The MPI Message Queue Dumping Interface

Version 1.0

MPI Forum Working Group on Tools
Accepted by the Message Passing Interface Forum
(date tbd.)

Acknowledgments

Author
Anh Vo

Contributing Authors
John DelSignore, Kathryn Mohror, Jeff Squyres

Editor
TBD

Reviewers
Dong Ahn, William Gropp, Martin Schulz

Contents

Chapter 1

Background

In early 1995, TotalView's Jim Cownie and Argonne National Laboratory's Bill Gropp and Rusty Lusk developed parallel debugging interfaces for use with MPI. They designed and implemented the interfaces in MPICH, one of the first widely available MPI implementations. Two interfaces were developed: one for process discovery and acquisition and one for message queue inspection. Coined the "MPIR" interfaces [1, 2], the MPI debugging interfaces eventually became *de facto* standards implemented by various MPI providers such as Compaq, HP, IBM, Intel, LAM/MPI, MPI Software Technologies, Open MPI, Quadrics, SCALI, SGI, Sun/Oracle, and other implementations of MPI.

In 2010, the MPI Forum published a document which formally described the MPIR Process Acquisition Interface but omitted the details about the MPI Message Queue Dumping (MQD) interface. This document complements the MPIR Process Acquisition Interface document by describing the existing MQD interface being used by most MPI debuggers and MPI implementations today to provide users with information about the state of message queues in an MPI program.

> *Rationale.* Note that this document does *not* introduce any improvements to the existing *de facto* use of the MQD interface. Nor does it addresses any shortcomings of the existing MQD interface, such as the inability to load different debugger dynamically linked libraries (DLLs) to support an environment where the debugger runs with a different bitness from the target. This document is solely intended to codify the current state of the art. (*End of rationale.*)

Chapter 2

Overview

Tools and debuggers use the MQD interface to extract information describing the conceptual message-passing state of an MPI process. While the original intent of the interface was to provide the functionality to debuggers, any tool that has debugger-like capabilities (e.g., providing symbol name look up) can use this interface to access the message-passing state. Note that this document uses the terms "tools" and "debuggers" interchangeably.

Within each MPI process, there are three distinct abstract message queues which represent the MPI subsystem. They are:

1. Send Queue: This queue represents all of the outstanding send operations.
2. Receive Queue: This queue represents all of the outstanding receive operations.
3. Unexpected Message Queue: This queue represents all the messages that have arrived at the process, but have not been matched yet.

The send and receive queues store information about all of the unfinished send and receive operations that the process has started within a given communicator. These might result either from blocking operations such as MPI_SEND and MPI_RECV or nonblocking operations such as MPI_ISEND and MPI_IRECV. Each entry in these queues contains the information that was passed to the function call that initiated the operation. Nonblocking operations remain on these queues until they have been completed by MPI_WAIT, MPI_TEST, or one of the related multiple completion routines.

The unexpected message queue contains a different class of information than the send and receive queues, because the elements on this queue were created by MPI calls in other processes. Therefore, less information is available about these elements (e.g., the data type that was used by the sender).

In all three queues, the order of the elements represents the order that the MPI subsystem will perform matching. This is important where many entries could match, for instance when a wildcard tag or source is used in a receive operation.

Note that these queues are conceptual; they are an abstraction for representing the progression of messages in an MPI program. The actual number of queues in an MPI implementation is implementation dependent. The MQD interface defines these conceptual queues so that message information can be presented to users independently of any particular MPI implementation. For example, an MPI implementation may maintain only two queues, the receive queue and the unexpected message queue. The implementation does not maintain an explicit queue of send operations; instead, all the information about an incomplete send operation is maintained in the associated `MPI_Request`.

Chapter 3

Definitions

3.1 Shared Library and DLL

3.1.1 Shared Library

A *shared library* is a file that is intended to be shared by excutable files and other shared libraries. Shared libraries may be relocated at runtime, and may be dynamically loaded at runtime.

3.1.2 Shared Object File

The term *shared object file* is used on UNIX and UNIX-based systems to describe a shared library.

3.1.3 Dynamic-link Library

On Windows and OS/2 systems, a shared library is referred to as a *dynamic-link library.*

3.1.4 Dynamically Loaded Library

A *dynamically loaded library* is a shared library that can be loaded and unloaded at runtime on request by calling routines like `dlopen`, `dlclose`, `dlsym` on UNIX and UNIX-based systems, or `LoadLibrary`, `FreeLibrary`, `GetProcAddress` on Windows systems. Debuggers require the dynamic loading of the MQD shared library to provide MQD support.

3.1.5 DLL

DLL is an overloaded term that refers to either a dynamic-link library, dynamically loaded library, or shared library.

3.2 Process and Image

3.2.1 Image

An *image file* is an executable or shared library file, which may contain symbol definitions needed by the MQD interface.

3.2.2 MPI Process

An *MPI process*, or simply *process* in the scope of this document, is defined to be an operating system (OS) process, which consists of an *address space* and a collection of execution contexts (threads or lightweight processes). The MPI process is part of the MPI application as described in the MPI standard. While the standard does not require that an MPI process be an OS process, this is a requirement for most debuggers and this interface was designed with that assumption.

3.2.3 Address Space

An *address space* is a region of memory that consists of executable code and data, and is partially composed of a collection of image files. The collection of image files may change at any point during the execution of the MPI process, and the image files may be relocated at runtime within the address space at the point they are loaded into memory.

3.2.4 "mqs_image"

An *mqs_image*, or sometimes simply referred to as an image in this document, is an abstract concept that represents the collection of image files loaded into the address space of the MPI process at any given time, and is debugger implementation defined. In static execution environments, where shared libraries are not supported, an *mqs_image* can represent an executable image file. However, in dynamic execution environments, where shared libraries, dynamically loaded shared libraries, and runtime relocation of shared libraries are supported, an *mqs_image* represents the collection of image files loaded into the address space of the MPI process at any given point in time. In this situation, *mqs_image* may in fact represent the MPI process itself.

3.3 "Starter" Process

The *starter process* is the process that is responsible for launching the MPI job. The starter process may be a separate process that is not part of the MPI application, or any MPI process may act as a starter process. By definition, the starter process contains functions, data structures, and symbol table information for the MPIR Process Acquisition Interface.

The MPI implementation determines which launch discipline is used, as described in the following subsections.

3.3.1 The MPI Process as the Starter Process

An MPI implementation might implement its launching mechanism such that an MPI process, e.g., the MPI_COMM_WORLD rank 0 process, launches the remaining MPI processes of the MPI application. In such implementations, the MPI process that started the other MPI processes is the starter process.

3.3.2 A Separate mpiexec as the Starter Process

Many MPI implementations use a separate `mpiexec` process that is responsible for launching the MPI processes. In these implementations, the `mpiexec` process is the starter process. Note that the name of the starter process executable varies by implementation; `mpirun` is a

name commonly used by several implementations, for example. Other names include (but are not limited to) `srun`, `aprun`, `orterun`, and `prun`.

3.4 MQD Host and Target Node

For the purposes of this document, the *host node* is defined to be the node running the tool process, and a *target node* is defined to be a node running the target application processes the tool is controlling. A target node might also be the host node; that is, the target application processes might be running on the same node as the tool process.

Chapter 4

Debugger/MPI Interaction Model

4.1 The MQD DLL

The debugger gains access to the message queue functionality by loading a DLL provided by the MPI implementation, the ~~debug~~MQD DLL. This allows the debugger to be insulated from the internals of the MPI library so that it can support multiple MPI implementations. Furthermore, MPI implementations can provide their users with debugging support without requiring source access to the debugger. The debugger learns about the location of this DLL by reading the variable `MPIR_dll_name` from the MPI process.

All calls to the ~~debug~~MQD DLL from the debugger are made from entry points whose names are known to the debugger. However, all calls from the ~~debug~~MQD DLL to the debugger are made through a table of function pointers that is passed to the initialization entry point of the debug DLL. This procedure ensures that the ~~debug~~MQD DLL is independent of any particular debugger or debugger version.

For efficiency, it is important that the ~~debug~~MQD DLL be able to easily associate information with some of these debugger-owned objects. For instance, it is convenient to extract information about the address of a global variable of interest to the ~~debug~~MQD DLL only once for each process being debugged, rather than every time the ~~debug~~MQD DLL needs to access the variable. Similarly, the offset of a field in a structure that the ~~debug~~MQD DLL needs to lookup is constant within a specific executable image or shared library, and again should only be looked up once. Therefore, callbacks are provided by the debugger to allow the ~~debug~~MQD DLL to store and retrieve information associated with image and process objects. Since retrieving the information is a callback, the debugger has the option of either extending its internal data structures to provide space for an additional pointer or of implementing a lookup scheme (e.g., a hash table) to associate the information with the process key.

> *Advice to implementors.* Since the ~~debug~~MQD DLL will run within the code space of the debugger, the implementation of the ~~debug~~MQD DLL should avoid any calls that might block or sleep for a long period of time. Such call will make the debugger become unresponsive to user interaction. (*End of advice to implementors.*)

4.2 Debugger/Debug DLL Interaction Use Case

Figure 4.1 illustrates the interaction between the debugger and the ~~debug~~MQD DLL to iterate over the messages within the message queues. This example assumes that there are

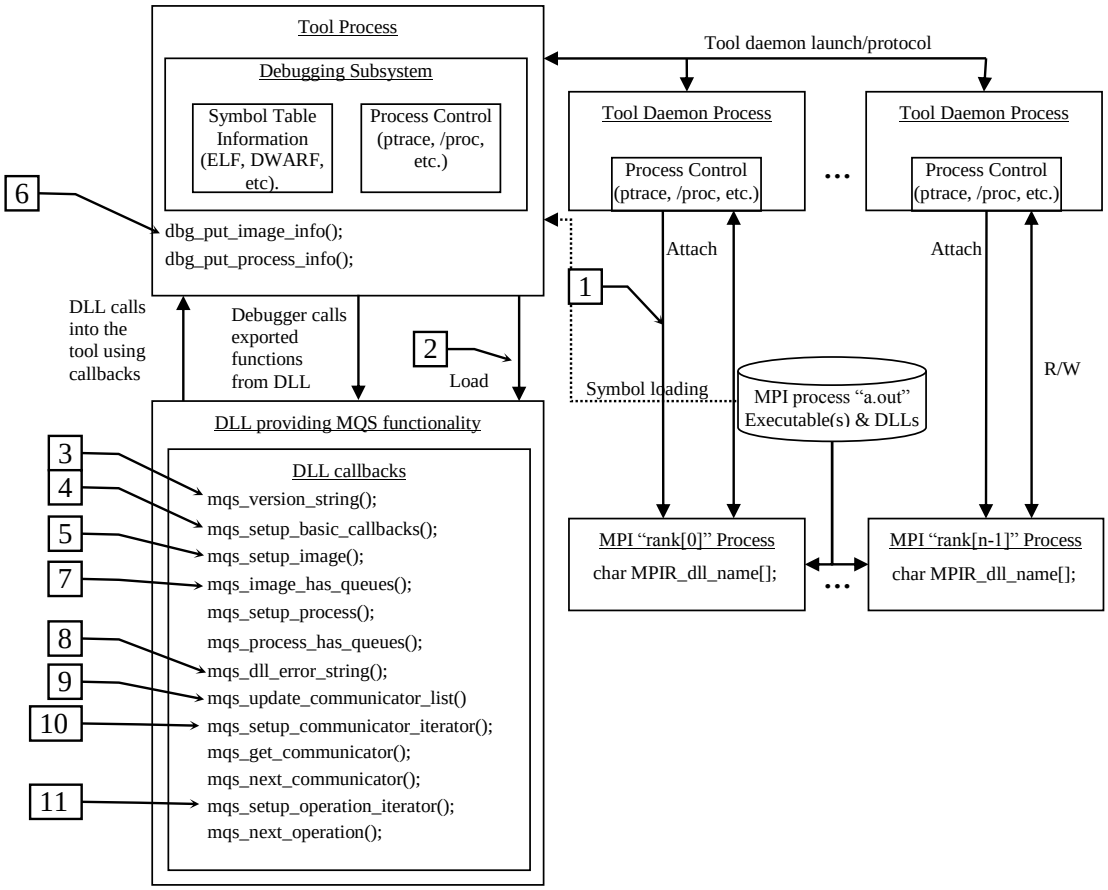

Figure 4.1: Example collaboration diagram for Debugger/DLL interaction

`n` MPI processes that were launched running the image "a.out".

1. The debugger looks for the global symbol `MPIR_dll_name` in the target process. If the symbol exists, it is expected to be a null-terminated string containing the name of the shared library (the shared object or DLL providing MQD functionality) to dynamically load into the debugger. If the symbol does not exist, the debugger might attempt to load a hardcoded shared library name. If no library exists, MQD functionality is disabled.

2. The debugger attempts to dynamically load the ~~debug~~MQD DLL.

3. Once the debugger has loaded the ~~debug~~MQD DLL it will check for version compatibility by calling `mqs_version_string()` to inquire the version of the ~~debug~~MQD DLL. It should also call `mqs_version_compatibility()` to inquire whether the ~~debug~~MQD DLL requires a different version of the debugger. Lastly, during this pre-initialization phase, the debugger should call `mqs_dll_taddr_width()` so that it knows the bit width with which the ~~debug~~MQD DLL was compiled.

4. The debugger initializes the ~~debug~~MQD DLL by calling `mqs_setup_basic_callbacks()` and passes the pointer to the structure containing the pointers to the basic callback functions provided by the debugger.

5. For each unique *mqs_image* that is used by the MPI processes the debugger calls `mqs_setup_image` and provides it with a pointer to callback structure containing image related callbacks.

6. The DLL will initialize any data structure necessary to store image specific information and will call `mqs_put_image_info` to have the debugger associate the *mqs_image* with the allocated data structure.

7. Once `mqs_setup_image` completes successfully, the debugger calls `mqs_image_has_queues` to indicate whether the *mqs_image* has MQD support or not. If the *mqs_image* has MQD support, the function will return `mqs_ok`, otherwise it will return an error. For each *mqs_image* that has queue support, the debugger should call `mqs_setup_process` on each process that is an instance of the *mqs_image* and subsequently call `mqs_setup_process_info` to allow the ~~debug~~MQD DLL to initialize any data structures that it uses to store process specific information. For each of the aforementioned processes, the debugger also calls `mqs_process_has_queues` to inquire whether the process has MQD support enabled.

8. If the ~~debug~~MQD DLL returns an error for any of the callbacks, the debugger should call `mqs_dll_error_string` to obtain more information about the error. On the other hand, if the debugger returns an error for any of the callbacks, the ~~debug~~MQD DLL should call `mqs_errorstring_fp` (part of the `mqs_basic_callbacks` structure) to get more information on the error.

9. Before querying the message queues, the debugger calls the function `mqs_update_communicator_list()` to verify that it has the latest information about the active communicators in a specific process and refreshes them if necessary.

10. The debugger then iterates over each communicator by first asking the ~~debug~~MQD DLL to setup the internal iterator to iterate over the active communicator list by calling `mqs_setup_communicator_iterator()`. Then it calls `mqs_get_communicator()` to obtain each communicator in the list and moves the iterator to the next communicator by calling `mqs_next_communicator()`. `mqs_next_communicator()` returns `mqs_ok` if there is another element to look at; otherwise it returns `mqs_end_of_list`.

11. Within each communicator, the debugger iterates over the message queues by first calling `mqs_setup_operation_iterator()` and indicates the queues it wants to iterate over. The debugger then calls `mqs_next_operation` to start iterating over the messages within the requested queue.

Chapter 5

Interface Specifications

The MPI Message Queue Dumping Interface is specified as a set of C-language definitions. The following sections enumerate those definitions. Unless otherwise noted, all definitions are required.

5.1 MPIR_dll_name

Global variable definition:

```
char MPIR_dll_name[]
```

`MPIR_dll_name` is a null-terminated string that contains the file system path name of the ~~debug~~MQD DLL provided by the MPI implementation. If defined, the symbol should be defined in the MPI process. If the symbol is not present in the MPI process, the debugger might attempt to load a default shared library, which is implementation dependent. If this also fails, MQD support is disabled.

> *Advice to implementors.* On some platforms it might be necessary to take additional efforts during compiling or linking to prevent this variable from being stripped or optimized out because it is usually not referenced from within the MPI implementation. (*End of advice to implementors.*)

5.2 Types for Target Independence

Since the code in the ~~debug~~MQD DLL is running inside the debugger, it could be running on a completely different platform than the target platform where the debugged process is running. For example, the debug DLL might be compiled as a 32-bit shared library, but the target MPI process might be compiled as a 64-bit application. Therefore, the interface uses explicit types to describe target types, rather than canonical C types.

5.2.1 mqs_tword_t

`mqs_tword_t` is a target independent `typedef` name that is the appropriate type for the ~~debug~~MQD DLL to use on the host to hold a target word (`long`). In other words, it is a signed integer wide enough to hold a target long.

5.2.2 mqs_taddr_t

`mqs_taddr_t` is a target independent `typedef` name that is the appropriate type for the ~~debug~~MQD DLL to use on the host to hold a target address (`void*`). In other words, it is an unsigned integer wide enough to hold a target address.

5.2.3 mqs_target_type_sizes

Type definition:

```
typedef struct
{
  int short_size;
  int int_size;
  int long_size;
  int long_long_size;
  int pointer_size;
} mqs_target_type_sizes;
```

`mqs_target_type_sizes` is a type definition for a `struct` that holds the size of common types in the target architecture. The ~~debug~~MQD DLL will use the callback `mqs_get_type_sizes_ft` provided by the debugger, which takes a variable of type `mqs_target_type_sizes`, and populates it with the size information that it has based on the target host:

- `short_size` holds the size of the type `short` in the target architecture.
- `int_size` holds the size of the type `int` in the target architecture.
- `long_size` holds the size of the type `long` in the target architecture.
- `long_long_size` holds the size of the type `long long` in the target architecture.
- `pointer_size` holds the size of a pointer (void*) in the target architecture.

5.3 Opaque Types Passed Through the Interface

The debugger exposes several objects to the ~~debug~~MQD DLL: an *mqs_image*, a specific process, and named types. In order to avoid exposing the debugger's internal representations of these types to the ~~debug~~MQD DLL, which has no need to see the internal structure of these objects, these objects are defined in the interface file as `typedef`s of undefined (opaque) structures and are always passed by reference (as a pointer to the opaque structure). The use of these opaque types allows the debugger the freedom to either pass true pointers to its internal data structures or to pass some other keys to the ~~debug~~MQD DLL from which it can later retrieve the internal objects associated with those keys. The use of `typedef`s provides more compile-time checking than the use of `void*` for objects.

The following are opaque types defined within the debugger and are exposed to the ~~debug~~MQD DLL as undefined `typedef`s. The ~~debug~~MQD DLL uses these types as keys to identify objects of interest, or to be passed back to the debugger through some callbacks.

- `mqs_image` identifies an *mqs_image*. That is, the object that describes the collection of image files loaded into the process' address space.
- `mqs_process` identifies an MPI process.
- `mqs_type` identifies a named target type symbol.

The following are opaque types defined within the debugger and the MQD DLL and are cast to and from explicit types within the ~~debug~~MQD DLL for the debug DLL's internal processing. These types exist so that the ~~debug~~MQD DLL can associate ~~some~~its own information with the debugger-owned objects.

- `mqs_image_info` is used to associate information pertaining to an object of type `mqs_image`.
- `mqs_process_info` is used to associate information pertaining to an object of type `mqs_process`.

5.4 Constants and Enums

5.4.1 mqs_lang_code

```
typedef enum {
  mqs_lang_c     = 'c',
  mqs_lang_cplus = 'C',
  mqs_lang_f77   = 'f',
  mqs_lang_f90   = 'F'
} mqs_lang_code;
```

Because symbol lookup mechanisms vary between different languages, it is necessary to indicate the language for which the lookup operation is intended. This enum is used to indicate the different language types.

5.4.2 mqs_op_class

```
typedef enum
{
  mqs_pending_sends,
  mqs_pending_receives,
  mqs_unexpected_messages
} mqs_op_class;
```

This enum is used by the debugger to indicate the queue of interests.

5.4.3 Interface compatibility enum

```
enum
{
  MQS_INTERFACE_COMPATIBILITY = 2
};
```

This constant defines the version of the interface header.

5.4.4 mqs_status

```
enum mqs_status
{
  mqs_st_pending,
  mqs_st_matched,
  mqs_st_complete
};
```

This enum is used to by the ~~debug~~MQD DLL to indicate the status of a message in the message queue.

5.4.5 Result code enums

```
enum {
  mqs_ok = 0,
  mqs_no_information,
  mqs_end_of_list,
  mqs_first_user_code = 100
};
```

This enum defines the various result codes for the message queue dumping functionality. This enum is used by both the ~~debug~~MQD DLL and the debugger.

5.4.6 Invalid MPI Process Rank enum

```
enum
{
  MQS_INVALID_PROCESS = -1
};
```

This constant provides a value indicating an invalid MP process rank.

5.5 Concrete Objects Passed Through the Interface

To allow the debugger to obtain useful information from the ~~debug~~MQD DLL, concrete types are defined to describe a communicator and a specific element in a message queue.

The information in the `mqs_communicator` structure includes the communicator's size, the local rank of the process within the communicator, and the name of the communicator as defined by the MPI implementation or set by the user using the MPI-2 function MPI_COMM_SET_NAME, which was added to the standard to aid in debugging and profiling.

The `mqs_pending_operation` structure contains enough information to allow the debugger to provide the user with details both of the arguments to a receive and of the incoming message that matched it. All references to other processes are available in the `mqs_pending_operation` structure both as indices into the group associated with the communicator and as indices into MPI_COMM_WORLD.

5.5.1 mqs_communicator

Type definition:

```
typedef struct
{
  mqs_taddr_t unique_id;
  mqs_tword_t local_rank;
  mqs_tword_t size;
  char        name[64];
} mqs_communicator;
```

The debugger uses this type definition to represent an MPI communicator.

- `unique_id` uniquely identifies a communicator.
- `local_rank` identifies the rank of the current MPI process in this communicator.
- `size` holds the size of the communicator.
- `name` contains the name of the communicator if it was given one.

5.5.2 mqs_pending_operation

Type definition:

```
typedef struct
{
  /* Fields for all messages */
  int         status;
  mqs_tword_t desired_local_rank;
  mqs_tword_t desired_global_rank;
  int         tag_wild;
  mqs_tword_t desired_tag;
  mqs_tword_t desired_length;
  int         system_buffer;
  mqs_taddr_t buffer;

  /* Fields valid if status >= matched or it is a send */
  mqs_tword_t actual_local_rank;
  mqs_tword_t actual_global_rank;
  mqs_tword_t actual_tag;
  mqs_tword_t actual_length;

  char extra_text[5][64];
} mqs_pending_operation;
```

This structure contains enough information to allow the debugger to provide the user with details about both of the arguments to a receive and of the incoming message that matched it. All references to other processes are available in the `mqs_pending_operation`

structure both as indices into the group associated with the communicator and as indices into MPI_COMM_WORLD. This avoids any need for the debugger to concern itself explicitly with this mapping

- `status` stores the status of the message. The value of this field should be either `mqs_st_pending`, `mqs_st_matched`, or `mqs_st_complete` as described in the enumeration `mqs_status` (section 5.4.4).
- `desired_local_rank` stores the rank of the target or the source for the communicator from which this message was initiated.
- `desired_global_rank` stores the rank of the target or the source with respect to MPI_COMM_WORLD.
- `tag_wild` is a boolean that identifies whether this message is a posted receive with tag `MPI_ANY_TAG`. If the receive was posted with `MPI_ANY_TAG`, `tag_wild` will be set to 1. Otherwise, it it set to 0.
- `desired_tag` holds the tag of the message. This field is ignored if `tag_wild` is not set.
- `desired_length` holds the length of the message buffer in bytes.
- `system_buffer` is a boolean that identifies whether this is a user or a system buffer. A value of 1 indicates that it is a system buffer. Otherwise, it is set to 0.
- `buffer` holds the address to the beginning of the message data.

The following fields are only meaningful if the message is a send or if the `status` fields indicates that this message is either matched (`mqs_st_matched`), or completed (`mqs_st_complete`).

- `actual_local_rank` holds the actual local rank within the communicator (after the message has matched).
- `actual_global_rank` holds the actual global rank with respect to MPI_COMM_WORLD.
- `actual_tag` holds the actual tag of the message.
- `actual_length` holds the actual length of the message buffer in bytes.
- `extra_text` is list of five ~~a~~ null-terminated ~~string~~strings that can be used by the ~~debug~~MQD DLL to provide more information to the user. The debugger does not interpret this field and can display it to the user. This field can be used to give the name of the function causing this request, for example.

5.6 Callbacks Provided by the Debugger

The debugger provides several callbacks that will be called by the ~~debug~~MQD DLL to extract information pertaining to the runtime state of the execution. All the callbacks are grouped into three different groups based on their functionalities: `mqs_basic_callbacks`, `mqs_image_callbacks`, and `mqs_process_callbacks`.

5.6.1 mqs_basic_callbacks

Type definition:

```
typedef struct mqs_basic_callbacks
{
  mqs_malloc_ft           mqs_malloc_fp;
  mqs_free_ft             mqs_free_fp;
  mqs_dprints_ft          mqs_dprints_fp;
  mqs_errorstring_ft      mqs_errorstring_fp;
  mqs_put_image_info_ft   mqs_put_image_info_fp;
  mqs_get_image_info_ft   mqs_get_image_info_fp;
  mqs_put_process_info_ft mqs_put_process_info_fp;
  mqs_get_process_info_ft mqs_get_process_info_fp;
} mqs_basic_callbacks;
```

This structure contains the pointers to the callbacks providing basic functionality.

5.6.2 mqs_malloc_ft

Function type definition:

typedef void* (*mqs_malloc_ft) (size_t size)

IN	size	number of bytes to allocate

Allocates a block of memory with the specified size. Note that because the debugger might operate within certain assumptions about memory allocation, the ~~debug~~MQD DLL cannot call any normal allocation routines (e.g., `malloc` or `strdup`); it has to use this function for memory allocation. The debugger guarantees that if the allocation fails, a NULL pointer will be returned. Memory allocated by `mqs_malloc_fp` must be deallocated using `mqs_free_fp`.

5.6.3 mqs_free_ft

Function type definition:

typedef void (*mqs_free_ft) (void* buf)

INOUT	buf	buffer to be freed

Frees a previously allocated memory buffer. Similarly to `mqs_malloc_fp`, the ~~debug~~MQD DLL has to use this function to free any memory allocated by `mqs_malloc_fp` (which is the only way to allocate memory from the ~~debug~~MQD DLL).

5.6.4 mqs_dprints_ft

Function type definition:

typedef void (*mqs_dprints_ft) (const char* buf)

INOUT	buf	buffer to be printed]

Prints a message to the debugger. This function is intended for debugging purposes only.

5.6.5 mqs_errorstring_ft

Function type definition:

typedef char* (*mqs_errorstring_ft) (int error_code)

IN	error_code	the error code for corresponding the error string

Converts an error code from the debugger into an error message. The function returns a pointer to a null terminated error string that corresponds to the given error code. The returned error string is owned by the debugger and must not be deallocated by the ~~debug~~MQD DLL.

5.6.6 mqs_put_image_info_ft

Function type definition:

typedef void (*mqs_put_image_info_ft) (mqs_image* image, mqs_image_info* imageinfo)

OUT	image	the *mqs_image* to receive the image info
IN	imageinfo	the image info to associate with the *mqs_image*

Associates the given image information with the given *mqs_image*. This allows the ~~debug~~MQD DLL to cache the information associated with the *mqs_image* (e.g., the pointer to the callbacks structure provided by the debugger) so that it can retrieve it later (using `mqs_get_image_info_fp`) when the debugger needs to invoke image-related functionalities (e.g., `mqs_image_has_queues` – see section 5.8.2). See section 5.8.1 for more details.

5.6.7 mqs_get_image_info_ft

Function type definition:

typedef mqs_image_info* (*mqs_get_image_info_ft) (mqs_image* image)

IN	image	the *mqs_image* to extract the image info from

Returns the image information associated with the given *mqs_image*.

5.6.8 mqs_put_process_info_ft

Function type definition:

typedef void (*mqs_put_process_info_ft) (mqs_process* process, mqs_process_info* processinfo)

OUT	process	the process to receive the process info
IN	processinfo	the process info to associate with the process

Associates the given process information with the given process. This allows the ~~debug~~MQD DLL to cache the information associated with the process (e.g., the pointer to the callbacks structure provided by the debugger) so that it can retrieve it later (using `mqs_get_process_info_fp`) when the debugger needs to invoke process-related functionalities (e.g., `mqs_process_has_queues` – see section 5.9.2). See section 5.8.1 for more details.

5.6.9 mqs_get_process_info_ft

Function type definition:

typedef mqs_process_info* (*mqs_get_process_info_ft) (mqs_process* process)

IN	process	the process to extract the process info from

Returns the process information associated with the given process.

5.6.10 mqs_image_callbacks

Type definition:

```
typedef struct mqs_image_callbacks
{
  mqs_get_type_sizes_ft   mqs_get_type_sizes_fp;
  mqs_find_function_ft    mqs_find_function_fp;
  mqs_find_symbol_ft      mqs_find_symbol_fp;
  mqs_find_type_ft        mqs_find_type_fp;
  mqs_field_offset_ft     mqs_field_offset_fp;
  mqs_sizeof_ft           mqs_sizeof_fp;
} mqs_image_callbacks;
```

This structure contains the pointers to the callbacks providing *mqs_image* related functionality.

5.6.11 mqs_get_type_sizes_ft

Function type definition:

typedef void (*mqs_get_type_sizes_ft) (mqs_process* process, mqs_target_type_sizes* sizes)

IN	process	the process to get the sizes from
OUT	sizes	the pointer to the structure to receive the sizes

Retrieves the size information about common data types for process and stores them in the structure pointed to by `sizes`. See section 5.2.3 for the definition of `mqs_target_type_sizes`.

5.6.12 mqs_find_function_ft

Function type definition:

typedef int (*mqs_find_function_ft) (mqs_image* image, char* fname, mqs_lang_code lang, mqs_taddr_t* addr)

IN	image	the *mqs_image* to search for the function
IN	fname	the name of the function to search for
IN	lang	the language code
OUT	addr	the address of the function

Given an *mqs_image*, returns the address of the specified function. The function returns `msq_ok` if successful and `mqs_no_information` if the function cannot be found.

Advice to implementors. Some implementations of the ~~debug~~MQD DLL might use this function to force the debugger to fully process all symbol table information in a compilation unit. The ~~debug~~MQD DLL chooses a function defined in a compilation unit that contains MPI type definitions to make sure that the debugger has fully read in the types. (*End of advice to implementors.*)

5.6.13 mqs_find_symbol_ft

Function type definition:

typedef int (*mqs_find_symbol_ft) (mqs_image* image, char* sname, mqs_taddr_t* addr)

IN	image	the *mqs_image* to search for the symbol
IN	sname	the name of the symbol to search for
OUT	addr	the address of the symbol

Given an *mqs_image*, returns the address of the specified symbol. The function returns `msq_ok` if successful and `mqs_no_information` if the symbol cannot be found.

5.6.14 mqs_find_type_ft

Function type definition:

typedef mqs_type* (*mqs_find_type_ft) (mqs_image* image, char* tname, mqs_lang_code lang)

IN	image	the *mqs_image* to search for the type
IN	tname	the name of the type to search for
IN	lang	the language code

Given an *mqs_image*, returns the type associated with the given named type. The function either returns a type handle (a pointer to an opaque `mqs_type` handle), or NULL if the type cannot be found.

> *Advice to implementors.* Since the debugger may load debug information lazily and/or the linker may remove such type information during optimization, the MPI run time library should ensure that the type definitions required occur in a file whose debug information will already have been loaded. See the advice for implementors for `mqs_find_function_fp` for an example of how to use `mqs_find_function_fp` to force the debugger to load debug information. (*End of advice to implementors.*)

5.6.15 mqs_field_offset_ft

Function type definition:

typedef int (*mqs_field_offset_ft) (mqs_type* type, char* fname)

IN	type	the type that contains the field
IN	fname	the field name to retrieve the offset

Given the type handle for a `struct` type, returns the byte offset of the named field. If the field cannot be found, the function returns -1.

5.6.16 mqs_sizeof_ft

Function type definition:

typedef int (*mqs_sizeof_ft) (mqs_type* type)

IN	type	the type to get the size for

Given the type handle for a type, returns the size of the type in bytes.

5.6.17 mqs_process_callbacks

Type definition:

```
typedef struct mqs_process_callbacks
{
  mqs_get_global_rank_ft       mqs_get_global_rank_fp;
  mqs_get_image_ft             mqs_get_image_fp;
  mqs_fetch_data_ft            mqs_fetch_data_fp;
  mqs_target_to_host_ft        mqs_target_to_host_fp;
} mqs_process_callbacks;
```

This structure contains the pointers to the callbacks providing process related functionality.

5.6.18 mqs_get_global_rank_ft

Function type definition:

typedef int (*mqs_get_global_rank_ft) (mqs_process* process)

IN	process	the process to get the global rank for

Given a process, returns its rank in MPI_COMM_WORLD. Returns MQS_INVALID_PROCESS if the rank of the process is not known.

5.6.19 mqs_get_image_ft

Function type definition:

typedef mqs_image* (*mqs_get_image_ft) (mqs_process* process)

IN	process	the process to get the *mqs_image* for

Given a process, returns a pointer to the *mqs_image* (i.e., the object describing the set of image files loaded into the process' address space).

5.6.20 mqs_fetch_data_ft

Function type definition:

typedef int (*mqs_fetch_data_ft) (mqs_process* process, mqs_taddr_t addr, int size, void* buf)

IN	process	the process to fetch the data from
IN	addr	the virtual address in the process' virtual address space
IN	size	the number of bytes to read
OUT	buf	the buffer to store the data

Fetches data from the process into the specified buffer. The function returns `msq_ok` if the data could be fetched successfully. Otherwise, it returns `mqs_no_information`. The

data returned in the buffer is in the same format as data stored in the target process when accessed as a byte array. The ~~debug~~MQD DLL must call `mqs_target_to_host_fp` to do any necessary byte reordering for multi-byte types, such as short, int, void*, double, etc.

5.6.21 mqs_target_to_host_ft

It is possible that although the debugger is running locally on the same machine as the target process, the target process may have different properties from the debugger. For example, on some operating systems it is possible to execute both 32- and 64-bit processes. To handle this situation, the debugger provides a callback that returns type size information for a specific process. To handle the possibility that the byte ordering may be different between the debug host and the target, the debugger provides a callback to perform any necessary byte reordering when viewing the target store as an object of a specific size. This callback enables the ~~debug~~MQD DLL to be entirely independent of the target process.

Function type definition:

typedef void (*mqs_target_to_host_ft) (mqs_process* process, const void* indata, void* outdata, int size)

IN	process	the process where the original data is from
IN	indata	the data to convert
OUT	outdata	the buffer to store the converted data
IN	size	the number of bytes to convert

Converts data from target representation to host representation.

5.7 Basic Callbacks Provided by the MQD DLL

This section contains the declarations for basic functions that are defined by the MQD DLL, and called by the debugger. These functions have C-linkage and must be exported by the MQD DLL.

5.7.1 mqs_setup_basic_callbacks

Function declaration:

extern void mqs_setup_basic_callbacks(const mqs_basic_callbacks* cb)

IN	cb	the basic callbacks table to provide to the ~~debug~~MQD DLL

This function is called by the debugger to the ~~debug~~MQD DLL to provide the ~~debug~~MQD DLL with the basic callbacks table. The ~~debug~~MQD DLL needs only save the pointer to the `mqs_basic_callbacks` object. The debugger must ensure the structure of ~~fuention~~function pointers remain valid for as long as the ~~debug~~MQD DLL is in use. The ~~struture~~structure is owned by the debugger, and should not be modified or deallocated by the ~~debug~~MQD DLL. This rule applies to all of the callback structures.

5.7.2 mqs_version_string

Function declaration:

```
extern char* mqs_version_string()
```

Returns the ~~debug~~MQD DLL version. The debugger can print or display the version string so that the user know which ~~debug~~MQD DLL was loaded. The returned version string is owned by the ~~debug~~MQD DLL and must not be deallocated by the debugger.

5.7.3 mqs_version_compatibility

Function declaration:

```
extern int mqs_version_compatibility()
```

Returns the ~~debug~~MQD DLL compatibility level (i.e., the value of MQS_INTERFACE_-COMPATIBILITY when the ~~debug~~MQD DLL was compiled). This allows the debugger to check whether this version of the ~~debug~~MQD DLL's MQD support is compatible with the debugger's version (e.g., whether the user needs a newer version of the debugger).

5.7.4 mqs_dll_taddr_width

Function declaration:

```
extern int mqs_dll_taddr_width()
```

Returns the size of `mqs_taddr_t` (i.e., *sizeof*(`mqs_taddr_t`)) that has been compiled into the ~~debug~~MQD DLL. It is not the width of an address or pointer for the target process, which could be a different size from an `mqs_taddr_t`. This function is useful, for example, when the ~~debug~~MQD DLL was compiled with a 32-bit `mqs_addr_t` type, but the debugger was compiled with a 64-bit `mqs_addr_t`.

5.7.5 mqs_dll_error_string

Function declaration:

```
extern char* mqs_dll_error_string(int error_code)
```

IN	error_code	the error code that corresponds to the error string

Returns a pointer to a null-terminated string that is associated with the error code. This function provides a means for the debugger to get the string associated with an error returned from the ~~debug~~MQD DLL. The returned error string is owned by the ~~debug~~MQD DLL and must not be deallocated by the debugger. Note that this function complements the function `mqs_errorstring_fp`, which provides a means for the ~~debug~~MQD DLL to get the string associated with an error code returned from the debugger.

5.8 Executable Image Related Functions

~~These functions are provided by the debug DLL and are called by the debugger.~~This section contains the declarations for executable image related functions that are defined by the MQD DLL, and called by the debugger. These functions have C-linkage and must be exported by the MQD DLL.

5.8.1 mqs_setup_image

Function declaration:

```
extern int mqs_setup_image(mqs_image* image, const mqs_image_callbacks* cb)
```

OUT	image	the *mqs_image* to setup the callbacks table
IN	cb	the image callbacks table

Sets up debug information for an *mqs_image*. This function must cache the provided callbacks and use those functions for accessing this image. The DLL should use the `mqs_put_image_info_fp` and `mqs_get_image_info_fp` functions to associate information to keep with the *mqs_image*. The debugger will call `mqs_destroy_image_info` when the information about the given image is no longer needed. This will be called once for each ~~address space~~unique *mqs_image* in the parallel program.

5.8.2 mqs_image_has_queues

Function declaration:

```
extern int mqs_image_has_queues(mqs_image* image, char** message)
```

IN	image	the *mqs_image* to query MQD functionality
OUT	message	buffer to store message from the ~~debug~~MQD DLL

Returns whether this *mqs_image* has the necessary symbols to allow access to the message queues. This function is called by the debugger once for each unique *mqs_image* in the MPI ~~process~~processes, and the information is cached within the debugger. The function returns `mqs_ok` if MQD support can be provided by this image.

If message is set to non-NULL, it must be a printf-style string returned by the ~~debug~~MQD DLL to provide additional information about the result of the call. A non-NULL string may be returned on success or failure. The string must contain at most one printf-style "%s", with which the image name will be substituted, and no other printf-style tokens. For example, the debugger can print the message string using `printf(message,image_name);`. The returned message string, if any, is owned by the ~~debug~~MQD DLL and must not be deallocated by the debugger.

5.8.3 mqs_destroy_image_info

Function declaration:

extern int mqs_destroy_image_info(mqs_image_info* imageinfo)

IN	imageinfo	the image info to free

Allows the ~~debug~~MQD DLL to clean up when the image information is no longer needed by the debugger.

5.9 Process Related Functions

~~These functions are provided by the debug DLL and are called by the debugger.~~This section contains the declarations for process related functions that are defined by the MQD DLL, and called by the debugger. These functions have C-linkage and must be exported by the MQD DLL.

5.9.1 mqs_setup_process

Function declaration:

extern int mqs_setup_process(mqs_process* process, const mqs_process_callbacks* cb)

OUT	process	the process to setup the callbacks table
IN	cb	the processcallbacks table

Sets up process specific information.

5.9.2 mqs_process_has_queues

Function declaration:

extern int mqs_process_has_queues(mqs_process* process, char** message)

IN	image	the process to query mqs functionality
OUT	message	buffer to store message from the ~~debug~~MQD DLL

Similar to the `mqs_process_has_queues` function, this allows for querying whether the process has MQD support. This function should only be called if the *mqs_image* associated with the processclaims to provide MQD support. For example, the *mqs_image* might have enabled message queues support if only certain environment variables are set at launch time. This function checks at runtime whether MQD support is enabled for this specific process.

If message is set to non-NULL, it must be a printf-style string returned by the ~~debug~~MQD DLL to provide additional information about the result of the call. A non-NULL string may be returned on success or failure. The string must contain at most one printf-style "%s", with which the process name will be substituted, and no other printf-style tokens. For example, the debugger can print the message string using `printf(message,process_name);`. The returned message string, if any, is owned by the ~~debug~~MQD DLL and must not be deallocated by the debugger.

5.9.3 mqs_destroy_process_info

Function declaration:

```
extern int mqs_destroy_process_info(mqs_process_info* processinfo)
```

IN	processinfo	the process info to free

Allows the ~~debug~~MQD DLL to clean up when the process information is no longer needed by the debugger.

5.10 Query Functions

This section contains the declarations for query functions that are defined by the MQD DLL, and called by the debugger. These functions have C-linkage and must be exported by the MQD DLL.

These functions provide the message queue query functionality. The model is that the debugger calls down to the ~~debug~~MQD DLL to initialize an iteration over a specific class of items, and then keeps calling the iterating function until `mqs_false` is returned. The DLL internally keeps track of the item being iterated (similar to a file cursor). For communicators, the stepping is separated from extracting information because the debugger will need the state of the communicator iterator to qualify the selections of the operation iterator. `mqs_true` is returned when the description has been updated; `mqs_false` means there is no more information to return, and therefore the description contains no useful information. There is only one of each type of iteration running at once; the ~~debug~~MQD DLL should save the iteration state in the `mqs_process_info`.

5.10.1 mqs_update_communicator_list

Function declaration:

```
extern int mqs_update_communicator_list(mqs_process* process)
```

IN	process	the process to refresh the list of active communicators

Given a process, refreshes the list of active communicators. The function returns `msq_ok` if the operation succeeds.

5.10.2 mqs_setup_communicator_iterator

Function declaration:

```
extern int mqs_setup_communicator_iterator(mqs_process* process)
```

IN	process	the process to prepare the iterator

Given a process, prepares the iterator to iterate over the active communicator list. The function returns `msq_ok` if the operation succeeds.

5.10.3 mqs_get_communicator

Function declaration:

```
extern int mqs_get_communicator(mqs_process* process, mqs_communicator* mqs_comm)
```

IN	process	the process to retrieve the communicator
OUT	mqs_comm	the buffer to receive the communicator data

Extracts information about the current communicator. The function returns `msq_ok` if the operation succeeds. See section 5.5.1 for the definition of `mqs_communicator`.

5.10.4 mqs_get_comm_group

Function declaration:

```
extern int mqs_get_comm_group(mqs_process* process, int* ranks)
```

IN	process	the process to obtain the group information
OUT	ranks	the buffer to receive the rank mapping

Extracts the group from the current communicator. The debugger already knows the size of the communicator, so it will allocate a suitably sized array for the result. The result is an array in which the element at index `i` contains the MPI_COMM_WORLD rank of the `i-th` rank in the current communicator. The function returns `msq_ok` if the operation succeeds.

5.10.5 mqs_next_communicator

Function declaration:

```
extern int mqs_next_communicator(mqs_process* process)
```

IN	process	the process to get the next communicator

Moves the internal iterator to the next communicator in this process. The function returns `mqs_ok` if the operation succeeds. It returns `mqs_false` if there are no more communicators left in the iteration.

Example 5.1

```
/* Iterate over each communicator displaying the messages */

mqs_communicator comm;

for (dll->setup_communicator_iterator (process);
     dll->get_communicator (process, &comm) == mqs_ok;
     dll->next_communicator(process)){
   /* Do something on each communicator, described by comm */
}
```

5.10.6 mqs_setup_operation_iterator

Function declaration:

```
extern int mqs_setup_operation_iterator(mqs_process* process, int opclass)
```

IN	process	the process to setup the operation
IN	opclass	the type of operation requested

Prepares to iterate over the pending operations in the currently active communicator in this process. `op` should be one of the operations specified in `mqs_op_class` (see section 5.4.2 for the definition of `mqs_op_class`). The function returns `msq_ok` if the operation succeeds.

5.10.7 mqs_next_operation

Function declaration:

```
extern int mqs_next_operation(mqs_process* process, mqs_pending_operation* op)
```

IN	process	the process to fetch the operation
OUT	op	the buffer to receive the operation data

Fetches information about the next appropriate pending operation in the current communicator. The function returns `mqs_false` when there are no more operations in the iteration.

Bibliography

[1] mpi-debug: Finding Processes. http://www-unix.mcs.anl.gov/mpi/mpi-debug/.

[2] James Cownie and William Gropp. A Standard Interface for Debugger Access to Message Queue Information in MPI. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 51–58, Barcelona, Spain, September 1999.