Didn't we agree on 1 document w/ MPIR?

Chapter 1

Background

In early 1995, TotalView's Jim Cownie and Argonne National Laboratory's Bill Gropp and Rusty Lusk decided to ~~join forces and~~ develop parallel debugging interfaces for use with MPICH, one of the first widely available MPI implementations. Two interfaces were developed: one for process discovery/acquisition and one for message queue extraction. 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 ~~the~~ a MPIR document which ~~described~~ formally standardized the MPIR Process Acquisition Interface but left out the details about about the interface for message queue extraction (MQS). This document ~~attempts to fill such gap~~ compliments the 2010 MPIR document by describing the existing MQS interfaces being used by most MPI debuggers and MPI implementations today to provide users with information about the message passing state of an MPI program.

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

Chapter 2

Overview

the terms

an

The message queue interface is used by tools and debuggers to extract information describing the conceptual message-passing state of ~~the~~ an 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~~ tool 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,~~ Note that this ~~the~~ document ~~might~~ uses the terms "tools" and "debuggers" interchangably.

Within each MPI ~~communication space~~ process, there are three distinct abstract message ~~queues,~~ 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~~ matched yet.

Use the 1st Font & all caps. Same apples throughout the doc.

The send and receive queues store information about all of the unfinished send and receive operations that the process has started within ~~the~~ a given communicator. These might result either from blocking operations such as MPI_Send and MPI_Recv or nonblocking operations such as MPI_Isend ~~or~~ and 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 been 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 a "wildcard" 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~~ describes these conceptual queues ~~from the implementation~~ 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 3

Definitions

\em

MQS

3.1 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 dynamic processes. For example, the phrase "MPI rank 0 process" denotes the process that is rank 0 in MPI_COMM_WORLD.

doesn't have to be rank 0, does it?

3.2 "Starter" Process Definition

\em

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

Many ~~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`.

3.3 MPIR Node Definitions

\em \em

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.

also

Chapter 4

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.

2 words

don't capitalize, unless you do it consistently thru the whole doc

I think it would be helpful to have a few high-level paragraphs (and picture?) describing the general scheme before diving into the details — just like we did in the MPIR doc.

Chapter 5

Interface Specifications

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

what do we care about the header file?

5.1 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.

what is this for?
Don't use this weird format from the MPIR doc.

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

↳ Filename

If this symbol is not present in the starter process...

5.2 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).

5.3 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*)

5.4 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.:

itemized list

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

5.5 Opaque Types Passed Through the Interface

The following ~~three types~~ are opaque types ~~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 these ~~this~~ type~~s~~ 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 explicit ~~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.

5.6 Constants and Enums

5.6.1 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 identify the ~~deal with the different~~ language type that the original target code was based on.

5.6.2 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~~ the queue ~~it is~~ in which it is interested ~~in~~.

5.6.3 mqs_interface_version

This constant defines the version of the interface header.

5.6.4 mqs_status

separate lines

```
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.

5.6.5 Other enums

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

This enum defines the various ~~result~~ return codes 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.

5.7 Concrete Objects Passed Through the Interface

5.7.1 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;
```

Provide a 1-sentence description of this struct

itemized list

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.

5.7.2 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

itemized list

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. (ref section #).

desired_local_rank stores the rank of the target or the source for the communicator from which this message was initiated.

list

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 → is it a boolean?

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. → in bytes?

system_buffer identifies whether this is a user or a system buffer. → boolean?

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. of the message.

actual_length holds the actual length. → in bytes?

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

5.8 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.

5.8.1 mqs_basic_callbacks

would be good to describe (at front of doc?) what debugger images + processes are.

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 size)

IN	size	number of bytes to allocate

Allocates a block of memory with the specified size. Be clear that the DLL is not allowed to call normal alloc routines like malloc - it can only call this func.

mqs_free_ft

Function type definition:

typedef void (*mqs_free_ft) (void* buf)

INOUT	buf	buffer to be freed

Frees a previously allocated memory. Ditto.

mqs_errorstring_ft

Function type definition:

typedef char* (*mqs_errorstring_ft) (int errno)

IN	errno	the error code to get the error string for

Converts an error code from the debugger into an error message. The function returns a null terminated error string that corresponds to the given error code.

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 image to receive the image info
IN	imageinfo	the image info to associate with the image

Associates the given image information with the given image. This allows the DLL to cache information . . . .

mqs_get_image_info_ft

Function type definition:

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

IN	image	the image to extract the image info from

Returns the image information associated with the given image.

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 image

Associates the given process information with the given process. This allows the DLL to cache information...

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.

5.8.2 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* process, mqs_target_type_sizes* sizes)

IN	process	the process to get the sizes from
OUT	sizes	the placeholder for the sizes

Retrieves the size information about common datatypes from the running process. See Section ___ for the definition of mqs-target-type-sizes 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 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 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.

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 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 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.

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

Given an image, returns the type associated with the given named type. 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.*)

→ and/or the linker may optimize out such type information,

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

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

5.8.3 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* process)

IN	process	the process to get the global rank for

Given a process, returns its rank in MPI_COMM_WORLD. wrong font

mqs_get_image_ft

Function type definition:

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

IN	process	the process to get the image for

Given a process, returns the image of which it is an instance.

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`.

mqs_target_to_host_ft

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.

5.9 Callbacks Provided by the DLL

5.9.1 mqs_setup_basic_callbacks

Function type definition:

no extra spaces around (and)

extern void mqs_setup_basic_callbacks(const mqs_basic_callbacks* cb)

IN	cb	the basic callbacks table to provide to the DLL

This function is called by the debugger to the DLL to provide the DLL with the basic callbacks table.

5.9.2 mqs_version_string

Function type definition:

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

Returns the DLL version.

5.9.3 mqs_version_compatibility

Function type definition:

```
extern int mqs_version_compatibility()
```

Returns the DLL compatibility level. → what are the acceptable values returned?

5.9.4 mqs_dll_taddr_width

Function type definition:

```
extern int mqs_dll_taddr_width()
```

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.

5.9.5 mqs_dll_error_string

Function type definition:

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

IN errno the error code to get the error string for

Provides a text string for an error value. Note that this function, which provides a mean for the debugger to get the string associated with an error returned from the DLL, complements the function `mqs_errorstring`, which provides a mean for the DLL to get the string associated with an error returned from the debugger.

5.10 Executable Image Related Functions

5.10.1 mqs_setup_image

Function type definition:

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

INOUT	image	the image to setup the callbacks table
IN	cb	the image callbacks table

~~Setups~~ Sets up debug information for a specific image. This must ~~save~~ cache the function callbacks provided, 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 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~~ image. This will be called once for each executable image in the parallel program.

5.10.2 mqs_image_has_queue

Function type definition:

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

IN	image	the image to query mqs functionality
OUT	message	buffer to store message from the DLL.

Returns whether this image have the necessary symbols to allow access to the message queues. This function is called once for each image, and the information is cached within the debugger. The function returns `mqs_ok` if mqs support can be provided by this image.

5.10.3 mqs_destroy_image_info

Function type definition:

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

IN	imageinfo	the image info to free

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

5.11 Process Related Functions

5.11.1 mqs_setup_process

Function type definition:

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

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

Setups process specific information.

5.11.2 mqs_process_has_queue

Function type definition:

```
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 DLL

Similar to the **mqs_process_has_queues** function, this allows for querying whether process has support for message queues. 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~~ this specific process.

5.11.3 mqs_destroy_process_info

Function type definition:

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

IN	processinfo	the process info to free

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

5.12 Query Functions

These functions provide the message queue query functionality. The model ~~here~~ is that the debugger calls down to the ~~library~~ DLL to initialize an iteration over a specific class of ~~things~~ items, and then keeps calling the "next" function until it ~~returns~~ gets 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.

5.12.1 mqs_update_communicator_list

Function type definition:

```
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. 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.

5.12.2 mqs_setup_communicator_iterator

Function type definition:

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

IN	process	the process to prepare the iterator

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

5.12.3 mqs_get_communicator

Function type definition:

```
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 the communicator data

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

5.12.4 mqs_get_comm_group

Function type definition:

```
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 communciator, so it ~~should allocate~~ will provide 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.

5.12.5 mqs_next_communicator

Function type definition:

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.

5.12.6 mqs_setup_operation_iterator

Function type definition

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`. The function returns `msq_ok` if the operation succeeds.

See section ___

5.12.7 mqs_next_operation

Function type definition:

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 is no more operation to iterate.