Tools and Debugging Interfaces to MPI

Version 1.0

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

Acknowledgments

Author
John Hancock

Contributing Authors
Furby F. Furby, Godzilla, and Katniss

Editor
Octocat

Reviewers
Murphy

Contents

Chapter 1

Introduction

A wonderful introduction will be here.

Put a citation here so that bibtex is happy: [**?**]

Chapter 2

Background

Need some background here.

Chapter 3

Overview

The message queue interface is used by tools and debuggers to extract information describing the conceptual message-passing state of the MPI application so that this can be displayed to the users. While the original intent of the interface was to provide the functionality to debuggers, any tools that has debugger-like capability (e.g., providing symbol name lookup) can use use this interface to have access to the message passing state. From this point on, the document might use tools and debuggers interchangably.

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

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

The send and receive queues store information about all of the unfinished send and receive operations that the process has started within the commnunicator. These might result either from blocking operations such as `MPI_Send` and `MPI_Recv` or nonblocking operations such as `MPI_Isend` or `MPI_Irecv`. Each entry on one of these queues contains the information that was passed to the function call that initiated the operation. Nonblocking operations will remain on these queues until they have completed and have been collected by a suitable `MPI_Wait`, `MPI_Test`, or one of the related multiple completion routines. The unepxected message queue represents a different class of information, since the elements on this queue have been created by MPI calls in other processes. Therefore, less information is available about these elements (e.g., the datatype that was used by the sender). In all cases the order of the queues represents the order that the MPI subsystem will perform matching (this is important where many entries could match, for instance when wild-card tag or source is used in a receive operation).

Note that these queues are conceptual: they are a description of how a user can think about the progression of messages through an MPI program. The number of actual *queues* is implementation dependent. The interface described here addresses how to extract these conceptual queues from the imlementation so that they can be presented to the user independently of the particular MPI implementation. For example, an MPI implementation may maintain only two queues, the Receive Queue and the Unexpected Message Queue. There is no explicit queue of send operations; instead all of the information about an incomplete send operation is maintained in the associated `MPI_Request`.

Chapter 4

Definitions

4.1 MPI Process Definition

An MPI process is defined to be a process that is part of the MPI application as described in the MPI standard.

In this document, the rank of a process is assumed to be relative to MPI_COMM_WORLD (recall that this version of the MPIR interface does not support MPI-2 dynamic processes). For example, the phrase "MPI rank 0 process" denotes the process that is rank 0 in MPI_COMM_WORLD.

4.2 "Starter" Process Definition

The starter process is the process that is primarily responsible for launching the MPI job. The starter process may be a separate process that is not part of the MPI application, or the MPI rank 0 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.

The MPI Rank 0 Process as the Starter Process

An MPI implementation might also implement its launching mechanism such that the MPI rank 0 process launches the remaining MPI processes of the MPI application. In such implementation, the MPI rank 0 process is the starter process.

A Separate mpiexec as the Starter Process

Most 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` and `prun`.

4.3 MPIR Node Definitions

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 be the host node, that is, the target application processes might be running on the same node as the tool process.

Chapter 5

Debugger/MPI Interaction Model

The debugger will have access to the message queue functionality by loading a shared library provided by the MPI implementation. 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 shared library by reading variable `MPIR_dll_name` from the MPI Starter Process.

All calls to the debug DLL from the debugger are made from entry points whose names are known to the debugger. However, all calls back to the debugger from the debug DLL are made through a table of function pointers that is passed to the intialization entrypoint of the debug DLL. This procedure ensures that the debug DLL is independent of the specific debugger from which it is being called.

Chapter 6

Interface Specifications

Unless otherwise noted, all definitions are required and shall be provided in the interface header file.

6.1 MPIR_dll_name

Global variable definition:

```
char* MPIR_dll_name
```

> Definition is required.
> Definition is contained within the address space of the starter process.
> Variable is written by the starter process, and read by the tool.

`MPIR_dll_name` contains the location of debugger DLL provided by the MPI implementation.

6.2 mqs_tword_t

`mqs_tword_t` is a target independence typedef name that is the appropriate type for the DLL to use on the host to hold a target word (`long`).

6.3 mqs_taddr_t

`mqs_tword_t` is a target independence typedef name that is the appropriate type for the DLL to use on the host to hold a target address (`void*`)

6.4 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 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 populate 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 in the target architecture

6.5 Opaque Types Passed Through the Interface

The following three types are opaque type that are defined within the debugger and are exposed to the debug DLL as undefined typedef's. The debug DLL has no need to see the internal structure of this type, but merely uses them as keys to identify objects of interest, or to be passed back to the debugger through some callback.

1. mqs_image identifies an executable image.
2. mqs_process identifies an MPI process.
3. mqs_type identifies a named target type.

The following two types are opaque types defined within the debugger and are cast to concrete types within the debug DLL for the debug DLL's internal processing. These types exist so that the debug DLL can associate some information with the debugger owned objects.

1. mqs_image_info is used to associate information pertaining to an object of type mqs_image.
2. mqs_process_info is used to associate information pertaining to an object of type mqs_process.

6.6 Constants and Enums

6.6.1 mqs_lang_code

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

This enum is used by both the debug DLL and the debuger to deal with the different language type that the original target code was based on.

6.6.2 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 which queue it is interested in.

6.6.3 mqs_interface_version

This constant defines the version of the interface header

6.6.4 mqs_status

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

This enum is used to indicate the status of a message in the message queue.

6.6.5 Other enums

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

This enum defines the various result code for the message queue functionality

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

This constant provides a value for the debugger to return error indicating an invalid process index.

6.7 Concrete Objects Passed Through the Interface

6.7.1 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;
```

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

6.7.2 mqs_pending_operation

Type defintion:

```
typedef struct
{
  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 refereces 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`.

`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 identifies whether this message is a posted receive with tag being MPI_ANY_-TAG

desired_tag holds the tag of the message. This field is only meaningful if tag_wild is not set.

desired_length holds the length of the message buffer.

system_buffer identifies whether this is a user or a system buffer.

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 (after the message has matched).

actual_global_rank holds the actual local rank with respect to MPI_COMM_WORLD.

actual_tag holds the actual tag.

actual_length holds the actual length.

extra_text can be used by the DLL to provide more information to the user. The debugger does not interpret this field and simply displays it to the user.

6.8 Callbacks Provided by the Debugger

The debugger provides several callbacks that will be called by the 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.

6.8.1 mqs_basic_callbacks

Type definition:

```
typedef struct mqs_basic_callbacks
{
  mqs_malloc_ft            mqs_malloc_fp;
  mqs_free_ft              mqs_free_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;
```

mqs_malloc_ft

Function type definition:

```
typedef void* (*mqs_malloc_ft) (size_t);
```

Allocates a block of memory. The size (in bytes) of this block shall be given as an argument to the function.

mqs_free_ft

Function type definition:

```
typedef void (*mqs_free_ft) (void*);
```

Frees a previously allocated memory. The pointer to the beginning of the allocated block of memory shall be given as an argument to the function.

mqs_errorstring_ft

Function type definition:

```
typedef char* (*mqs_errorstring_ft) (int);
```

Converts an error code fromt the debugger into an error message. The error code is given as an argument to the function.

mqs_put_image_info_ft

Function type definition:

```
typedef void (*mqs_put_image_info_ft) (mqs_image*, mqs_image_info*);
```

Associates the given image information with the given image. The image as well as the image information, in the order mentioned, shall be given as arguments to the function.

mqs_get_image_info_ft

Function type definition:

```
typedef mqs_image_info* (*mqs_get_image_info_ft) (mqs_image*);
```

Returns the image information associated with the given image. The image shall be given as the argument to the function.

mqs_put_process_info_ft

Function type definition:

```
typedef void (*mqs_put_process_info_ft) (mqs_process*, mqs_process_info*);
```

Associates the given process information with the given process. The process as well as the process information, in the order mentioned, shall be given as arguments to the function.

mqs_get_process_info_ft

Function type definition:

```
typedef mqs_process_info* (*mqs_get_process_info_ft) (mqs_process*);
```

Returns the process information associated with the given process. The process shall be given as the argument to the function.

6.8.2 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;
```

mqs_get_type_sizes_ft

Function type definition:

```
typedef void (*mqs_get_type_sizes_ft) (mqs_process*, mqs_target_type_sizes*);
```

Populates the size information about common datatypes from the running process. The process and the struct to hold the sizes, in the order mentioned, shall be given as arguments to the function.

mqs_find_function_ft

Function type definition:

```
typedef int (*mqs_find_function_ft) (mqs_image*, char*, mqs_lang_code, mqs_taddr_t* );
```

Given an image, returns the address of the specified function. The image, the function name, the language, and the output buffer to store the address of the function, in the order mentioned, shall be given as arguments to the function. The function returns `msq_ok` if successful and `mqs_no_information` otherwise.

mqs_find_symbol_ft

Function type definition:

```
typedef int (*mqs_find_symbol_ft) (mqs_image*, char*, mqs_taddr_t* );
```

Given an image, returns the address of the specified symbol. The image, the function name, and the output buffer to store the address of the symbol, in the order mentioned, shall be given as arguments to the function. The function returns `msq_ok` if successful and `mqs_no_information` otherwise.

mqs_find_type_ft

Function type definition:

```
typedef mqs_type* (*mqs_find_type_ft) (mqs_image*, char*, mqs_lang_code);
```

Given an image, returns the type associated with the given named type. The image, the name of the type, and the language, in the order mentioned, shall be given as arguments to the function, in the order mentioned. The function either returns a type handle, or NULL if the type cannot be found.

> *Advice to implementors.* Since the debugger may load debug information lazily, the MPI run time library should ensure that the type definitions required occur in a file whose debug information will already ahve been loaded. For instance, by placing them in the same file as the startup breakpoint function. (*End of advice to implementors.*)

mqs_field_offset_ft

Function type definition:

```
typedef int (*mqs_field_offset_ft) (mqs_type*, char*);
```

Given the handle for a struct type, returns the byte offset of the named field. The handle of the type and the name of the field, in the order mentioned, shall be given as arguments to the function. If the field cannot be found, the function returns -1.

mqs_sizeof_ft

Function type definition:

```
typedef int (*mqs_sizeof_ft) (mqs_type*);
```

Given the handle for a type, return the size of the type in bytes. The handle to the type shall be given as an argument to the function.

6.8.3 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;
```

mqs_get_global_rank_ft

Function type definition:

```
typedef int (*mqs_get_global_rank_ft) (mqs_process*);
```

Given a process, returns its rank in `MPI_COMM_WORLD`. The process shall be given as the argument to the function.

mqs_get_image_ft

Function type definition:

```
typedef mqs_image* (*mqs_get_image_ft) (mqs_process*);
```

Given a process, returns the image of which it is an instance. The process shall be given as the argument to the function.

mqs_fetch_data_ft

Function type definition:

```
typedef int (*mqs_fetch_data_ft) (mqs_process*, mqs_taddr_t, int, void*);
```

Fetches data from the process into the specified buffer. The process handle, the address of the desired data, the number of bytes to read, and the specified buffer, in the order mentioned, shall be given as arguments to the function. The function returns `msq_ok` if the data could be fetched successfully. Otherwise, it returns `mqs_no_information`.

mqs_target_to_host_ft

Function type definition

```
typedef void (*mqs_target_to_host_ft) (mqs_process*, const void*, void*, int);
```

Converts data from target representation to host representation. The process, the original data, the buffer to store the converted data, and the number of bytes to convert, in the order mentioned, shall be given as argument to the function.

6.9 Callbacks Provided by the DLL

6.9.1 mqs_setup_basic_callbacks

Function type definition:

```
extern void mqs_setup_basic_callbacks( const mqs_basic_callbacks* );
```

This function is called by the debugger to populate the basic callbacks table for the DLL.

6.9.2 mqs_version_string

Function type definition:

```
extern char* mqs_version_string( void );
```

Returns the DLL version.

6.9.3 mqs_version_compatibility

Function type definition:

```
extern int mqs_version_compatibility( void );
```

Returns the DLL compatibility level.

6.9.4 mqs_dll_taddr_width

Function type definition:

```
extern int mqs_dll_taddr_width( void );
```

Gives the width of an address pointer which has been compiled into the DLL, it is not the width of a specific process, which could be different from this.

6.9.5 mqs_dll_error_string

Function type definition:

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

Provides a text string for an error value. The error value shall be given as an argument to the function.

6.10 Executable Image Related Functions

6.10.1 mqs_setup_image

Function type definition:

```
extern int mqs_setup_image( mqs_image*, const mqs_image_callbacks* );
```

Setups debug information for a specific image, this must save the callbacks, and use those functions for accessing this image. The image and the pointer to the `mqs_image_callbacks` table, in the order mentioned, shall be given as arguments to the function. The DLL should use the `mqs_put_image_info` and `mqs_get_image_info` functions to associate the information it wants to keep with the image. 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 program.

6.10.2 mqs_image_has_queue

Function type definition:

```
extern int mqs_image_has_queues( mqs_image*, char** );
```

Returns whether this image have the necessary symbols to allow access to the message queue. The image and the name of the queues, in the order mentioned, shall be given as arguments to the function. This function is called once for each image, and the information cached within the debugger.

6.10.3 mqs_destroy_image_info

Function type definition:

```
extern int mqs_destroy_image_info( mqs_image_info* );
```

Allows for cleaning up when the image information is no longer needed.

6.11 Process Related Functions

6.11.1 mqs_setup_process

Function type definition:

```
extern int mqs_setup_process( mqs_process*, const mqs_process_callbacks* );
```

Setups process specific information. The process and the pointer to the `mqs_process_callbacks` table, in the order mentioned, shall be given as arguments to the function.

6.11.2 mqs_process_has_queue

Function type definition:

```
extern int mqs_process_has_queues( mqs_process*, char** );
```

Similar to the `mqs_process_has_queues` function, this allows for querying whether process has support for message queues. The process and the names of the queues, in the order mentioned, shall be given as arguments to the function. This function should only be called if the image claims to provide message queues. For example, the image might have enabled message queues support if only certain environment variables are set at launched. This function checks whether at runtime, message queues support is enabled for the process.

6.11.3 mqs_destroy_process_info

Function type definition:

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

Allows for cleaning up when the process information is no longer needed.

6.12 Query Functions

These functions provide the message queue query functionality. The model here is that the debugger calls down to the library to initialize an iteration over a specific class of things, and then keeps calling the "next" function until it returns `mqs_false`. 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, so the library should save the iteration state in the `mqs_process_info`.

6.12.1 mqs_update_communicator_list

Function type definition:

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

Given a process, refreshes the list of active communicators. The process pointer shall be given as an argument to the function. Ideally this list is cached somewhere within the DLL and the debugger invokes it when necessary. The function returns `msq_ok` if the operation succeeds.

6.12.2 mqs_setup_communicator_iterator

Function type definition:

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

Given a process, prepares the iterator to walk the communicator list. The process pointer shall be given as the argument for the function. The function returns `msq_ok` if the operation succeeds.

6.12.3 mqs_get_communicator

Function type definition:

```
extern int mqs_get_communicator( mqs_process*, mqs_communicator* );
```

Extracts information about the current communicator. The process pointer and pointer to the output communicator, in the order mentioned, shall be given as arguments to the function. The function returns `msq_ok` if the operation succeeds.

6.12.4 mqs_get_comm_group

Function type definition:

```
extern int mqs_get_comm_group( mqs_process*, int* );
```

Extracts the group from the current communicator. The debugger already knows the size of the communciator, so it should 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 rank `i-th` in the current communicator. The function returns `msq_ok` if the operation succeeds.

6.12.5 mqs_next_communicator

Function type definition:

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

Moves the internal iterator to the next communicator in this process. The function returns |mqs_ok— if the operation succeeds.

6.12.6 mqs_setup_operation_iterator

Function type definition

```
extern int mqs_setup_operation_iterator( mqs_process*, int);
```

Prepares to iterate over the pending operations in the currently active communicator in this process. The process pointer and the type of pending operations (see `mqs_op_class`), in the order mentioned, shall be given as arguments to the function. The function returns `msq_ok` if the operation succeeds.

6.12.7 mqs_next_operation

Function type definition:

```
extern int mqs_next_operation( mqs_process*, mqs_pending_operation* );
```

Fetches information about the next appropriate pending operation in the current communicator. The process pointer and the output pointer for result, in the order mentioned, shall be given as arguments to the function. The function returns `mqs_false` when there is no more operation to iterate.

Chapter 7

The MPI Handle Introspection Interface

A wonderful chapter will be here.