in a debugging context.

Implementations typically have a different underlying type and then typedef the MPI-specified name to the underlying type. For example, Open MPI has the following in mpi.h:

```
typedef struct ompi_communicator_t *MPI_Comm;
```

The MPI-specified type is `MPI_Comm`; the OMPI-specific type is `struct ompi_communicator_t *`. The debugger cannot simply deduce the "real" types of MPI handles by looking at the types of well-known global variables (such as MPI_COMM_WORLD) because these names may be preprocessor macros in mpi.h; the real name of the symbol may be implementation-dependent and not easy to guess.

Hence, if `MPI_*` types are unable to be found by the debugger within the application image, the MPI implementation will need to provide a list of the actual types to the debugger when the MPI handle interpretation functionality is initialized. Once the debugger knows the types of MPI handles, it can provide context-sensitive information when displaying the values of variables within the application image.

Some MPI implementations use integers for handles in C. Such implementations are strongly encouraged to use `typedef` for creating handle types in mpi.h (vs. using #define), such as:

```
typedef int MPI_Comm;
typedef int MPI_Datatype;
/* etc. */
```

So that the debugger can identify a variable as an `MPI_Comm` (vs. an `int`) and therefore know that it is an MPI communicator.

In Fortran's mpif.h/mpi module, however, all MPI handles are defined by the MPI standard to be of type `INTEGER`. As such, there is no way for the debugger to automatically know that a given `INTEGER` variable is an MPI handle, nor which kind of MPI handle. It is therefore up to the debugger's UI to allow users to designate specific `INTEGER` variables as a given MPI handle type.

In the MPI-3 Fortran `mpi_f08` module, handles are distinct types, and, unlike C handles, there is no room for interpretation for their actual type. All `mpi_f08` handle types are derived types with a single integer variable. For example, the following is the communicator handle type:

```
type, BIND(C) :: MPI_Comm
   integer :: MPI_VAL
end type MPI_Comm
```

MPI handles can be "freed" by the application, but this actually only marks the underlying MPI object for freeing; the object itself may not be freed until all corresponding handles and pending actions have completed. Additionally, for debugging purposes, an MPI implementation can choose to *never* free MPI objects (in order to show that they were marked as freed and/or actually freed).

13.4 Assumptions

Some terminology:

- host: machine and process on which debugging process is executing.
- debugger: machine and debugging UI, where the debugger is running. The two machines may be distinct from a hardware point of view, they may have differing endinanness, word-size, etc.

MPI typically denotes function names in all capitol letters (MPI_INIT) as a language-neutral form. The Fortran binding for the function is dependent upon the compiler; the C binding for the function capitolizes the "MPI" and the first letter of the next token (e.g., MPI_Init). This text will use the language-neutral names for all MPI function names.

The debugger will access the handle-interpretation functionality by loading a plugin provided by the MPI implementation into its process space. The plugin will contain "query" functions that the debugger can invoke to obtain information about various MPI handle types. The query functions generally return the additional information or "not found" kinds of errors.

The MPI-implementation-provided plugin shares many of the same characteristics as the Etnus MPI message queue plugin design, but is loaded slightly differently. The plugin will use the `mqs_*()` functions defined by the Etnus message queue access interface to read the process image to obtain MPI object information that is then passed back to the debugger.

The plugin's query functions should not be called before MPI_INIT has completed nor after MPI_FINALIZE has started. MPI handles are only meaningful between the time that MPI_INIT completes and MPI_FINALIZE starts, anyway.

When MPI handles are marked for freeing by the MPI implementation, there should be some notice from the debugger that the underlying objects should not *actually* be freed, but rather orphaned. The debugger can track these objects and keep a reference count of how many handles are still referring to the underlying objects. When the reference count goes to 0, the debugger can call a function in the application telling the MPI implementation that the object is safe to be freed.

In this way, valuable debugging information is still available to the user if they have stale MPI handles because the underlying object will still be available in memory (marked as "stale"); but MPI object memory usage is not cumulative.

The debugger may not be able to provide any additional information in Fortran mpif.h/mpi module subroutines because all MPI handles are of type `INTEGER` (and there's no way to tell that any given integer is an MPI communicator handle, for example) unless the debugger provides some way in the UI to indicate that a given `INTEGER` variable is a specific type of MPI handle.

Note that the following pattern is valid in MPI:

```
MPI_Request a, b;
MPI_Isend(..., &a);
b = a;
MPI_Request_free(&a);
```

After executing MPI_REQUEST_FREE, the handle "a" has been set to MPI_REQUEST_NULL, but the handle "b" still points to the (potentially ongoing) request. "b" would therefore report that it has been marked for freeing by the application, but would not report that it was completed / marked for freeing by the MPI implementation until the corresponding ISEND actually completes.

The query functions will return newly-allocated structs of information to the debugger (allocated via mqs_malloc()). The debugger will be responsible for freeing this memory. Arrays/lists of information contained in the structs must be individually freed if they are not NULL (i.e., they will each be allocated via mqs_malloc()).

Finally, note that not all of this needs to be supported by the debugger at once. Interpretation of some of the more common handle types can be implemented first (e.g., communicators and requests), followed by more types over time.

13.5 MPI handle types

13.5.1 Communicator

- C: `MPI_Comm`
- C++: `MPI::Comm`, `MPI::Intracomm`, `MPI::Intercomm`, `MPI::Cartcomm`, `MPI::Graphcomm`
- MPI F08: `type, BIND(C) :: MPI_Comm`

A communicator is an ordered set of MPI processes and a unique communication context. There are 3 predefined communicators (MPI_COMM_WORLD, MPI_COMM_SELF, MPI_COMM_PARENT), and applications can create their own communicators. There are two types of communicators:

1. Intracommunicator: contains one group of processes. Intracommunicators may optionally have a topology:
   - Cartesian: a dense, N-dimensional Cartesian topology
   - Graph: an arbitrary unweighted, undirected, connected graph
2. Intercommunicator: contains a local group and a remote group of processes.

Information available from communicators:

- String name of length MPI_MAX_OBJECT_NAME
- Flag indicating whether it's a predefined communicator or not
- Whether the handle has been marked for freeing by the application or not
- This process' unique ID within the communicator ("rank", from $0 - (N - 1)$)
- This process' unique ID in the entire MPI universe
- Number of peer processes in the communicator
- Type of communicator: inter or intra
  - If Inter, the number and list of peer processes in the communicator
  - If intra, whether the communicator has a graph or cartesian topology
  - If have a Cartesian toplogy (mutually exclusive with having a graph topology), the topology information:
    * "ndims", "dims", "periods" arguments from the corresponding call to MPI_CART_CREATE, describing the Cartesian topology.
  - If have a graph topology (mutually exclusive with having a cartesian topology):
    * "index" and "edges" arguments from the corresponding call to MPI_GRAPH_CREATE, describing the nodes and edges in the graph topology
- Cross reference to the underlying MPI group(s)
- Cross reference to the underlying MPI error handler
- C handle (if available)
- Fortran integer index for this handle (if available)

Extra/bonus information that the MPI may be able to provide:

- A list of MPI attributes that are attached to the communicator
- A list of any MPI requests currently associated with the communicator (e.g., ongoing or completed but not released communicator requests, or, in multi-threaded scenarios, ongoing communicator operations potentially from other threads)
- Whether the underlying object has been freed by the MPI implementation (i.e., made inactive, and would have actually been freed if not running under a debugger)
- A list of MPI windows using this communicator
- A list of MPI files using this communicator

13.5.2 Datatype

- C: `MPI_Datatype`
- C++: `MPI::Datatype`
- MPI F08: `type, BIND(C) :: MPI_Datatype`

MPI datatypes are used to express the size and shape of user-defind messages. For example, a user can define an MPI datatype that describes a C structure, and can then use that MPI datatype to send and receive instances (or arrays) of that struct. There are many predefined datatypes, and applications can create their own datatypes.

Information available from datatypes:

- String name
- Flag indicating whether it's a predefined datatype or not
- C handle (if available)
- Fortran integer index for this handle (if available)
- A type map of the overall datatype, composed of *only* intrinsic MPI datatypes (MPI_INT, MPI_DOUBLE, etc.) that can be rendered by the debugger into a form similar to MPI-1:3.12
- What function was used to create the datatype:
  - <MPI_TYPE_>CREATE_DARRAY, CREATE_F90_COMPLEX, CREATE_F90_INTEGER, CREATE_F90_REAL, CREATE_HINDEXED, CREATE_HVECTOR, CREATE_INDEXED_BLOCK, CREATE_RESIZED, CREATE_STRUCT, CREATE_SUBARRAY
  - <MPI_>TYPE_HINDEXED, TYPE_INDEXED, TYPE_HVECTOR, TYPE_VECTOR, TYPE_STRUCT, TYPE_CONTIGUOUS,

NOTE: with the type map provided by MPI, a debugger can show "holes" in a datatype (potentially indicating missed optimizations by the application). Very cool/useful!

Extra/bonus information that the MPI may be able to provide:

- Ongoing communication actions involving the datatype (point-to-point, collective, one-sided)
- Whether the handle has been marked for freeing by the application or not
- Whether the underlying object has been freed by the MPI implementation (i.e., made inactive, and would have actually been freed if not running under a debugger)
- Whether the datatype has been committed or not
- A list of datatypes used to create this datatype (JMS: may require caching by the debugger!!)

13.5.3 Error handler

- C: `MPI_Errhandler`
- C++: `MPI::Errhandler`
- MPI F08: `type, BIND(C) :: MPI_Errhandler`

MPI allows applications to define their own error handlers. The default error handler is to abort the MPI job. Error handlers can be attached to communicators, files, and windows. There are 3 predefined error handlers (MPI_ERRORS_ARE_FATAL, MPI_ERRORS_RETURN, MPI::ERRORS_THROW_EXCEPTIONS), and applications can create their own error handlers.

Information available from error handlers:

- Flag indicating whether it's a predefined error handler or not
- C handle (if available)
- Fortran integer index for this handle (if available)
- Type of errorhandler: communicator, file, window
- If user-defined (i.e., not predefined), the function pointer for the user function that MPI will invoke upon error
- Whether the callback function is in C, C++, Fortran mpi.h/mpi module, or Fortran mpi_f08 module.

Extra/bonus information that the MPI may be able to provide:

- String name for predefined handles
- Whether the handle has been marked for freeing by the application or not
- Whether the underlying object has been freed by the MPI implementation (i.e., made inactive, and would have actually been freed if not running under a debugger)
- List of communicators/files/windows that this error handler is currently attached to

13.5.4 File

- C: `MPI_File`
- C++: `MPI::File`
- MPI F08: `type, BIND(C) :: MPI_File`

MPI has the concept of parallel IO, where a group of processes collectively open, read/write, and close files. An `MPI_File` handle represents both an ordered set of processes and a file to which they are accessing. There is one pre-defined file: MPI_FILE_NULL; applications can open their own files.

Information available from files:

- String file name (or MPI_FILE_NULL)

- Flag indicating whether it's a predefined file or not
- C handle (if available)
- Fortran integer index for this handle (if available)
- Communicator that the file was opened with
- Info key=value pairs that the file was opened with
- Mode that the file was opened with

Extra/bonus information that the MPI may be able to provide:

- Whether the handle has been marked for freeing (closing) by the application or not
- Whether the underlying object has been freed by the MPI implementation (i.e., made inactive, and would have actually been freed if not running under a debugger)
- A list of any MPI requests currently associated with the file (e.g., ongoing or completed but not released file requests, or, in multi-threaded scenarios, ongoing file operations potentially from other threads)

13.5.5 Group

- C: `MPI_Group`
- C++: `MPI::Group`
- MPI F08: `type, BIND(C) :: MPI_Group`

An unordered set of processes. There are predefined and user-defined groups. Every communicator contains exactly 1 or 2 groups (depending on the type of communicator). There are 2 predefined groups (MPI_GROUP_NULL and MPI_GROUP_EMPTY); applications can create their own groups.

Information available from groups:

- C handle (if available)
- Fortran integer index for this handle (if available)
- This process' unique ID in this group
- List of peer processes in this group

Extra/bonus information that the MPI may be able to provide:

- String name for predefined handles
- Whether the handle has been marked for freeing by the application or not
- Whether the underlying object has been freed by the MPI implementation (i.e., made inactive, and would have actually been freed if not running under a debugger)
- A list of MPI communicators using this group

13.5.6 Info

- C: `MPI_Info`
- C++: `MPI::Info`
- MPI F08: `type, BIND(C) :: MPI_Info`

A set of key=value pairs (the key and value are separate strings) that can be used to pass "hints" to MPI. There is only one predefined info handle, MPI_INFO_NULL; applications can create their own info handles.

Information available from info:

- C handle (if available)
- Fortran integer index for this handle (if available)
- Number of key=value pairs on the info
- List of key=value pairs (each key and value is an individual string)

Extra/bonus information that the MPI may be able to provide:

- Whether the handle has been marked for freeing by the application or not
- Whether the underlying object has been freed by the MPI implementation (i.e., made inactive, and would have actually been freed if not running under a debugger)
- A list of places where the info object is currently being used

13.5.7 Request

- C: `MPI_Request`
- C++:: `MPI::Request`, `MPI::Grequest`, `MPI::Prequest`
- MPI F08: `type, BIND(C) :: MPI_Request`

A pointer to an ongoing or completed-but-not-yet-released action. There are three types of requests:

- Point-to-point communication: non-blocking sends, receives
- File actions: non-blocking reads, writes
- Generalized actions: Users can define their own asynchronous actions that can be subject to MPI completion semantics
- Nonblocking collectives (new in MPI-3): ibcast and friends
- RMA operations (new in MPI-3): from MPI_RPUT, MPI_RGET, etc.

There is one predefined request (MPI_REQUEST_NULL); applications can create their own requests.

Information available from requests:

- Flag indicating whether it's a predefined request or not
- Flag indicating whether the request is persistent or not
- C handle (if available)
- Fortran integer index for this handle (if available)
- Type of the request: pt2pt, file, generalized, collective, RMA
- Function that created this request:
  - Pt2pt: <MPI_>ISEND, IBSEND, ISSEND, IRSEND, IRECV, SEND_INIT, BSEND_INIT, SSEND_INIT, RSEND_INIT, RECV_INIT
  - File: <MPI_FILE_>IREAD, IREAD_AT, IREAD_SHARED, IWRITE, IWRITE_AT, IWRITE_SHARED
  - Collective: <MPI_>IBARRIER, IBCAST, IGATHER, IGATHERV, ISCATTER, ISCATTERV, IALLGATHER, IALLGATHERV, IALLTOALL, IALLTOALLV, IALLTOALLW, IREDUCE, IALLREDUCE, IREDUCE_SCATTER_BLOCK, IREDUCE_SCATTER, ISCAN, IEXSCAN
  - - RMA: <MPI_>RGET, RPUT, RACCUMULATE, RGET_ACCUMULATE
- Whether the request has been marked complete or not

Extra/bonus information that the MPI may be able to provide:

- String name for predefined handles
- Whether the handle has been marked for freeing by the application or not
- Whether the underlying object has been freed by the MPI implementation (i.e., made inactive, and would have actually been freed if not running under a debugger)
- Peer process(es) invovled with the request (if available)
- If pt2pt, communicator associated with the request
- If file, file associated with the request
- If pt2pt or file, whether the data transfer has started yet
- If pt2pt or file, whether the data transfer has completed yet

13.5.8 Operation

- C: `MPI_Op`
- C++:: `MPI::Op`
- MPI F08: `type, BIND(C) :: MPI_Op`

A reduction operator used in MPI collective and one-sided operations (e.g., sum, multiply, etc.). There are several predefined operators; applications can also create their own operators.

Information available from operators:

- Flag indicating whether it's a predefined operator or not
- C handle (if available)
- Fortran integer index for this handle (if available)
- If user-defined, the function pointer for the user function that MPI will invoke
- Whether the callback function is in Fortran or C, C++, Fortran mpif.h/mpi module, or Fortran mpi_f08 module
- Whether the operator is commutative or not

Extra/bonus information that the MPI may be able to provide:

- String name for predefined handles
- Whether the handle has been marked for freeing by the application or not
- Whether the underlying object has been freed by the MPI implementation (i.e., made inactive, and would have actually been freed if not running under a debugger)
- List of ongoing collective / one-sided communications associated with this operator

13.5.9 Status

- C: `MPI_Status`
- C++: `MPI::Status`
- MPI F08: `type, BIND(C) ::  MPI_Status`

A user-accessible struct that contains information about a completed communication. The MPI status is a little different from other MPI handles in that it is the object itselt; not a handle to an underlying MPI status. For example, if a point-to-point communication was started with a wildcard receive, the status will contain information about the peer to whom the communication completed. There are no predefined statuses.

Information available from status:

- Public member MPI_SOURCE: source of the communication
- Public member MPI_TAG: tag of the communication
- Public member MPI_ERROR: error status of the communication
- Number of bytes in the communication

Extra/bonus information that the MPI may be able to provide:

- Number of data elements in the communication

13.5.10 Window

- C: `MPI_Win`
- C++: `MPI::Win`
- MPI F08: `type, BIND(C) :: MPI_Win`

An ordered set of processes, each defining their own "window" of memory for one-sided operations.

Information available from windows:

- Communicator that the window was created with
- Base address, length, and displacement units of the window *in this process*
- Info key=value pairs that the file was opened with

Extra/bonus information that the MPI may be able to provide:

- Whether the handle has been marked for freeing by the application or not
- Whether the underlying object has been freed by the MPI implementation (i.e., made inactive, and would have actually been freed if not running under a debugger)
- Whether LOCK has been called on this window without a corresponding UNLOCK yet
- Whether START has been called on this window without a corresponding COMPLETE yet
- Whether POST has been called on this window without a corresopnding TEST/WAIT yet
- What the last synchronization call was on the window: FENCE, LOCK/UNLOCK, START/COMPLETE, POST/TEST/WAIT

13.5.11 Message (new in MPI-3)

- C: `MPI_Message`
- C++: Not defined in C++
- MPI F08: `type, BIND(C) :: MPI_Message`

A message that has been matched (by MPI_MPROBE or MPI_IMPROBE) but not yet received.

Information available from windows:

- Source of the message
- Tag of the message
- Communicator of the message

- Size (in bytes) of the message

Extra/bonus information that the MPI may be able to provide:

- Whether the handle has been marked for freeing by the application or not
- Whether the underlying object has been freed by the MPI implementation (i.e., made inactive, and would have actually been freed if not running under a debugger)

13.5.12 Address integer

- C: `MPI_Aint`
- C++: `MPI::Aint`
- Fortran (all): `KIND=MPI_ADDRESS_KIND`

This is an MPI-specific type, but is always an integer value that is large enough to hold addresses. It is typically 32 or 64 bits long. Hence, the debugger should be able to directly display this value.

13.5.13 Offset

- C: `MPI_Offset`
- C++: `MPI::Offset`
- Fortran (all): `KIND=MPI_OFFSET_KIND`

This is a MPI-specific type, but is always an integer value that is large enough to hold file offsets. It is typically 32 or 64 bits long. Hence, the debugger should be able to directly display this value.

13.5.14 Count (new in MPI-3)

- C: `MPI_Count`
- C++: None!
- Fortran (all): `KIND=MPI_COUNT_KIND`

This is a MPI-specific type, but is always an integer value that is large enough to hold huge counts. It is typically 64 or 128 bits long. Hence, the debugger should be able to directly display this value.

13.6 Proposed Interfaces

OPEN QUESTIONS

- should all Fortran mpif.h/mpi module `INTEGER` references be `mqs_taddr_t` in case they're not the same size as C int?

Note that we'll also include the Etnus debugger message queue interface so that we can use much of its infrastructure (e.g., the `mqs_basic_callbacks`, `mqs_image_callbacks`, and `mqs_process_callbacks`).

```
enum {
    MPIDBG_MAX_OBJECT_NAME = MPI_MAX_OBJECT_NAME
};
enum {
    MPIDBG_MAX_FILENAME = 1024
};
enum {
    MPIDBG_INTERFACE_VERSION = 1
};
```

13.6.1 Global initialization information for the DLL

Structure containing types for C and C++ MPI handles.

```
struct mpidbg_handle_info_t {
    /* C handle types.  They are typically pointers to something or
       integers. */
    /* Back-end type for MPI_Aint */
    mqs_type *hi_c_aint;
    /* Back-end type for MPI_Comm */
    mqs_type *hi_c_comm;
    /* Back-end type for MPI_Datatype */
    mqs_type *hi_c_datatype;
    /* Back-end type for MPI_Errhandler */
    mqs_type *hi_c_errhandler;
    /* Back-end type for MPI_File */
    mqs_type *hi_c_file;
    /* Back-end type for MPI_Group */
    mqs_type *hi_c_group;
    /* Back-end type for MPI_Info */
    mqs_type *hi_c_info;
    /* Back-end type for MPI_Offset */
    mqs_type *hi_c_offset;
    /* Back-end type for MPI_Op */
    mqs_type *hi_c_op;
    /* Back-end type for MPI_Request */
    mqs_type *hi_c_request;
    /* Back-end type for MPI_Status */
    mqs_type *hi_c_status;
    /* Back-end type for MPI_Win */
    mqs_type *hi_c_win;

    /* C++ handle types.  Note that these will always be *objects*,
       never pointers. */
```

```
    /* Back-end type for MPI::Aint */
    mqs_type *hi_cxx_aint;
    /* Back-end type for MPI::Comm */
    mqs_type *hi_cxx_comm;
    /* Back-end type for MPI::Intracomm */
    mqs_type *hi_cxx_intracomm;
    /* Back-end type for MPI::Intercomm */
    mqs_type *hi_cxx_intercomm;
    /* Back-end type for MPI::Graphcomm */
    mqs_type *hi_cxx_graphcomm;
    /* Back-end type for MPI::Cartcomm */
    mqs_type *hi_cxx_cartcomm;
    /* Back-end type for MPI::Datatype */
    mqs_type *hi_cxx_datatype;
    /* Back-end type for MPI::Errhandler */
    mqs_type *hi_cxx_errhandler;
    /* Back-end type for MPI::File */
    mqs_type *hi_cxx_file;
    /* Back-end type for MPI::Group */
    mqs_type *hi_cxx_group;
    /* Back-end type for MPI::Info */
    mqs_type *hi_cxx_info;
    /* Back-end type for MPI::Offset */
    mqs_type *hi_cxx_offset;
    /* Back-end type for MPI::Op */
    mqs_type *hi_cxx_op;
    /* Back-end type for MPI::Request */
    mqs_type *hi_cxx_request;
    /* Back-end type for MPI::Prequest */
    mqs_type *hi_cxx_prequest;
    /* Back-end type for MPI::Grequest */
    mqs_type *hi_cxx_grequest;
    /* Back-end type for MPI::Status */
    mqs_type *hi_cxx_status;
    /* Back-end type for MPI::Win */
    mqs_type *hi_cxx_win;

    /* Fortran types */
    /* Back-end type for INTEGER(KIND=MPI_ADDRESS_KIND) */
    mqs_type *hi_f_aint;
    /* Back-end type for INTEGER(KIND=MPI_OFFSET_KIND) */
    mqs_type *hi_f_offset;
    /* Back-end type for INTEGER(KIND=MPI_COUNT_KIND) */
    mqs_type *hi_f_count;
};

enum mpidbg_return_codes_t {
    /* Success */
```

```
    MPIDBG_SUCCESS,
    /* Something was not found */
    MPIDBG_ERR_NOT_FOUND,
    /* Something is not supported */
    MPIDBG_ERR_NOT_SUPPORTED,
    /* Something is out of range */
    MPIDBG_ERR_OUT_OF_RANGE,
    /* Something is not available */
    MPIDBG_ERR_UNAVAILABLE,
    /* Ran out of memory */
    MPIDBG_ERR_NO_MEM,
    /* Sentinel max value */
    MPIDBG_MAX_RETURN_CODE
};
```

13.6.2 General data structures

Information about MPI processes

```
struct mpidbg_process_t {
    /* JMS: need something to uniquely ID MPI processes in the
       presence of MPI_COMM_SPAWN */

    /* Global rank in MPI_COMM_WORLD */
    int mpi_comm_world_rank;
};
```

JMS Should we just use `mqs_process_location` instead? George thinks that this is unncessary – perhaps due to the fact that we could use `mqs_process_location`...? Need to get some feedback from others on this one. Need to check Euro PVM/MPI '06 paper...

General name → handle address mappings. This is an optional type that is used to describe MPI's predefined handles if the pre-defined names do not appear as symbols in the MPI process. E.g., if MPI_COMM_WORLD is a #define that maps to some other value, this data structure can be used to map the string "MPI_COMM_WORLD" to the actual value of the handle that it corresponds to (e.g., 0 or a pointer value).

```
struct mpidbg_name_map_t {
    /* Name of the handle */
    char *map_name;

    /* Handle that the name corresponds to.  Will be 0/NULL if there
       is no corresponding back-end object. */
    mqs_taddr_t map_handle;
};
```

MPI attribute / value pairs. Include both a numeric and string key; pre-defined MPI keyvals (e.g., MPI_TAG_MAX) have a human-readable string name. The string will be NULL for non-predefined keyvals.

The int keyval member is last to avoid "holes" in the memory layout.

```
struct mpidbg_attribute_pair_t {
    /* Keyval name; will be non-NULL for attributes that have a
       human-readable name (i.e., MPI predefined keyvals) */
    char *keyval_name;
    /* Value */
    char *value;
    /* Keyval */
    int keyval;
};
```

Arbitrary key=value string pairs. These are used to pass arbitrary, non-semantic information back about a given handle instance. I.e., an MPI implementation can add a list of these key=value strings back to the debugger for a given handle. The debugger won't know what this information means, but it can display the information in the debugger's UI.

```
struct mpidbg_keyvalue_pair_t {
    /* String key name */
    char *key_name;
    /* String value */
    char *value;
};
```

Communicators

Using an enum instead of #define because debuggers can show the *names* of enum values, not just the values.

```
enum mpidbg_comm_capabilities_t {
    /* Whether this MPI DLL supports returning basic information about
       communicators */
    MPIDBG_COMM_CAP_BASIC =                  0x01,
    /* Whether this MPI DLL supports returning names of
       communicators */
    MPIDBG_COMM_CAP_STRING_NAMES =           0x02,
    /* Whether this MPI DLL supports indicating whether a communicator
       has been freed by the user application */
    MPIDBG_COMM_CAP_FREED_HANDLE =           0x04,
    /* Whether this MPI DLL supports indicating whether a communicator
       object has been freed by the MPI implementation or not */
    MPIDBG_COMM_CAP_FREED_OBJECT =           0x08,
    /* Whether this MPI DLL supports returning the list of MPI request
       handles that are pending on a communicator */
    MPIDBG_COMM_CAP_REQUEST_LIST =           0x10,
    /* Whether this MPI DLL supports returning the list of MPI window
       handles that were derived from a given communicator */
    MPIDBG_COMM_CAP_WINDOW_LIST =            0x20,
    /* Whether this MPI DLL supports returning the list of MPI file
       handles that were derived from a given communicator */
    MPIDBG_COMM_CAP_FILE_LIST =              0x40,
```

```
    /* Sentinel max value */
    MPIDBG_COMM_CAP_MAX
};

enum mpidbg_comm_info_bitmap_t {
    /* Predefined communicator if set (user-defined if not set) */
    MPIDBG_COMM_INFO_PREDEFINED =      0x001,
    /* Whether this communicator is a cartesian communicator or not
       (mutually exclusive with _GRAPH and _INTERCOMM) */
    MPIDBG_COMM_INFO_CARTESIAN =       0x002,
    /* Whether this communicator is a graph communicator or not
       (mutually exclusive with _CARTESIAN and _INTERCOMM) */
    MPIDBG_COMM_INFO_GRAPH =           0x004,
    /* If a cartesian or graph communicator, whether the processes in
       this communicator were re-ordered when the topology was
       assigned. */
    MPIDBG_COMM_INFO_TOPO_REORDERED =  0x008,

    /* Whether this is an intercommunicator or not (this communicator
       is an intracommunicator if this flag is not yet). */
    MPIDBG_COMM_INFO_INTERCOMM =       0x010,

    /* This communicator has been marked for freeing by the user
       application if set */
    MPIDBG_COMM_INFO_FREED_HANDLE =    0x020,
    /* This communicator has actually been freed by the MPI
       implementation if set */
    MPIDBG_COMM_INFO_FREED_OBJECT =    0x040,
    /* The queried communicator is MPI_COMM_NULL */
    MPIDBG_COMM_INFO_COMM_NULL =       0x080,

    /* The queried communicator is a C handle */
    MPIDBG_COMM_INFO_HANDLE_C =        0x100,
    /* The queried communicator is a C++ handle */
    MPIDBG_COMM_INFO_HANDLE_CXX =      0x200,
    /* The queried communicator is a F77/F90 integer handle */
    MPIDBG_COMM_INFO_HANDLE_FINT =     0x400,

    /* Sentinel max value */
    MPIDBG_COMM_INFO_MAX
};
```

Predefined handle → address mappings. This enum is the index to an array indicating the handle addresses of the 4 pre-defined communicators.

```
enum mpidbg_predefined_comm_t {
    MPIDBG_COMM_WORLD,
    MPIDBG_COMM_SELF,
```

```
    MPIDBG_COMM_PARENT,
    MPIDBG_COMM_NULL,
    MPIDBG_COMM_MAX
};
```

When a communicator is looked up in an MPI process, the following handle is returned. This handle can be used as a "base class" by the DLL to cache additional information, if desired.

This handle is a distinct type (rather than, for example, a typedef to `(void*)`) to provide compile-time checking, ensuring that handles are not queried from one type and used with another.

```
struct mpidbg_comm_handle_t {
    /* Image that this handle is in */
    mqs_image *image;
    mqs_image_info *image_info;

    /* Process that this handle is in */
    mqs_process *process;
    mqs_process_info *process_info;

    /* Value of the communicator handle in the MPI process (passed in
       via mpidbg_comm_query()) */
    mqs_taddr_t c_comm;
};
```

Requests

Using an enum instead of #define because debuggers can show the *names* of enum values, not just the values.

```
enum mpidbg_request_capabilities_t {
    /* Whether this MPI DLL supports returning basic information about
       requests */
    MPIDBG_REQUEST_CAP_BASIC =           0x01,
    /* Sentinel max value */
    MPIDBG_REQUEST_CAP_MAX
};

enum mpidbg_request_info_bitmap_t {
    /* Predefined request if set (user-defined if not set) */
    MPIDBG_REQUEST_INFO_PREDEFINED =      0x01,
    /* Sentinel max value */
    MPIDBG_REQUEST_INFO_MAX
};
```

See note about `mpidbg_comm_handle_t`, above.

```
struct mpidbg_request_handle_t {
    /* Image that this handle is in */
    mqs_image *image;
    mqs_image_info *image_info;

    /* Process that this handle is in */
    mqs_process *process;
    mqs_process_info *process_info;

    /* Value of the request handle in the MPI process (passed in via
       mpidbg_request_query()) */
    mqs_taddr_t c_request;
};
```

Statuses

```
enum mpidbg_status_capabilities_t {
    /* Whether this MPI DLL supports returning basic information about
       statuses */
    MPIDBG_STATUS_CAP_BASIC =           0x01,
    /* Sentinel max value */
    MPIDBG_STATUS_CAP_MAX
};

enum mpidbg_status_info_bitmap_t {
    /* Predefined status if set (user-defined if not set) */
    MPIDBG_STATUS_INFO_PREDEFINED =      0x01,
    /* Sentinel max value */
    MPIDBG_STATUS_INFO_MAX
};
```

See note about `mpidbg_comm_handle_t`, above.

```
struct mpidbg_status_handle_t {
    /* Image that this handle is in */
    mqs_image *image;
    mqs_image_info *image_info;

    /* Process that this handle is in */
    mqs_process *process;
    mqs_process_info *process_info;

    /* Value of the status handle in the MPI process (passed in
       via mpidbg_status_query()) */
    mqs_taddr_t c_status;
};
```

Error handlers

Using an enum instead of #define because debuggers can show the *names* of enum values, not just the values.

```
enum mpidbg_errhandler_capabilities_t {
    /* Whether this MPI DLL supports returning basic information about
       error handlers */
    MPIDBG_ERRH_CAP_BASIC =           0x01,
    /* Whether this MPI DLL supports returning names of the predefined
       error handlers */
    MPIDBG_ERRH_CAP_STRING_NAMES =    0x02,
    /* Whether this MPI DLL supports indicating whether an error
       handler has been freed by the user application */
    MPIDBG_ERRH_CAP_FREED_HANDLE =    0x04,
    /* Whether this MPI DLL supports indicating whether an error
       handler object has been freed by the MPI implementation or
       not */
    MPIDBG_ERRH_CAP_FREED_OBJECT =    0x08,
    /* Whether this MPI DLL supports returning the list of MPI handles
       that an MPI error handler is attached to */
    MPIDBG_ERRH_CAP_HANDLE_LIST =     0x10,
    /* Sentinel max value */
    MPIDBG_ERRH_CAP_MAX
};

enum mpidbg_errhandler_info_bitmap_t {
    /* Predefined error handler if set (user-defined if not set) */
    MPIDBG_ERRH_INFO_PREDEFINED =     0x01,
    /* Communicator error handler if set */
    MPIDBG_ERRH_INFO_COMMUNICATOR =   0x02,
    /* File error handler if set */
    MPIDBG_ERRH_INFO_FILE =           0x04,
    /* Window error handler if set */
    MPIDBG_ERRH_INFO_WINDOW =         0x08,
    /* Callback is in C if set */
    MPIDBG_ERRH_INFO_C_CALLBACK =     0x10,
    /* Callback is in Fortran if set */
    MPIDBG_ERRH_INFO_FORTRAN_CALLBACK =0x20,
    /* Callback is in C++ if set */
    MPIDBG_ERRH_INFO_CXX_CALLBACK =   0x40,
    /* This errorhandler has been marked for freeing by the user
       application if set */
    MPIDBG_ERRH_INFO_FREED_HANDLE =   0x80,
    /* This errorhandler has actually been freed by the MPI
       implementation if set */
    MPIDBG_ERRH_INFO_FREED_OBJECT =   0x100,
    /* Sentinel max value */
    MPIDBG_ERRH_INFO_MAX
```

```
};
```

Predefined handle → address mappings. This enum is the index to an array indicating the handle addresses of the pre-defined errhandlers.

```
enum mpidbg_predefined_errhandler_t {
    MPIDBG_ERRHANDLER_ARE_FATAL,
    MPIDBG_ERRHANDLER_RETURN,
    MPIDBG_ERRHANDLER_THROW_EXCEPTIONS,
    MPIDBG_ERRHANDLER_NULL,
    MPIDBG_ERRHANDLER_MAX
};
```

See note about `mpidbg_comm_handle_t`, above.

```
struct mpidbg_errhandler_handle_t {
    /* Image that this handle is in */
    mqs_image *image;
    mqs_image_info *image_info;

    /* Process that this handle is in */
    mqs_process *process;
    mqs_process_info *process_info;

    /* Value of the errhandler handle in the MPI process (passed in
       via mpidbg_errhandler_query()) */
    mqs_taddr_t c_errhandler;
};
```

13.6.3 Global variables

Note that mpidbg_dll_locations is in the MPI application; all others are in the DLL.

Array of filenames instantiated *in the application or starter process* (**NOT** in the DLL) that provides an set of locations where DLLs may be found. The last pointer in the array will be a NULL sentinel value. The debugger will scan the entries in the array, find one that matches the debugger (i.e., whether the dlopen works or not), and try to dynamically open the dl_filename. Notes:

1. It is not an error if a dl_filename either does not exist or is otherwise un-openable (the debugger can just try the next match).
2. This array values are not valid until MPIR_Breakpoint.
3. If a filename is absolute, the debugger will attempt to load exactly that. If the filename is relative, the debugger may try a few prefix variations to find the DLL.

```
extern char **mpidbg_dll_locations;
```

Global variable *in the DLL* describing the DLL's capabilties with regards to communicators. This value is valid after a successfull call to mpidbg_init_per_process().

```
extern enum mpidbg_comm_capabilities_t mpidbg_comm_capabilities;
```

Global variable *in the DLL* that is an array of MPI communicator handle names -¿ handle mappings (the last entry in the array is marked by a NULL string value). For example, MPI_COMM_WORLD may not appear as a symbol in an MPI process, but the debugger needs to be able to map this name to a valid handle. MPI implementations not requiring this mapping can either have a NULL value for this variable or have a single entry that has a NULL string value. This variable is not valid until after a successfull call to mpidbg_init_per_process().

```
extern mqs_taddr_t mpidbg_predefined_comm_map[MPIDBG_COMM_MAX];
```

Global variable *in the DLL* describing the DLL's capabilties with regards to error handlers. This value is valid after a successfull call to mpidbg_init_per_process().

```
extern enum mpidbg_errhandler_capabilities_t mpidbg_errhandler_capabilities;
```

Global variable *in the DLL* that is an array of MPI error handler handle names -¿ handle mappings. It is analogous to mpidbg_predefined_comm_map; see above for details.

```
extern mqs_taddr_t mpidbg_predefined_errhandler_map[MPIDBG_ERRHANDLER_MAX];
```

13.6.4 DLL infrastructure functions

This function will be called by the debugger exactly once before any other mpidbg_*() function is called, and before most other global mpidbg_* data is read. It is only necessary to call this function once for a given debugger instantiation. This function will initialize all mpidbg global state, to include setting all relevant global capability flags.

mpidbg_init_once()

Parameters:

- IN: callbacks: Table of pointers to the debugger functions. The DLL need only save the pointer, the debugger promises to maintain the table of functions valid for as long as needed. The table remains the property of the debugger, and should not be altered or deallocated by the DLL. This applies to all of the callback tables.

This function will return:

- MPIDBG_SUCCESS: if all initialization went well
- MPIDBG_ERR_*: if something went wrong.

```
int mpidbg_init_once(const mqs_basic_callbacks *callbacks);
```

mpidbg_interface_version_compatibility()

Query the DLL to find out what version of the interface it supports.

Parameters:

- None.

This function will return:

- MPIDBG_INTERFACE_VERSION

```
int mpidbg_interface_version_compatibility(void);
```

mpidbg_version_string()

Returns a string describing this DLL.

Parameters:

- None

This function will return:

- A null-terminated string describing this DLL.

```
char *mpidbg_version_string(void);
```

mpidbg_dll_taddr_width()

Returns the address width that this DLL was compiled with.

Parameters:

- None

This function will return:

- `sizeof(mqs_taddr_t)`

```
int mpidbg_dll_taddr_width(void);
```

mpidbg_init_per_image()

Setup debug information for a specific image, this must save the callbacks (probably in the mqs_image_info), and use those functions for accessing this image.

The DLL should use the mqs_put_image_info() and mqs_get_image_info() functions to associate whatever information it wants to keep with the image (e.g., all of the type offsets it needs could be kept here). The debugger will call mqs_destroy_image_info() when it no longer wants to keep information about the given executable.

This will be called once for each executable image in the parallel job.

Parameters:

- IN: image: the application image.

- IN: callbacks: Table of pointers to the debugger image-specific functions. The DLL need only save the pointer, the debugger promises to maintain the table of functions valid for as long as needed. The table remains the property of the debugger, and should not be altered or deallocated by the DLL. This applies to all of the callback tables.

- INOUT: handle_types: a pointer to a pre-allocated struct containing `mqs_types` for each of the MPI handle types. Must be filled in with results from mqs_find_type() for each MPI handle type.

This function will return:

- MPIDBG_SUCCESS: if all initialization went well

- MPIDBG_ERR_NOT_SUPPORTED: if the image does not support the MPIDBG interface. In this case, no other mpidbg functions will be invoked on this image (not even mpidbg_finalize_per_image()).

- MPIDBG_ERR_*: if something went wrong.

```
int mpidbg_init_per_image(mqs_image *image,
                          const mqs_image_callbacks *callbacks,
                          struct mpidbg_handle_info_t *handle_types);
```

mpidbg_finalize_per_image()

This function will be called once when an application image that previously had mpidbg_init_per_image() successfully invoked that is now ending (e.g., the debugger is exiting, the debugger has unloaded this image, etc.). This function can be used to clean up any image-specific data.

Parameters:

- IN: image: the application image.

- IN: image_info: the info associated with the application image.

```
void mpidbg_finalize_per_image(mqs_image *image,
                                                mqs_image_info *image_info);
```

mpidbg_init_per_process()

This function will only be called if mpidbg_init_per_image() returned successfully, indicating that the image contains information for MPI handle information. If you cannot tell whether a process will have MPI handle information in it by examining the image, you should return SUCCESS from mpidbg_init_per_image() and use this function to check whether MPI handle information is available in the process.

Set up whatever process specific information we need. For instance, addresses of global variables should be handled here rather than in the image information, because if data may be in dynamic libraries which could end up mapped differently in different processes.

Note that certain global variables are not valid until after this call completes successfully (see above; e.g., mpidbg_comm_capabilities, mpidbg_comm_name_mapping, etc.).

Parameters:

- IN: process: the process
- IN: callbacks: Table of pointers to the debugger process-specific functions. The DLL need only save the pointer, the debugger promises to maintain the table of functions valid for as long as needed. The table remains the property of the debugger, and should not be altered or deallocated by the DLL. This applies to all of the callback tables.
- INOUT: handle_types: the same handle_types that was passed to mqs_init_per_image(). It can be left unaltered if the results from mqs_init_per_image() were sufficient, or modified if necessary to be specific to this process.

This function will return:

- MPIDBG_SUCCESS: if all initialization went well
- MPIDBG_ERR_NOT_SUPPORTED: if the process does not support the MPIDBG interface. In this case, no other mpidbg functions will be invoked on this image (not even mpidbg_finalize_per_process()).
- MPIDBG_ERR_*: if something went wrong.

```
int mpidbg_init_per_process(mqs_process *process,
                            const mqs_process_callbacks *callbacks,
                            struct mpidbg_handle_info_t *handle_types);
```

mpidbg_finalize_per_process()

This function will be called once when an application image that previously had mpidbg_init_per_process() successfully invoked that is now ending (e.g., the debugger is exiting, the debugger has stopped executing this process, etc.). This function can be used to clean up any process-specific data.

Parameters:

- IN: process: the application process.
- IN: process_info: the info associated with the application process.

```
void mpidbg_finalize_per_process(mqs_process *process,
                                 mqs_process_info *process_info);
```

13.6.5 MPI handle query functions

mpidbg_comm_query()

Query a specific `MPI_Comm` handle and, if found and valid, return a handle that can subsequently be queried for a variety of information about this communicator.

Note that the returned handle is only valid at a specific point in time. If the MPI process advances after the handle is returned, the handle should be considered stale and should therefore be freed. This function should be invoked again to obtain a new handle.

The intent behind this design (query the communicator once and return a handle for subsequent queries) is to allow a DDL to scan the MPI process *once* for all the relevant

information about the communicator and cache it locally (presumably on the returned handle). The subsequent queries are then all local, not involving the probing the MPI process – which chould be somewhat cheaper / faster / easier to implement.

Of course, a DLL is free to cache only the communicator value in the handle and actually probe the MPI process in any of the subsequent query functions, if desired. To be clear: this design *allows* for the pre-caching of all the MPI process communicator data, but does not mandate it.

Upon successful return (i.e., returning MPIDBG_SUCCESS), the returned `comm_handle->c_comm` will equal the `c_comm` parameter value. The caller must also eventually free the returned handle via mpidbg_comm_handle_free().

Parameters:

- IN: image: image
- IN: image_info: image info that was previously "put"
- IN: process: process
- IN: process_info: process info that was previously "put"
- IN: comm: communicator handle
- OUT: comm_handle: handle to be passed to the query functions (below)

This function will return:

- MPIDBG_SUCCESS: if the communicator handle is valid, was found, and the OUT parameter was filled in successfully.
- MPIDBG_ERR_NOT_FOUND: if the handle is not valid / found.
- MPIDBG_ERR_STALE: if the handle was previously valid, but is now stale.
- MPIDBG_ERR_NOT_SUPPORTED: if this function is unsupported.

```
int mpidbg_comm_query(mqs_image *image,
                      mqs_image_info *image_info,
                      mqs_process *process,
                      mqs_process_info *process_info,
                      mqs_taddr_t comm,
                      struct mpidbg_comm_handle_t **handle);
```

mpidbg_comm_free()

Free a handle returned by the mpidbg_comm_query() function.

Parameters:

- IN: handle: handle previously returned by mpidbg_comm_query()

This function will return:

- MPIDBG_SUCCESS: if the handle is valid and was freed successfully.
- MPIDBG_ERR_NOT_FOUND: if the handle is not valid / found.

```
int mpidbg_comm_handle_free(struct mpidbg_comm_handle_t *handle);
```

mpidbg_comm_query_basic()

Query a handle returned by mpidbg_comm_query() and, if found and valid, return basic information about the original communicator.

Parameters:

- IN: handle: handle returned by mpidbg_comm_query()
- OUT: comm_name: string name of the communicator, max of MPIDBG_MAX_OBJECT_NAME characters, \0-terminated if less than MPIDBG_MAX_OBJECT_NAME characters.
- OUT: comm_bitflags: bit flags describing the communicator
- OUT: comm_rank: rank of this process in this communicator
- OUT: comm_size: total number of processes in this communicator
- OUT: comm_fortran_handle: `INTEGER` Fortran handle corresponding to this communicator, or MPIDBG_ERR_UNAVAILABLE if currently unavailable, or MPIDBG_ERR_NOT_SUPPORTED if not supported
- OUT: comm_cxx_handle: Pointer to C++ handle corresponding to this communicator, or MPIDBG_ERR_UNAVAILABLE if currently unavailable, or MPIDBG_ERR_NOT_SUPPORTED if not supported
- OUT: comm_extra_info: Array of pointers to `mpidbg_keyvalue_pair_t` instances for extra info that this implementation wants to return to the tool about this communicator. It is likely that the tool will not understand what these key=value pairs will mean, but at a minimum, they should be suitable for display.

`*comm_extra_info` can be NULL if there are no key/value pairs to return. If non-NULL, `*comm_extra_info` will point to an array of pointers to individual `mpigb_keyvalue_pair_t` instances. Note that the memory associated with `*comm_extra_info` is associated with the handle, and will be freed when mpidb_comm_handle_free() is invoked.

This function will return:

- MPIDBG_SUCCESS: if the handle is valid, was found, and the OUT parameters were filled in successfully.
- MPIDBG_ERR_NOT_FOUND: if the handle is not valid / found.
- MPIDBG_ERR_NOT_SUPPORTED: if this function is unsupported.

```
extern int mpidbg_comm_query_basic(
                     struct mpidbg_comm_handle_t *handle,
                     char comm_name[\const{MPIDBG\_MAX\_OBJECT\_NAME}],
                     enum mpidbg_comm_info_bitmap_t *comm_bitflags,
                     int *comm_rank,
                     int *comm_size,
                     int *comm_fortran_handle,
                     mqs_taddr_t *comm_cxx_handle,
                     struct mpidbg_keyvalue_pair_t ***comm_extra_info);
```

mpidbg_comm_query_procs()

Query a handle returned by mpidbg_comm_query() and, if found and valid, return information about the MPI processes in this communicator.

All arrays returned in OUT variables are allocated by this function, but are the responsibility of the caller to be freed.

Parameters:

- IN: comm_handle: handle returned by mpidbg_comm_query()
- OUT: comm_num_local_procs: filled with the length of the `comm_local_procs` OUT array.
- OUT: comm_local_procs: filled with a pointer to an array of `mpidbg_process_t` instances describing the local processes in this communicator.
- OUT: comm_num_remote_procs: filled with the length of the `comm_remote_procs` OUT array. Will be 0 if the communicator is not an intercommunicator.
- OUT: comm_remote_procs: filled with a pointer to an array of `mpidbg_process_t` instances describing the local processes in this communicator. Will be NULL if the communicator is not an intercommunicator.

This function will return:

- MPIDBG_SUCCESS: if the handle is valid, was found, and the OUT parameters were filled in successfully.
- MPIDBG_ERR_NOT_FOUND: if the handle is not valid / found.
- MPIDBG_ERR_NOT_SUPPORTED: if this function is unsupported.

```
int mpidbg_comm_query_procs(struct mpidbg_comm_handle_t *handle,
                            int *comm_num_local_procs,
                            struct mpidbg_process_t **comm_local_procs,
                            int *comm_num_remote_procs,
                            struct mpidbg_process_t **comm_remote_procs);
```

mpidbg_comm_query_topo()

Query a handle returned by mpidbg_comm_query() and, if found and valid, return information about the topology associated with the communicator (if any). It is not an error to call this function with communicators that do not have associated topologies; such communicators will still return MPIDBG_SUCCESS but simply return a value of 0 in the `comm_out_length` OUT parameter.

All arrays returned in OUT variables are allocated by this function, but are the responsibility of the caller to be freed.

Parameters:

- IN: comm_handle: handle returned by mpidbg_comm_query()
- OUT: comm_out_length:

  - For cartesian communicators, filled with the number of dimensions.
  - For graph communicators, filled with the number of nodes.
  - For all other communicators, filled with 0.

- OUT: comm_cart_dims_or_graph_indexes:
  - For cartesian communicators, filled with a pointer to an array of length `*comm_out_length` representing the dimenstion lengths.
  - For graph communicators, filled with a pointer to an array of length `*comm_out_length` representing the node degrees.
  - For all other communicators, filled with NULL.

- OUT: comm_cart_periods_or_graph_edges:
  - For cartesian communicators, filled with a pointer to an array of length `*comm_out_length` representing whether each dimension is periodic or not.
  - For graph communicators, filled with a pointer to an array of length `*comm_out_length` representing the array of edges.
  - For all other communicators, filled with NULL.

This function will return:

- MPIDBG_SUCCESS: if the handle is valid, was found, and the OUT parameters were filled in successfully.
- MPIDBG_ERR_NOT_FOUND: if the handle is not valid / found.
- MPIDBG_ERR_NOT_SUPPORTED: if this function is unsupported.

```
int mpidbg_comm_query_topo(struct mpidbg_comm_handle_t *handle,
                           int *comm_out_length,
                           int **comm_cart_dims_or_graph_indexes,
                           int **comm_cart_periods_or_graph_edges);
```

mpidbg_comm_query_attrs()

Query a handle returned by mpidbg_comm_query() and, if found and valid, return information about the attributes associated with the communicator (if any). It is not an error to call this function with communicators that do not have associated attributes; such communicators will still return MPIDBG_SUCCESS but simply return a value of 0 in the `comm_attrs_length` OUT parameter.

All arrays returned in OUT variables are allocated by this function, but are the responsibility of the caller to be freed.

Parameters:

- IN: comm_handle: handle returned by mpidbg_comm_query()
- OUT: comm_num_attrs: length of the array returned in `comm_attrs`; if 0, the value of `comm_attrs` is undefined.

- OUT: comm_attrs: array of length `comm_attrs_length` containing keyval/value pairs.

This function will return:

- MPIDBG_SUCCESS: if the handle is valid, was found, and the OUT parameters were filled in successfully.
- MPIDBG_ERR_NOT_FOUND: if the handle is not valid / found.
- MPIDBG_ERR_NOT_SUPPORTED: if this function is unsupported.

```
int mpidbg_comm_query_attrs(struct mpidbg_comm_handle_t *comm_handle,
                            int *comm_num_attrs,
                            struct mpidbg_attribute_pair_t *comm_attrs);
```

mpidbg_comm_query_requests()

Query a handle returned by mpidbg_comm_query() and, if found and valid, return an array of pending requests on this communicator.

All arrays returned in OUT variables are allocated by this function, but are the responsibility of the caller to be freed.

Parameters:

- IN: comm_handle: handle returned by mpidbg_comm_query()
- OUT: comm_num_requests: filled with the length of the `comm_pending_requests` array.
- OUT: comm_requests: filled with a pointer to an array of pending requests on this communicator.

This function will return:

- MPIDBG_SUCCESS: if the handle is valid, was found, and the OUT parameters were filled in successfully.
- MPIDBG_ERR_NOT_FOUND: if the handle is not valid / found.
- MPIDBG_ERR_NOT_SUPPORTED: if this function is unsupported.

```
int mpidbg_comm_query_requests(struct mpidbg_comm_handle_t *handle,
                               int *comm_num_requests,
                               mqs_taddr_t **comm_pending_requests);
```

mpidbg_comm_query_derived()

Query a handle returned by mpidbg_comm_query() and, if found and valid, return arrays of `MPI_File` and `MPI_Win` handles that were derived from this communicator.

All arrays returned in OUT variables are allocated by this function, but are the responsibility of the caller to be freed.

Parameters:

- IN: comm_handle: handle returned by mpidbg_comm_query()
- OUT: comm_num_derived_files: filled with the length of the `comm_derived_files` array.
- OUT: comm_derived_files: filled with a pointer to an array of `MPI_File` handles derived from this communicator.
- OUT: comm_num_derived_windows: filled with the length of the `comm_derived_windows` array.
- OUT: comm_derived_windows: filled with a pointer to an array of `MPI_Win` handles derived from this communicator.

This function will return:

- MPIDBG_SUCCESS: if the handle is valid, was found, and the OUT parameters were filled in successfully.
- MPIDBG_ERR_NOT_FOUND: if the handle is not valid / found.
- MPIDBG_ERR_NOT_SUPPORTED: if this function is unsupported.

```
int mpidbg_comm_query_derived(struct mpidbg_comm_handle_t *handle,
                              int *comm_num_derived_files,
                              mqs_taddr_t **comm_derived_files,
                              int *comm_num_derived_windows,
                              mqs_taddr_t **comm_derived_windows);
```

mpidbg_errhandler_query()

These functions are analogous to the mpidbg_comm_* functions, but for `MPI_Errhandler`.

```
int mpidbg_errhandler_query(mqs_image *image,
                            mqs_image_info *image_info,
                            mqs_process *process,
                            mqs_process_info *process_info,
                            mqs_taddr_t errhandler,
                            struct mpidbg_errhandler_handle_t **handle);
```

mpidbg_errhandler_handle_free()

Free a handle returned by the mpidbg_errhandler_query() function.

Parameters:

- IN: handle: handle previously returned by mpidbg_errhandler_query()

This function will return:

- MPIDBG_SUCCESS: if the handle is valid and was freed successfully.
- MPIDBG_ERR_NOT_FOUND: if the handle is not valid / found.

```
int mpidbg_errhandler_handle_free(struct mpidbg_errhandler_handle_t *handle);
```

mpidbg_errhandler_query_basic()

Query a handle returned by mpidbg_errhandler_query() and, if found and valid, return basic information about the original errhandler.

Parameters:

- IN: handle: handle returned by mpidbg_comm_query()
- OUT: errhandler_name: string name of the communicator, max of MPIDBG_MAX_OBJECT_NAME characters, \0-terminated if less than MPIDBG_MAX_OBJECT_NAME characters.
- OUT: errhandler_bitflags: bit flags describing the communicator
- OUT: errhandler_fortran_handle: `INTEGER` Fortran handle corresponding to this communicator, or MPIDBG_ERR_UNAVAILABLE if currently unavailable, or MPIDBG_ERR_NOT_SUPPORTED if not supported
- OUT: errhandler_cxx_handle: Pointer to C++ handle corresponding to this communicator, or MPIDBG_ERR_UNAVAILABLE if currently unavailable, or MPIDBG_ERR_NOT_SUPPORTED if not supported
- OUT: errhandler_extra_info: Array of pointers to `mpidbg_keyvalue_pair_t` instances for extra info that this implementation wants to return to the tool about this errhandler. It is likely that the tool will not understand what these key=value pairs will mean, but at a minimum, they should be suitable for display.

`*errhandler_extra_info` can be NULL if there are no key/value pairs to return. If non-NULL, `*errhandler_extra_info` will point to an array of pointers to individual `mpigb_keyvalue_pair_t` instances. Note that the memory associated with `*errhandler_extra_info` is associated with the handle, and will be freed when mpidb_errhandler_handle_free() is invoked.

This function will return:

- MPIDBG_SUCCESS: if the handle is valid, was found, and the OUT parameters were filled in successfully.
- MPIDBG_ERR_NOT_FOUND: if the handle is not valid / found.
- MPIDBG_ERR_NOT_SUPPORTED: if this function is unsupported.

```
int mpidbg_errhandler_query_basic(
            struct mpidbg_errhandler_handle_t *handle,
            char errhandler_name[MPIDBG_MAX_OBJECT_NAME],
            enum mpidbg_errhandler_info_bitmap_t *errhandler_bitflags,
            int *errhandler_fortran_handle,
            mqs_taddr_t *errhandler_cxx_handle,
            struct mpidbg_keyvalue_pair_t ***errhandler_extra_info);
```

mpidbg_errhandler_query_handles()

Query a handle returned by mpidbg_errhandler_query() and, if found and valid, return an array of MPI handle that are using this error handler. The type of the MPI handle returned will be indicated by one of the following set in the errhandler bitmap flags:

- MPIDBG_ERRH_INFO_COMMUNICATOR
- MPIDBG_ERRH_INFO_FILE
- MPIDBG_ERRH_INFO_WINDOW

All arrays returned in OUT variables are allocated by this function, but are the responsibility of the caller to be freed.

Parameters:

- IN: errhandler_handle: handle returned by mpidbg_errhandler_query()
- OUT: errhandler_num_handles: filled with the length of the `errhandler_derived_files` array.
- OUT: erhandler_handles: filled with a pointer to an array of MPI handles that are using this errhandler.

This function will return:

- MPIDBG_SUCCESS: if the handle is valid, was found, and the OUT parameters were filled in successfully.
- MPIDBG_ERR_NOT_FOUND: if the handle is not valid / found.
- MPIDBG_ERR_NOT_SUPPORTED: if this function is unsupported.

```
int mpidbg_errhandler_query_handles(struct mpidbg_errhandler_handle_t *handle,
                                    int *errhandler_num_handles,
                                    mqs_taddr_t **errhandler_handles);
```

mpidbg_errhandler_query_callback()

Query a handle returned by mpidbg_errhandler_query() and, if found and valid, return the function pointer callback on this errhandler. The signature of the function pointer returned is determined by the combintion of errhandler bitmap flags:

- MPIDBG_ERRH_INFO_COMMUNICATOR, or MPIDBG_ERRH_INFO_FILE, or MPIDBG_ERRH_INFO_WINDOW
- MPIDBG_ERRH_INFO_C_CALLBACK, or MPIDBG_ERRH_INFO_FORTRAN_CALLBACK, or MPIDBG_ERRH_INFO_CXX_CALLBACK

Note that the output function pointer will be NULL if MPIDBG_ERRH_INFO_PREDEFINED is set on the errhandler bitmap flags.

Parameters:

- IN: errhandler_handle: handle returned by mpidbg_errhandler_query()
- OUT: errhandler_func_ptr: filled with the function pointer to the callback function.

This function will return:

- MPIDBG_SUCCESS: if the handle is valid, was found, and the OUT parameters were filled in successfully.
- MPIDBG_ERR_NOT_FOUND: if the handle is not valid / found.
- MPIDBG_ERR_NOT_SUPPORTED: if this function is unsupported.

```
int mpidbg_errhandler_query_callback(struct mpidbg_errhandler_handle_t *handle,
                                     mqs_taddr_t *errhandler_func_ptr);
```

mpidbg_request_query()

These functions are analogous to the mpidbg_comm_* functions, but for `MPI_Request`.

```
int mpidbg_request_query(mqs_image *image,
                         mqs_image_info *image_info,
                         mqs_process *process,
                         mqs_process_info *process_info,
                         mqs_taddr_t request,
                         struct mpidbg_request_handle_t **handle);
```

mpidbg_request_handle_free()

Free a handle returned by the mpidbg_request_query() function.

Parameters:

- IN: handle: handle previously returned by mpidbg_request_query()

This function will return:

- MPIDBG_SUCCESS: if the handle is valid and was freed successfully.
- MPIDBG_ERR_NOT_FOUND: if the handle is not valid / found.

```
int mpidbg_request_handle_free(struct mpidbg_request_handle_t *handle);
```

mpidbg_status_query()

These functions are analogous to the mpidbg_comm_* functions, but for `MPI_Status`.

```
int mpidbg_status_query(mqs_image *image,
                        mqs_image_info *image_info,
                        mqs_process *process,
                        mqs_process_info *process_info,
                        mqs_taddr_t status,
                        struct mpidbg_status_handle_t **handle);
```

mpidbg_status_handle_free()

Free a handle returned by the mpidbg_status_query() function.

Parameters:

- IN: handle: handle previously returned by mpidbg_status_query()

This function will return:

- MPIDBG_SUCCESS: if the handle is valid and was freed successfully.
- MPIDBG_ERR_NOT_FOUND: if the handle is not valid / found.

```
int mpidbg_status_handle_free(struct mpidbg_status_handle_t *handle);
```

Index