

Myrinet Express (MX):
A High-Performance, Low-Level,
Message-Passing Interface
for Myrinet

Version 1.2
October 01, 2006

Table of Contents

I OVERVIEW.....	1
I.1 UPCOMING FEATURES.....	3
II CONCEPTS.....	4
II.1 NAMING SCHEME.....	4
II.2 ENDPOINTS.....	4
II.3 ENDPOINT ADDRESSING.....	5
II.4 MATCHING.....	5
II.5 UNEXPECTED MESSAGES.....	6
II.6 REQUESTS.....	6
II.7 REQUEST STATES.....	7
III INITIALIZATION.....	9
III.1 LIBRARY INITIALIZATION.....	9
III.1.1 mx_init().....	9
III.1.2 mx_finalize().....	9
III.2 INFORMATION RETRIEVAL.....	11
III.2.1 mx_get_info().....	11
III.2.2 MX_NIC_IDS.....	13
III.2.3 MX_COUNTERS_*.....	13
III.3 ENDPOINT OPENING AND CLOSING.....	14
III.3.1 mx_open_endpoint().....	14
III.3.2 mx_close_endpoint().....	16
III.3.3 mx_wakeup().....	17
IV SPECIFYING ENDPOINTS.....	18
IV.1 HOSTNAMES AND NIC IDs.....	18
IV.1.1 mx_hostname_to_nic_id().....	18
IV.1.2 mx_nic_id_to_hostname().....	18
IV.2 BOARD NUMBERS AND NIC IDs.....	19
IV.2.1 mx_board_number_to_nic_id().....	19
IV.2.2 mx_nic_id_to_board_number().....	19
IV.3 ENDPOINT ADDRESSES.....	20
IV.3.1 mx_connect().....	20
IV.3.2 mx_iconnect().....	21
IV.3.3 mx_decompose_endpoint_addr().....	22
IV.4 LOCAL ENDPOINT ADDRESS.....	23
IV.4.1 mx_get_endpoint_addr().....	23
IV.5 ENDPOINT ADDRESS CONTEXT.....	24
IV.5.1 mx_set_endpoint_addr_context().....	24
IV.5.2 mx_get_endpoint_addr_context().....	24
V POINT-TO-POINT COMMUNICATION.....	25
V.1 SEND OPERATIONS.....	25
V.1.1 mx_isend().....	25

V.1.2	<code>mx_issend()</code>	26
V.2	RECEIVE OPERATIONS.....	30
V.2.1	<code>mx_irecv()</code>	30
VI	REQUEST STATE FUNCTIONS	33
VI.1	BUFFERED STATE.....	33
VI.1.1	<code>mx_ibuffered()</code>	33
VI.2	REQUEST COMPLETION.....	34
VI.2.1	<code>mx_test()</code>	34
VI.2.2	<code>mx_wait()</code>	35
VI.3	QUERYING FOR ANY COMPLETION.....	36
VI.3.1	<code>mx_peek()</code>	37
VI.3.2	<code>mx_peek()</code>	37
VI.4	OBTAINING THE CONTEXT.....	38
VI.5	COMBINED QUERY AND COMPLETION.....	38
VI.5.1	<code>mx_test_any()</code>	39
VI.5.2	<code>mx_wait_any()</code>	39
VII	PROBING.....	41
VII.1.1	<code>mx_iprobe()</code>	41
VII.1.2	<code>mx_probe()</code>	42
VII.1.3	<code>mx_register_unexp_callback()</code>	43
VII.1.4	<code>mx_register_unexp_handler()</code>	44
VIII	CONTROLLING THE PROGRESSION.....	46
VIII.1.1	<code>mx_disable_progression()</code> and <code>mx_reenable_progression()</code>	46
VIII.1.2	<code>mx_progress()</code>	46
IX	CANCELING MX REQUESTS.....	47
IX.1.1	<code>mx_cancel()</code>	47
IX.1.2	<code>mx_forget()</code>	47
X	APPLICATION PROGRAMMING NOTES.....	49
X.1	COMPLETING REQUESTS.....	49
X.2	MULTI-THREADED APPLICATIONS.....	49
X.3	STRINGIFIED ERRORS AND STATUSES.....	49
X.3.1	<code>mx_strerror()</code>	49
X.3.2	<code>mx_strerrorstatus()</code>	50
XI	ERROR HANDLING.....	51
XII	DISCONNECTION HANDLING.....	53
XII.1.1	<code>mx_disconnect()</code>	53
XII.1.2	<code>mx_set_request_timeout()</code>	53
XIII	USING MX IN THE KERNEL.....	55
XIII.1	ENABLING THE KERNEL LIBRARY.....	55
XIII.2	KERNEL-SPECIFIC BEHAVIOR.....	55

XIII.3	EXCEPTIONS TO THE GENERIC MX API.....	55
XIII.4	EXTENDED KERNEL API.....	55

I Overview

Myrinet Express (MX) is a high-performance, low-level, message-passing software interface tailored for Myrinet. MX exploits the processing capabilities embedded in the Myrinet NIC to provide exceptional performance for modern middleware interfaces such as MPI or VI, enables low-overhead Ethernet emulation at link speed, and offers a simple API for third-party Myrinet software developments.

The MX API package includes these design goals:

- Protected and independent access for user-level applications: MX endpoints virtualize the network interfaces at the process level, providing OS-bypass communication.
- Transparent memory registration: Most modern message-passing interfaces do not require explicit memory-registration operations. In MX, explicit memory registration by the application or middleware is avoided altogether by the use of PIO or memory copies for small messages, and is made implicit and very low cost for larger messages.
- Very low total latency for small messages: In order to minimize latency for small messages, MX implements an extremely short critical path without intermediate copies or memory registration.
- Fully asynchronous communication primitives: The initiation of any communication is separated from its completion. Once an operation has been initiated, the application is not involved until it checks or waits for it.
- Support for non-contiguous sends and receives: All MX communication primitives involving user-level data accept scatter-gather lists containing up to 256 contiguous segments. Different mappings can be used on the sender and receiver sides for the same communication, allowing spatial data transformations to be an implicit part of the communication. Some cases are actually supported through a memory copy which may limit the overall performance.
- Virtually unlimited number of pending sends and receives: While the number of pending sends and receives natively supported by MX at the NIC level is large, MX offers a multiplexing capability to provide an unlimited number of pending sends and receives, bounded by the amount of available host resources (memory).
- Generic matching mechanism: MX provides an efficient matching mechanism between incoming messages and posted receives. The matching field is large enough (64 bits) to support the matching requirements of all modern message-passing interfaces.
- Overlap of communication and computation, even for large messages: As the MX communication primitives are fully asynchronous, it is possible for the application to continue its execution between the initiation of an operation and its completion. In the absence of unusual out-of-resource conditions, MX does not require the user-level application to be involved in the progression of the protocol, thus allowing overlap between communication and computation.
- Reliable in-order matching: MX provides an ordered matching protocol. Two messages sent from one endpoint in-order will match posted receives in-order, but

may be delivered out-of-order, or their completion may be notified out-of-order. The in-order matching is sufficient to support all of the modern message-passing interfaces, and the out-of-order delivery allows MX to use multiple routes between NIC ports and multiple ports per NIC.

- Efficient support for unexpected messages: An unexpected message is one for which a matching receive request has not yet been posted. Unexpected messages are processed by receiving the entire message eagerly in an unexpected queue if it is small, and by receiving only its header if it is too large. The size threshold distinguishing the handling methods can be controlled by the application. MX guarantees in-order matching, even if unexpected messages have been buffered.
- Transient network fault recovery and high-availability support: No network fabric is perfect, and transient errors (corruption, loss of packets) may occur, although not frequently in Myrinet fabrics. MX automatically recovers from any faults where recovery is possible through means such as retransmitting packets or routing around dead links. Catastrophic or unrecoverable errors due to hardware or software failure will be communicated to the application for handling by a higher-level recovery strategy.
- Basic per-message authentication mechanism: Messages in MX include a user-supplied identifier (called a filter) that provides a basic authentication mechanism between the source and the destination endpoints. Messages sent with a filter value that does not match that of the destination endpoint will be rejected at the NIC level.
- Per-message or per-endpoint polling or blocking completion functions: MX provides functions to check the completion of a specific pending operation or the completion of any of the pending operations related to an endpoint. Similarly, there are functions to block waiting for completion of a specific pending operation or the completion of all of the pending operations related to an endpoint. These blocking semantics release the processor for other application computation.
- Per-call timeouts on blocking completion functions: MX functions used to block on a specific operation or on all operations of an endpoint take a timeout as an argument. The granularity of this timeout is one millisecond.
- Efficient support for multiple links per NIC: MX does segmentation and re-assembly of large messages at the NIC level, and does not ensure in-order delivery or notification, only in-order matching. Support is provided to ensure reliability for a large number of out-of-order fragments (packets) without requiring retransmission, leading to high efficiency on all links. Intentional dropping of out-of-order packets and reliance on retransmission is used only when the number of out-of-order packets exceeds available buffering resources.
- Support for route dispersion: Multiple routes are computed for each destination. When necessary, different routes are chosen to avoid hot spots on the network fabric, and to achieve resistance to individual link failures.
- Integrated Myrinet fabric mapping: Network topology discovery and route computation is performed automatically as soon as the MX driver is loaded, and the network is automatically remapped as necessary when a network connectivity issue is detected.

- Support for cancellation of pending receive requests: Pending receive operations can be canceled in MX. The cancel operation will fail gracefully if the pending request has completed asynchronously while its cancel is attempted.
- Single-threaded and thread-safe libraries: MX provides both thread-safe and single-threaded libraries to allow users to select which is most appropriate for the application. There is no API difference between the two libraries; they are completely source and binary compatible. Thread safety is transparent to the application; many threads can initiate sends or receives, or even wait on different handles at the same time. The single-threaded MX library is provided so that applications may avoid the overhead of thread-safe operations if they are not needed.

1.1 Upcoming Features

Future releases of MX will include support for:

- One-sided communication and collective operations such as barrier, broadcast and all reduce.
- A recovery mechanism from the majority of SRAM parity errors: The MX software will have the capability of transparently reinitializing the NIC firmware and data structures. When an SRAM parity error occurs, in many cases it is recoverable. In a few cases, however, it will not be possible to recover and a reboot of the host will still be required for security purposes.

II Concepts

This section describes the terms used in the MX API and how they relate to each other.

Host and **process** have their usual meanings as in UNIX and Windows parlance. MX provides a mechanism for processes on the same or different hosts to communicate with each other through a Myrinet communication fabric. Each individual Myrinet card in a host is called a physical NIC, or just **NIC**. Processes communicate using MX by opening **endpoints** that are associated with NICs.

All messages in MX contain **matching** information. This information is used to match incoming sends to receive requests. In order to **receive** a message, a receive request is posted. To **send** a message, a send request is posted. Both are MX calls that specify matching information, a destination endpoint, and a list of user memory segments and their respective lengths.

Specific code examples will follow, but the typical flow of a process wishing to communicate with another is thus: initialize the library, open an endpoint, connect to your target host(s), start sending and receiving messages commingled with calls to collect status on these operations, close the endpoint, and finalize the library.

II.1 Naming Scheme

Many MX functions have both a blocking and a non-blocking variant. The naming scheme of the MX functions strongly reflects the MPI naming scheme in order to facilitate the ease of understanding of the semantics of the functions. As in the MPI standard, the letter 'i' prefixed in the function name denotes a non-blocking operation, which returns immediately. The blocking counterparts of these function names do not contain the letter 'i' and will block until completion or expiration of a timeout, effectively suspending the execution of the current process. As an example, **mx_iprobe()** is a non-blocking function whereas **mx_probe()** is its blocking counterpart. However, the blocking variant of a function is not always defined in the MX API. For example, **mx_isend()** is the non-blocking sending function, but **mx_send()** is not provided by MX.

II.2 Endpoints

An MX **endpoint** is a virtualization of a network interface at the process level. A process is defined from the UNIX point of view as a collection of execution threads sharing the same virtual address space. An **endpoint** provides an entry point to the interconnect hardware, protected from other processes, with fairness relative to the other endpoints opened on the same NIC or collection of NICs.

An endpoint is also an instance of a software interface. It is referenced by a variable of type **mx_endpoint_t**, used by many of the MX operations. All operations on an open

endpoint are restricted to this endpoint. MX objects, such as send and receive request handles, are relative to a specific endpoint and have no meaning to another endpoint, even opened by the same process.

Nothing prevents a process from simultaneously opening several endpoints on the same network interface.

An endpoint is created by a call to **mx_open_endpoint()**, which returns a handle for referencing the endpoint in an **mx_endpoint_t**. If **mx_open_endpoint()** does not return **MX_SUCCESS**, then the **mx_endpoint_t** passed in will remain unmodified.

A value of **NULL** is guaranteed never to be a valid endpoint.

II.3 Endpoint Addressing

In order to communicate with a remote endpoint, an application must have the **endpoint address** of that remote endpoint, represented by an **mx_endpoint_addr_t**. An endpoint address can be constructed from three pieces of information, the **NIC ID** of a NIC on the remote host (which may be the same as the local host), an **endpoint ID**, and an **endpoint filter** value. An **mx_endpoint_addr_t** is created by a call to **mx_connect()**, and can only be used with the local endpoint used in the **mx_connect()** call

Each NIC has a unique 64-bit **NIC ID** (for Myrinet NICs, this is the MAC Address encoded as a 64-bit number), and it is this **NIC ID** that is used to create an endpoint address. The IDs of the NICs within a given host can be queried from any application on the Myrinet using **mx_board_number_to_nic_id()**.

The **endpoint ID** is an integer associated with each open endpoint that can be assigned either by the application that opens it, or by the MX library. This value is an index and must be within the range [0...(MX_MAX_ENDPOINTS-1)].

The **endpoint filter** is an integer that is assigned by the application to distinguish between different instances of the application. Through careful use of this parameter, the application can “filter out” MX messages from lingering or zombie processes attempting to communicate with the previous owner of a particular endpoint ID.

II.4 Matching

The **matching** in MX is the process of associating an incoming message to a pending receive. Each message-passing interface defines its own matching rules based on elements provided by the sending side and/or the receiving side. A rich matching capability is required to build a complex message-passing interface on top of a low-level interface, or directly implement applications on top of it.

MX provides a flexible yet powerful matching interface. Each message in MX contains 64 bits of matching information. The sender specifies this information, *match_send*, as part of the sending operation, and the receiving side provides *match_rcv* and a *match_mask* when posting a receive. An incoming message will be associated with a pending receive if and only if the incoming *match_send* data masked with the *match_mask* matches the *match_rcv* information of the posted receive.

II.5 Unexpected Messages

A sub-optimal yet common occurrence in message passing is to send a message before a matching receive is posted on the receive side. This occurrence can be due to a slight timing drift or, more simply, to poor application programming methods. A message that arrives on the receive side without a matching receive is, in many low-level interfaces, dropped and retransmitted later.

In MX, a buffer is allocated at endpoint opening time to handle such unexpected messages. Inasmuch as it is often the case that the receive operation might be a little bit late and be posted just after the incoming message arrives, it is appropriate to copy the unexpected message into a temporary area. Then, when a matching receive is posted by the application, the message can be delivered immediately.

This unexpected buffer is limited in size, so only small messages will be buffered in this way. Larger messages will leave their matching information along with information about the sending endpoint. The threshold in message size between a full copy in the unexpected buffer and a copy of only the header (matching information and sender endpoint address) is specified by the application when the endpoint is opened.

Unexpected message handling is transparent at the MX API level. When a receive is posted, the application does not need to know if the incoming matching send has already been saved in the unexpected queue. If this is the case and if the message was small enough to fit in its entirety, the message is delivered and the receive is completed immediately. If the message was larger than the unexpected threshold set by the application, MX will notify the sending side that a matching receive has been posted and this will trigger an immediate transmission from the sender, without involvement of the application on the send side. If no matching unexpected messages are found, the receive information is recorded for matching against future incoming messages.

This mechanism provides an efficient eager protocol for small messages and a loose rendezvous protocol for larger messages, allowing overlap of communication and computation even in the case of unexpected messages.

II.6 Requests

Requests are identifiers used to specify particular instances of pending asynchronous operations. All asynchronous MX operations fill in an **mx_request_t** object passed in by the application, which is used to specify this pending operation to subsequent interface

calls. These handles are generated by **mx_isend()**, **mx_issend()** and **mx_irecv()**, and can be passed as arguments to **mx_test()**, **mx_wait()**, **mx_ibuffered()**, and **mx_cancel()**. If any of the functions that fill in an **mx_request_t** object does not return **MX_SUCCESS**, the **mx_request_t** object passed in will remain unchanged.

Every posted receive must have a matching successful call to **mx_test()** or **mx_wait()** in order to release and recycle the resources associated with the request. The request handle of a completed operation ceases to be a valid argument to any subsequent MX function calls, unless and until the same value is assigned to a newly posted request.

A value of “NULL” is guaranteed not to be a valid request.

II.7 Request States

Once a request has been posted, it enters a three-state life cycle. The states of this life cycle are **pending**, **buffered (for send requests only)**, and **complete**. **Pending** means that the request has been delivered to the MX subsystem, and it is in progress. Once the buffers associated with a request can be safely used by the posting application, the request enters the **buffered** state. At this time the buffers can be read or written without affecting the outcome of the request. Finally, when all activity needed for a request has finished, the request enters the **complete** state.

The progression through the various states is different for different request types. A receive request enters the **pending** state when issued, and remains there until a matching message has been placed in the associated buffers. At this time, the request changes directly to the **complete** state.

A send request enters the **pending** state when issued, but the subsequent state transitions are slightly different for **mx_isend()** and **mx_issend()**.

For requests posted with **mx_isend()**, once the data has been copied out of the associated buffers, possibly into a queue of unexpected messages on the receiving node, the request changes directly to the **complete** state.

For requests posted with **mx_issend()** (the second ‘s’ is for “synchronous”), the **buffered** state is used. Once the data being sent has been copied out of the buffers, possibly into a queue of unexpected messages on the receiving node, the request enters the **buffered** state. Only after a matching receive has been issued on the receiving side does the request enter the **complete** state. In this case, the **complete** state can be used as a synchronization point with the receiver.

There are five functions used to observe and wait on the state of requests. These are **mx_ibuffered()**, **mx_test()**, **mx_wait()**, **mx_peek()** and **mx_peek()**.

mx_ibuffered() returns **MX_SUCCESS** if the referenced request has been safely buffered and the memory buffers associated with the request may be reused. If a request

is in either the **buffered** or **complete** state, **mx_ibuffered()** will return **MX_SUCCESS**. Calls to **mx_ibuffered()** do not affect the state of the request in any way.

mx_test() returns **MX_SUCCESS** if the referenced request has completed. In this case, the status structure referenced in the call will be updated with more detailed information about the requests completion. A successful return from **mx_test()** does not mean that the request was successful, just that it is complete. The status structure will contain all codes related to the outcome of the request, such as successful, cancelled, or rejected. After a successful return from **mx_test()**, the referenced request no longer exists as far as MX is concerned. If the return from **mx_test()** is unsuccessful for any reason, the resources associated with the request are not released. **mx_wait()** is the blocking counterpart of **mx_test()**.

mx_peek() returns the handle of a request for a specific endpoint that is in the **complete** state. If **mx_peek()** returns **MX_SUCCESS**, the returned request handle is guaranteed to be successfully completed in a call to **mx_test()** or **mx_wait()**. If multiple requests are in the **complete** state in the endpoint, only one of them will be returned by **mx_peek()**, but which one is non-deterministic. **mx_peek()** is the blocking variant of **mx_peek()**.

III Initialization

III.1 Library Initialization

III.1.1 mx_init()

Before any other MX calls may be made, the library must be initialized by a call to **mx_init()**:

```
| mx_return_t  
| mx_init(void);
```

If the MX library has been successfully initialized, **mx_init()** returns **MX_SUCCESS**. Otherwise, the function may return one of the following error codes.

Error return codes:

MX_NO_DEV	The OS-specific /dev/mx devices are not present.
MX_NO_DRIVER	The MX driver does not seem to be loaded.
MX_NO_PERM	No permission to access the mx device.
MX_BAD_BAD_BAD	Something bad happened with the driver, maybe the wrong driver. You need to check the kernel log.
MX_ALREADY_INITIALIZED	mx_init() has already been called.
MX_NO_RESOURCES	Shortage of memory or other system resources.

This function allocates and initializes all data structures used by the MX API library.

III.1.2 mx_finalize()

The complement of the **mx_init()** function is **mx_finalize()**:

```
| mx_return_t  
| mx_finalize(void);
```

The current implementation of **mx_finalize()** always returns **MX_SUCCESS**.

This function cleans up the MX library and releases any resources previously allocated.

Example III.1: Initialization of the MX library.

```
| #include "myriexpress.h"  
|  
| int
```

```
main(void)
{
    mx_return_t return_code;

    /* Initialize the MX library */
    return_code = mx_init();

    /* do work here ... */

    /* Finalize the MX library */
    return_code = mx_finalize();

    return 0;
}
```

III.2 Information Retrieval

III.2.1 mx_get_info()

A variety of information about MX or about a specific endpoint can be retrieved using **mx_get_info()**:

```
mx_return_t  
mx_get_info(mx_endpoint_t endpoint,  
            mx_get_info_key_t key,  
            void *in_val,  
            uint32_t in_len,  
            void *out_val,  
            uint32_t out_len);
```

Parameters:

IN	endpoint	The MX endpoint to focus the scope of the information inquiry; NULL if information is global.
IN	key	The enumeration value of the information key.
IN	in_val	A pointer to the parameters needed for this call.
IN	in_len	The length of the memory area referenced by <i>in_val</i> .
OUT	out_val	A pointer to the memory area where the requested information will be placed.
IN	out_len	The size of the out_val buffer.

If the value of the specified information key has been successfully retrieved, **mx_get_info()** returns MX_SUCCESS. Otherwise, the function may return one of the following error codes.

Error return codes:

MX_BAD_INFO_KEY	Unknown key.
MX_BAD_INFO_LENGTH	The buffer length is too small.

mx_get_info() provides a way to obtain information about MX at the global level of the library or at the limited level of an MX endpoint. Such information is accessible under the form of (**key**, **value**) pairs where the key is an enumeration and the value can be of multiple types. The size of the content of the value is specific for each information key.

If the *endpoint* parameter is NULL, the information retrieval applies to the MX library itself. If information associated to a specific endpoint is requested, then the parameter *endpoint* must be defined to the appropriate MX endpoint.

The argument *key* is one value of the enumeration referencing all the information keys. If this key is not recognized as one of the valid keys listed below, the return code

MX_BAD_INFO_KEY is returned. The parameter *in_val* is a pointer to a memory area which contains any needed parameters for that *key* request. The parameter *in_len* is the length associated with *in_val*. The parameter *out_val* is the memory where the information requested will be returned. The argument *out_len* is the size of the memory area designated by *out_val*. If this size is not large enough to contain the value associated with the *key*, MX_BAD_INFO_LENGTH is returned and the contents of the memory referenced by *out_val* are undefined.

The following keys are recognized as valid:

Global Information (endpoint can be NULL)	
Key	MX_MAX_NATIVE_ENDPOINTS
Description	The maximum number of endpoints interfaced directly with the NIC (thus providing OS-bypass).
Input	None
Output	uint32_t
Output Size	sizeof(uint32_t)
Key	MX_NIC_COUNT
Description	The number of NICs available to this application.
Input	None
Output	uint32_t
Output Size	sizeof(uint32_t)
Key	MX_NIC_IDS
Description	Identifier (MAC address) of all NICs in the system in a 0-terminated array. (see Section III.2.2)
Input	None
Output	uint64_t[]
Output Size	variable * sizeof(uint64_t)
Key	MX_NATIVE_REQUESTS
Description	The number of requests that can be handled natively by the NIC.
Input	None
Output	uint32_t
Output Size	sizeof(uint32_t)
Key	MX_COUNTERS_COUNT
Description	The number of counters in the count table.
Input	uint32_t, the board id
Output	uint32_t
Output Size	sizeof(uint32_t)
Key	MX_COUNTERS_LABELS
Description	The text names for each counter.
Input	uint32_t, the board id
Output	uint8_t[][MX_MAX_STR_LEN]
Output Size	Variable * MX_MAX_STR_LEN
Key	MX_COUNTERS_VALUES
Description	The counters' values
Input	uint32_t, the board id
Output	uint32_t[]

Output Size	variable * uint32_t
Key	MX_PRODUCT_CODE
Descriptions	The product string for this Myrinet NIC.
Input	uint32_t, the board id
Output	uint8_t[MX_MAX_STR_LEN]
Output Size	MX_MAX_STR_LEN
Key	MX_PART_NUMBER
Description	The part number string for this Myrinet NIC.
Input	uint32_t, the board id
Output	uint8_t[MX_MAX_STR_LEN]
Output Size	MX_MAX_STR_LEN
Key	MX_SERIAL_NUMBER
Description	The serial number for this Myrinet NIC.
Input	uint32_t, the board id
Output	uint32_t
Output Size	sizeof(uint32_t)
Key	MX_PORT_COUNT
Description	The number of ports for this Myrinet NIC.
Input	uint32_t, the board id
Output	uint32_t
Output Size	sizeof(uint32_t)

III.2.2 MX_NIC_IDS

Before calling **mx_get_info()** with the key `MX_NIC_IDS`, an application should first call **mx_get_info()** with the key `MX_NIC_COUNT`. A subsequent call with `MX_NIC_IDS` will fill in an array of NIC IDs in `uint64_t`'s followed by a 0. Thus, the memory area passed to **mx_get_info()** for `MX_NIC_IDS` should be large enough to hold $N+1$ 64-bit integers, where N is the number returned by `MX_NIC_COUNT`.

For example, if `MX_NIC_COUNT` indicates there are 2 NICs in the system, `MX_NIC_IDS` should be passed an array with size at least $3 * \text{sizeof}(\text{uint64}_t)$. The first two elements of the array will contain the two NIC IDs, and the third element will be zero. This array is terminated with a zero.

III.2.3 MX_COUNTERS_*

Before calling **mx_get_info()** with the key `MX_COUNTERS_LABELS`, or the key `MX_COUNTERS_VALUES`, an application should first call **mx_get_info()** with the key `MX_COUNTERS_COUNT`. The memory area passed to the *out_val* parameter of **mx_get_info()** should be large enough to hold the data returned. For `MX_COUNTERS_LABELS`, this should be $N * \text{MX_MAX_STR_LEN}$, and for `MX_COUNTERS_VALUES` this should be $N * \text{sizeof}(\text{uint32}_t)$, where N is the number returned by `MX_COUNTERS_COUNT`.

III.3 Endpoint Opening and Closing

III.3.1 mx_open_endpoint()

Once the MX library is initialized, the application needs to open an **endpoint** to be able to send or receive messages. This operation is performed by the function **mx_open_endpoint()**:

```
mx_return_t  
mx_open_endpoint(uint32_t board_num,  
                uint32_t endpoint_id,  
                uint32_t filter,  
                mx_param_t *params_list,  
                uint32_t params_count,  
                mx_endpoint_t *endpoint);
```

Parameters:

IN	board_num	The local board rank of the NIC on which MX will try to open an endpoint.
IN	endpoint_id	The index of the endpoint to open on the specified NIC.
IN	filter	A user-assigned value used to filter incoming messages and reject mx_connect() (or any unauthorized messages).
IN	params_list	The array of parameters that specifies the configuration of the endpoint to open; NULL if no parameters.
IN	params_count	The number of entries in the array of parameters, 0 if no parameters.
OUT	endpoint	The MX endpoint successfully opened.

If the endpoint has been successfully opened, **mx_open_endpoint()** returns **MX_SUCCESS**. Otherwise, the function may return one of the following error codes.

Error return codes:

MX_BUSY	The endpoint (or all possible endpoint matching requirements if wildcards are used) is (are) busy.
MX_NO_DEV	Some OS-specific /dev/mx devices are not present.
MX_NO_DRIVER	The MX driver does not seem to be loaded.
MX_NO_PERM	No permission to access the mx device.
MX_BAD_BAD_BAD	Something bad happened with the driver, maybe the wrong driver. You need to check the kernel log.

MX_BOARD_UNKNOWN	Invalid board number.
MX_BAD_PARAM_LIST	The parameter list is empty while the count is not null.
MX_BAD_PARAM_NAME	The parameter key is invalid.
MX_NO_RESOURCES	Shortage of memory or other system resources.

mx_open_endpoint() opens a specific MX endpoint if available. This function requires a pointer to an **mx_endpoint_t** object allocated by the application. This object should be passed to all of the other MX functions operating on an MX endpoint. Opening an endpoint also creates it, and these terms may be used interchangeably in this document. If the return value is not **MX_SUCCESS**, then the *endpoint* passed in to **mx_open_endpoint()** remains unmodified.

A board number *board_num*, is passed to specify with which NIC this endpoint should be associated. This is referred to as the **primary NIC** for the endpoint.

The application can let MX choose the best NIC on which to open an endpoint by using the **MX_ANY_NIC** constant.

The second input parameter is the index of the endpoint to be opened. This endpoint number must be in the [0, (MX_MAX_ENDPOINTS-1)] range. The value of **MX_MAX_ENDPOINTS** can be retrieved using **mx_get_info()**.

The application can let MX choose the best endpoint to open by using the **MX_ANY_ENDPOINT** constant.

The *params_list* argument is a pointer to an array of **mx_param_t** entries. This array specifies the user configuration of the requested endpoint. MX endpoint parameters are (key, value) pairs where the keys are member of an enumeration and the values are pointers to memory areas, allocated by the application and containing the values of the respective parameters.

The *params_count* parameter specifies the number of entries in the list of endpoint parameters. The *params_list* argument may be NULL, along with a *params_count* of 0, in which case default values are used for all settings.

The following keys are recognized as valid:

Parameter	MX_PARAM_UNEXP_QUEUE_MAX
Description	Sets the maximum length of the unexpected queue.
Format	uint32_t
Size	sizeof(uint32_t)
Default Value	Value of the MX_UNEXP_QUEUE_LENGTH_MAX
Parameter Key	MX_PARAM_ERROR_HANDLER
Description	Sets the error handler.
Format	mx_error_handler_t
Size	sizeof(mx_error_handler_t)
Default Value	No error handler

III.3.2 mx_close_endpoint()

Once opened, an MX endpoint can be closed. This operation is performed by the function **mx_close_endpoint()**:

```
mx_return_t  
mx_close_endpoint(mx_endpoint_t endpoint);
```

Parameters:

IN	endpoint	The MX endpoint to close.
----	----------	---------------------------

The current implementation of **mx_close_endpoint()** returns `MX_SUCCESS` if the endpoint was successfully closed. Other possible return codes include:

Error return codes:

<code>MX_NOT_INITIALIZED</code>	The library has not been initialized.
<code>MX_BAD_ENDPOINT</code>	The endpoint parameter is not a valid endpoint.
<code>MX_CLOSE_IN_UNEXPECTED_CALLBACK</code>	Called inside the unexpected callback.

mx_close_endpoint() closes an opened MX endpoint. All pending operations are cancelled and the endpoint is deregistered from the NIC. This function requires a pointer to the **mx_endpoint_t** that references the MX endpoint to close.

The endpoint is closed immediately but cannot be reopened until all messages in flight have been dropped. To satisfy this condition, the endpoint may remain unusable for a brief period of time.

III.3.3 mx_wakeup()

The function **mx_wakeup()** causes blocking functions to abort their wait.

```
mx_return_t  
mx_wakeup(mx_endpoint_t endpoint);
```

Parameters:

IN	endpoint	The MX endpoint associated with the blocking call.
----	----------	--

The current implementation of **mx_wakeup()** always returns `MX_SUCCESS`.

mx_wakeup() is useful in multithreaded applications where it may be necessary to notify a thread that the current blocking operation will never be satisfied.

Example III.2: Allocation and release of a MX endpoint.

```
/* error checking excised for brevity */

#include "myriexpress.h"

int
main(void)
{
    mx_return_t rc;
    mx_endpoint_t endpoint;
    uint32_t filter;

    /* Initialize the MX library */
    rc = mx_init();

    /* open an MX endpoint */
    filter = 0xcafebabe; /* app specific unique value */
    rc = mx_open_endpoint(MX_ANY_NIC, MX_ANY_ENDPOINT,
                        filter, 0, 0, &endpoint);

    /* do work here ... */

    /* close the MX endpoint */
    rc = mx_close_endpoint(endpoint);

    /* Finalize the MX library */
    rc = mx_finalize();

    return 0;
}
```

Once an endpoint has successfully been opened, it can be used to post asynchronous send and receive operations, and test or wait for their completion.

IV Specifying Endpoints

IV.1 Hostnames and NIC IDs

IV.1.1 mx_hostname_to_nic_id()

In order to facilitate identifying remote hosts, MX provides utility functions, **mx_hostname_to_nic_id()** and **mx_nic_id_to_hostname()**, to convert from a hostname to a NIC ID and vice-versa. The hostname in the context of these functions is actually a `nic_name`, it is different for each NIC on multi-NIC hosts, and is initialized by default to `<hostname>:<board-rank>`.

mx_hostname_to_nic_id() returns the NIC ID given the name of a NIC:

```
mx_return_t  
mx_hostname_to_nic_id(char *hostname,  
                      uint64_t *nic_id);
```

Parameters:

IN	hostname	The name of the host whose NIC ID we want.
OUT	nic id	The NIC ID of the host.

If the NIC ID for the specified host has been successfully retrieved, **mx_hostname_to_nic_id()** returns `MX_SUCCESS`. Otherwise, the function may return one of the following error codes.

Error return codes:

<code>MX_HOST_NOT_FOUND</code>	The hostname was not found in the network peer table.
--------------------------------	---

IV.1.2 mx_nic_id_to_hostname()

The complementary routine, **mx_nic_id_to_hostname()**, converts a NIC ID to a hostname.

```
mx_return_t  
mx_nic_id_to_hostname(uint64_t nic_id,  
                      char *hostname);
```

Parameters:

IN	nic id	The NIC ID of the host whose name we want.
----	--------	--

OUT	hostname	The name of the host.
-----	----------	-----------------------

If the hostname for the specified NIC ID has been successfully returned, **mx_nic_id_to_hostname()** returns MX_SUCCESS. Otherwise, the function may return one of the following error codes.

Error return codes:

MX_HOST_NOT_FOUND	No such NIC is in the network peer table .
-------------------	--

Note that MX_MAX_HOSTNAME_LEN includes a trailing '0' used in C string representations.

IV.2 Board numbers and NIC IDs

IV.2.1 mx_board_number_to_nic_id()

In order to facilitate identifying a specific NIC when there are multiple NICs in the same host, MX provides utility functions, **mx_board_number_to_nic_id()** and **mx_nic_id_to_board_number()**, to convert from a board number to a NIC ID and vice-versa.

mx_board_number_to_nic_id() returns the MAC address of a board with a given rank.

```

mx_return_t
mx_board_number_to_nic_id(uint32_t board_number,
                          uint64_t *nic_id);

```

Parameters:

IN	board number	The board number whose NIC ID we want.
OUT	nic id	The NIC ID assigned to this board number.

If the NIC ID for the board number has been successfully retrieved, **mx_board_number_to_nic_id()** returns MX_SUCCESS. Otherwise, the function may return one of the following error codes.

Error return codes:

MX_BOARD_UNKNOWN	Invalid board number.
------------------	-----------------------

IV.2.2 mx_nic_id_to_board_number()

The complementary routine, **mx_nic_id_to_board_number()**, is used if an application wants to open an endpoint on a NIC with a given MAC address. It converts the MAC address into a board rank, as is required by **mx_open_endpoint()**.

```

mx_return_t
mx_nic_id_to_board_number(uint64_t nic_id,
                          uint32_t *board_number);

```

Parameters:

IN	nic_id	The NIC ID assigned to the board number we want.
OUT	board_number	The board number.

If the board number for the specified NIC ID has been successfully returned, **mx_nic_id_to_board_number()** returns MX_SUCCESS. Otherwise, the function may return one of the following error codes.

Error return codes:

MX_BOARD_UNKNOWN	No local board was found with this nic_id.
------------------	--

IV.3 Endpoint Addresses

The function **mx_connect()** is used to build an MX endpoint address, and the function **mx_decompose_endpoint_addr()** is used to extract information from the MX endpoint address.

IV.3.1 mx_connect()

Remote endpoints are specified through the use of an **mx_endpoint_addr_t**. An **mx_endpoint_addr_t** is formed by combining the NIC ID of a network interface on the node on which the remote endpoint resides, the ID of the endpoint, and a filter key. An **mx_endpoint_addr_t** is initialized from these elements using the function **mx_connect()**, which checks that the remote endpoint is open and accepts our filter value. An **endpoint_addr_t** is endpoint specific. If a process has multiple local endpoints open, it will need to call **mx_connect()** for each local endpoint even if all the local endpoints will be talking to the same remote endpoint.

```
mx_return_t  
mx_connect(mx_endpoint_t endpoint,  
           uint64_t nic_id,  
           uint32_t endpoint_id,  
           uint32_t filter,  
           uint32_t timeout,  
           mx_endpoint_addr_t *endpoint_addr);
```

Parameters:

IN	endpoint	The local MX endpoint that will be used for communication.
IN	nic_id	NIC ID of remote node with which we wish to communicate.
IN	endpoint id	ID of the remote endpoint.
IN	filter	Filter value for the remote endpoint.
IN	timeout	Specifies the amount of time to wait for a connection, in seconds.
OUT	endpoint addr	The newly built endpoint address.

If the endpoint address has been successfully built, **mx_connect()** returns **MX_SUCCESS**. Otherwise, the function may return one of the following error codes.

Error return codes:

MX_NIC_NOT_FOUND	The target NIC was not found in the network peer table.
MX_CONNECTION_FAILED	The remote endpoint is closed.

MX_BAD_CONNECTION_KEY	Wrong credentials key, the peer rejected the connection or message.
MX_TIMEOUT	The specified timeout was exceeded while waiting for the target to reply.
MX_NO_RESOURCES	Shortage of memory or other system resources.

The **mx_endpoint_addr_t** returned by this function can be passed either to **mx_isend()** or **mx_issend()** for point-to-point communications.

IV.3.2 mx_iconnect()

This function is only available in mx_extensions.h for now.

mx_iconnect() is the asynchronous version of **mx_connect()**. It takes similar arguments and return a request associated to a matching value and a user context. Then, as any send or receive request, it may be tested or waited. Once completed, on success, the status of the request will be **MX_STATUS_SUCCESS** and the **mx_endpoint_addr_t** of the connected endpoint will be available through the **source** field of the status structure.

```

mx_return_t
mx_iconnect(mx_endpoint_t endpoint,
            uint64_t nic_id,
            uint32_t endpoint_id,
            uint32_t filter,
            uint64_t match_info,
            void * context,
            mx_request_t * request);

```

Parameters:

IN	endpoint	The local MX endpoint that will be used for communication.
IN	nic_id	NIC ID of remote node with which we wish to communicate.
IN	endpoint_id	ID of the remote endpoint.
IN	filter	Filter value for the remote endpoint.
IN	match_info	The matching information from that will be used when returning the request into mx_test_any() or mx_wait_any() .
IN	context	A user-defined pointer that will be passed back to the application as part of the status structure when this request completes.
OUT	request	The pointer to the MX Request object that references the pending send operation.

If the asynchronous connect request has been successfully posted, **mx_iconnect()** returns `MX_SUCCESS`. Otherwise, the function may return one of the following error codes.

Error return codes:

<code>MX_NIC_NOT_FOUND</code>	The target NIC was not found in the network peer table.
<code>MX_NO_RESOURCES</code>	Shortage of memory or other system resources.

IV.3.3 `mx_decompose_endpoint_addr()`

The function, **mx_decompose_endpoint_addr()** can be used to extract the information associated with `mx_endpoint_addr_t` (for instance to identify the source of a message from the `mx_status_t.source` field returned at receive completion).

```

mx_return_t
mx_decompose_endpoint_addr(mx_endpoint_addr_t endpoint_addr,
                           uint64_t *nic_id,
                           uint32_t *endpoint_id);

```

Parameters:

IN	<code>endpoint_addr</code>	An <code>mx_endpoint_addr_t</code> from which we wish to extract component parts.
OUT	<code>nic_id</code>	NIC ID of remote node to which this endpoint address refers.
OUT	<code>endpoint id</code>	ID of the remote endpoint.

The current implementation of **mx_decompose_endpoint_addr()** always returns `MX_SUCCESS`.

IV.4 Local Endpoint Address

IV.4.1 mx_get_endpoint_addr()

It is frequently useful to know the **endpoint address** of a local endpoint to either send a message to oneself, or extract the NIC id and endpoint id when using `MX_ANY_NIC` or `MX_ANY_ENDPOINT` to communicate it to others. The function **mx_get_endpoint_addr()** returns the endpoint address of an opened endpoint.

```
mx_return_t  
mx_get_endpoint_addr(mx_endpoint_t endpoint,  
                    mx_endpoint_addr_t *endpoint_addr);
```

Parameters:

IN	endpoint	The handle of the open local endpoint whose address we wish to know.
OUT	endpoint_addr	A pointer to an <code>mx_endpoint_addr_t</code> where the address of this endpoint is to be stored.

The current implementation of **mx_get_endpoint_addr()** always returns `MX_SUCCESS`.

IV.5 Endpoint Address Context

When an application describes the peer it communicates with with its own peer descriptor, when a new incoming message arrives, it might be required to translate its source address into a peer descriptor. Since it usually imply an expensive lookup in a hash table, MX provides the ability to store a reference to this descriptor in any **mx_endpoint_addr_t**. This reference must be set after the address is obtained, and remains valid as long as the connection is valid, i.e. as long as the local or remote endpoint is not closed.

IV.5.1 mx_set_endpoint_addr_context()

This function is only available in mx_extensions.h for now.

Once **mx_connect()**, **mx_icoonect()** or **mx_get_endpoint_addr()** returns a **mx_endpoint_addr_t** to the application, **mx_set_endpoint_addr_context()** may be used to associate any pointer to this address, known as an address context.

```
mx_return_t  
mx_set_endpoint_addr_context(mx_endpoint_addr_t address,  
                             void * context);
```

mx_set_endpoint_addr_context() always returns **MX_SUCCESS**. If called multiple times on the same address, the previous context will be overwritten with the new one.

IV.5.2 mx_get_endpoint_addr_context()

This function is only available in mx_extensions.h for now.

mx_get_endpoint_addr_context() returns the context associated with a **mx_endpoint_addr_t**, i.e. the pointer that the application stored previously.

```
mx_return_t  
mx_get_endpoint_addr_context(mx_endpoint_addr_t address,  
                             void ** context);
```

mx_get_endpoint_addr_context() always returns **MX_SUCCESS**.

V Point-to-point Communication

V.1 Send Operations

MX provides two functions to initiate asynchronous sends, **mx_isend()** and **mx_issend()**.

V.1.1 mx_isend()

mx_isend() follows the semantics of the *standard mode* in MPI: the request will be completed when the send buffer described by the gather list can be reused by the application. Completion of the operation does not give any indication on the fate of the message, either being buffered on the send or receive side, or matched by a posted receive on the receive side, or even lost due to fatal errors in the network.

```
mx_return_t  
mx_isend(mx_endpoint_t endpoint,  
         mx_segment_t *segments_list,  
         uint32_t segments_count,  
         mx_endpoint_addr_t destination,  
         uint64_t match_send,  
         void *context,  
         mx_request_t *request);
```

Parameters:

IN	endpoint	The local MX endpoint used to post the send.
IN	segments_list	The array of contiguous segments constituting the gather list describing the buffer to send.
IN	segments count	The number of segments in the gather list.
IN	destination	The MX Addr of the destination of the message.
IN	match_send	The matching information from the send side that will be used to find a matching receive on the remote side.
IN	context	A user-defined pointer that will be passed back to the application as part of the status structure when this request completes.
OUT	request	The pointer to the MX Request object that references the pending send operation.

If the send operation has been successfully posted, **mx_isend()** returns **MX_SUCCESS**. Otherwise, the function may return one of the following error codes.

Error return codes:

MX_BAD_SEG_COUNT	The segment list contains more than 256 segments,
MX_NO_RESOURCES	Shortage of memory or other system resources.

This function notifies the network interface that a new send is pending and returns to the application as soon as possible. The send buffer is described by a gather list via the parameter *segments_list* and a number of segments, *segments_count*. The gather list is an array of *mx_segment_t* structures. Each segment describes a contiguous memory area, using a pointer and a length. The maximum number of segments of a specific endpoint is available under the key `MX_MAX_SEGMENTS` via the `mx_get_info()` mechanism. The maximum length is an unsigned integer (32 bit). Thus, it is possible to send a contiguous buffer using only one segment, or a non-contiguous buffer without any constraints other than the maximum number of segments.

Segments of length 0 are allowed but ignored. Results are non-deterministic if segments within a segment list overlap. If the total length of the message is 0, it is then allowed to pass NULL as a list of segments and 0 as the number of segments.

The destination is specified by the parameter *destination*. It is an `mx_endpoint_addr_t` object returned by `mx_connect()`. The sender also provides the matching information for the message in the parameter *match_send*.

The parameter *context* specifies a user-defined pointer that will be included in the status structure returned when this post completes. When a pending send request is completed, either successfully or unsuccessfully, `mx_test()` or `mx_wait()` will return a status structure with the *context* field filled in with this user-supplied value. This mechanism may be used to implement callbacks on top of the status functions. The context can also be extracted from the request by `mx_context()`.

Finally, the last argument *request* is a pointer to an `mx_request_t` object, allocated by the application. This handle will be assigned by the library and used to reference the pending send operation when checking or blocking for its completion.

The data buffer(s) specified in a send operation must not be modified until the request is in the **buffered** state. This state is detected by a successful return from `mx_ibuffered()`. The segment list itself may be modified immediately after `mx_isend()` returns; however, the data buffers to which the list refers should not be modified until the operation is complete.

The operation is **complete** as soon as a call to `mx_test()` or `mx_wait()` indicates that this pending operation is complete. Note that being complete also indicates that the send buffers are available for the application. As `mx_isend()` follows the semantics of the MPI standard mode, a send request in the buffered state can be completed immediately by calling `mx_test()` or `mx_wait()`. Thus, there is no advantage to use `mx_ibuffered()` before `mx_test()` or `mx_wait()` on requests initiated by `mx_isend()`.

V.1.2 `mx_issend()`

MX also supports the concept of a synchronous send, which means that the send request is not considered complete until it is successfully received by the destination endpoint, it is cancelled, or an unrecoverable error has occurred sending the message. The function to initiate a non-blocking synchronous send is **mx_issend()**:

```

mx_return_t
mx_issend(mx_endpoint_t endpoint,
          mx_segment_t *segments_list,
          uint32_t segments_count,
          mx_endpoint_addr_t destination,
          uint64_t match_send,
          void *context,
          mx_request_t *request);

```

Parameters:

IN	endpoint	The local MX endpoint used to post the send.
IN	segments_list	The array of contiguous segments constituting the gather list describing the buffer to send.
IN	segments count	The number of segments in the gather list.
IN	destination	The MX Addr of the destination of the message.
IN	match_send	The matching information from the send side that will be used to find a matching receive on the remote side.
IN	context	A user-defined pointer that will be passed back to the application as part of the status structure when this request completes.
OUT	request	The pointer to the MX Request object that references the pending send operation.

If the send operation has been successfully posted, **mx_issend()** returns **MX_SUCCESS**. Otherwise, the function may return one of the following error codes.

Error return codes:

MX_BAD_SEG_COUNT	The segment list contains more than 256 segments,
MX_NO_RESOURCES	Shortage of memory or other system resources.

The arguments and return codes are identical to the previous function **mx_isend()**. The difference between **mx_issend()** and **mx_isend()** lies in the send completion semantics: a send request initiated by **mx_issend()** can be completed by a call to **mx_test()** or **mx_wait()** only if the message has been safely delivered to a matching receive request on the destination, has been cancelled, or an error has occurred.

The request will be pending, and will use resources in the MX library and in the NIC associated to the local endpoint, as long as the message is not received. Posting too many synchronous sends with **mx_issend()** when no matching receives are posted on the receive side will lead to resource exhaustion on the send side.

The data buffer(s) specified in a send operation must not be modified until the request enters the **buffered** state. This state is detected by a successful return from **mx_ibuffered()**. The segments list itself may be modified immediately after **mx_issend()** returns, just not the data buffers to which it refers.

The operation is **complete** as soon as a call to **mx_test ()** or **mx_wait()** indicates that this pending operation is complete. Note that being complete also indicates that the buffers are available for use.

In the specific case of a send request initiated by **mx_issend()**, it may be useful for the application to know when the send buffers can be reused, before the message is effectively received on the remote side and the send request is ready to be completed. Indeed, data can be buffered on the send or receive side with the synchronous send request still pending. **mx_ibuffered()** is used to check if the send request is in **buffered** state but not yet in **complete** state.

Example V.1: Post of a non-blocking non-contiguous synchronous send

```
#include "myriexpress.h"

int
main(void)
{
    mx_return_t    rc;
    mx_endpoint_t  endpoint;
    mx_endpoint_addr_t destination;
    uint64_t       nic_id;
    mx_request_t   send_handle;
    mx_segment_t   buffer_desc[2];
    uint8_t        workspace[256];
    uint64_t       match_send;
    mx_status_t    status;
    uint32_t       result;

    /* Init and open endpoint [...] */

    /* Build address of remote endpoint, hostname = remotehost,
       Endpoint ID = 6, Filter key = 0x12345678 */
    rc = mx_hostname_to_nic_id("remotehost", &nic_id);
    rc = mx_connect(endpoint, nic_id, 6, 0x12345678, MX_INFINITE, &destination);

    /* post an synchronous non-contiguous send composed of
       2 contiguous segments */
    buffer_desc[0].segment_ptr = &(workspace[64]);
    buffer_desc[0].segment_length = 17;
    buffer_desc[1].segment_ptr = &(workspace[0]);
```

```

buffer_desc[1].segment_length = 50;
match_send = 0x1111111122223333L;

rc = mx_issend(endpoint, buffer_desc, 2, destination, match_send,
              NULL, &send_handle);

/* safe to modify segment list here */

/* wait for it to be safe to change values in workspace */
do {
    rc = mx_ibuffered(endpoint, &send_handle, &result);
} while (rc == MX_SUCCESS && !result);

/* Now OK to modify data buffer, "workspace" */

/* wait for send completion. mx_wait could be used to release the
   CPU instead of looping on mx_test, see section VII.2 */
do {
    rc = mx_test(endpoint, &send_handle, &status, &result);
} while (rc == MX_SUCCESS && !result);

/* endpoint closing and finalize [...] */
}

```

V.2 Receive Operations

V.2.1 mx_irecv()

The receive operation has arguments similar to the send operations. MX provides **mx_irecv()** to post asynchronous receives.

```
mx_return_t  
mx_irecv(mx_endpoint_t endpoint,  
         mx_segment_t *segments_list,  
         uint32_t segments_count,  
         uint64_t match_rcv,  
         uint64_t match_mask,  
         void *context,  
         mx_request_t *request);
```

Parameters:

IN	endpoint	The MX Endpoint used to post the receive.
IN	segments_list	The array of contiguous segments constituting the scatter list describing the receive buffer.
IN	segments count	The number of segments in the scatter list.
IN	match_rcv	The matching information to be matched by the incoming message after masking it by the match mask.
IN	match_mask	The mask applied to the matching information of the incoming message to match the match_rcv associated to the pending receive.
IN	context	A user-defined pointer that will be passed back to the application as part of the status structure when this request completes or fails.
OUT	request	The pointer to the MX Request object that references the pending receive operation.

If the receive operation has been successfully posted, **mx_irecv()** returns **MX_SUCCESS**. Otherwise, the function may return one of the following error codes.

Error return codes:

MX_BAD_SEG_COUNT	The segment list contains more than 256 segments,
MX_BAD_MATCH_MASK	The matching information is not masked by the matching mask.
MX_NO_RESOURCES	Shortage of memory or other system resources.

The application describes the receive buffer in the same way as in the send case, using a scatter list *segments_list* composed of *segments_count* entries which are **mx_segment_t** structures. A user-defined pointer, *context*, can be associated to the receive request that will be returned in the **mx_status_t** structure when this request completes. The caller specifies *request*, a pointer to an **mx_request_t** object allocated by the application, to receive a handle by which this receive operation is referenced in future calls. The address of the remote endpoint (**mx_endpoint_addr_t**) which sends the message ultimately matched to this receive will be included in the **mx_status_t** structure to be returned when this request completes.

mx_irecv() differs from its send counterparts by specifying matching data, *match_recv* and *match_mask*. The *match_send* value of any incoming message will be first bitwise ANDed with *match_mask* and the result then compared to *match_recv*. If the values are the same, the message matches the receive and the sent data is placed in the buffer(s) associated with this receive.

Data in excess of the total buffer size provided is discarded, and the status of the receive operation will be **MX_STATUS_TRUNCATED**. The total amount of data delivered is specified in the **mx_status_t** structure returned from **mx_test()** or **mx_wait()**.

The rules for accessing data buffers are analogous to those for sending. The data in receive buffers is non-deterministic between the time the **mx_irecv()** call returns and when **mx_test()** or **mx_wait()** indicates that the receive has been completed. Writing to the buffers after the receive has been posted but before the status routine indicates completion may corrupt the receive data. As with posting a send, the segment list may be reused as soon as the call to **mx_irecv()** returns.

Inasmuch as receive requests cannot be buffered, calls to **mx_ibuffered()** do not apply for receive requests. Only **mx_test()** or **mx_wait()** are required and used to recycle the request's resources.

Example V.2: Post of an asynchronous non-contiguous receive with a context value:

```
#include "myriexpress.h"

int
main(void)
{
    mx_return_t rc;
    mx_endpoint_t endpoint;
    mx_request_t recv_handle;
    mx_segment_t buffer_desc[2];
    uint8_t workspace[256];
    uint64_t match_recv;
    uint64_t match_mask;
    mx_status_t status;
    some_private_struct my_context;
    uint32_t result;
```

```

/* Init and open local endpoint [...] */

/* post an asynchronous non-contiguous receive with a
   wildcard for the middle 16 bits of the match data (part B) */
buffer_desc[0].segment_ptr = &(workspace[64]);
buffer_desc[0].segment_length = 17;
buffer_desc[1].segment_ptr = &(workspace[0]);
buffer_desc[1].segment_length = 50;
match_rcv = UINT64_C(0x1111111100003333);
match_mask = UINT64_C(0xffffffff0000ffff);
rc = mx_irecv(endpoint, buffer_desc, 2,
              match_rcv, match_mask,
              &my_context, &recv_handle);

/* it is not yet safe to change values in workspace,
   though it is safe to modify buffer_desc */

/* wait for receive completion */
rc = mx_wait(endpoint, &recv_handle, MX_INFINITE, &status, &result);

/* status.context now holds &my_context, and
   it is now safe to write into workspace */

/* endpoint closing and finalize [...] */
}

```

VI Request State Functions

Since all communication requests within MX are non-blocking, applications must be able to check for the completion or the intermediate buffered state of these requests. **mx_ibuffered()**, **mx_test()**, and **mx_peek()** are used to check the state of requests without blocking. **mx_wait()** and **mx_peek()** are used to block, waiting for a request to complete or for the associated buffer(s) to be reusable for the application, effectively releasing the CPU for use by other threads or processes in the meantime. **mx_context()** is used to extract the *context* associated with a particular request.

VI.1 Buffered State

VI.1.1 mx_ibuffered()

The function used to check if the application can reuse the buffer(s) committed to a pending operation is **mx_ibuffered()**.

```
mx_return_t
mx_ibuffered(mx_endpoint_t endpoint,
             mx_request_t *request,
             uint32_t *result);
```

Parameters:

IN	endpoint	The MX endpoint on which the operation is pending.
IN	request	The handle of the pending request.
OUT	result	Filled in with a non-zero value if the request is buffered.

mx_ibuffered() always returns **MX_SUCCESS** in the current implementation.

The argument *request* is the handle referencing the pending MX operation. If the value returned in *result* is non-zero, the buffer(s) involved in the pending operation can be recycled by the application. Otherwise, the data is not buffered yet and the application cannot safely reuse the buffer(s).

VI.2 Request Completion

A successful return from a completion function like `mx_test()` or `mx_wait()` is required for each pending request in order to release the resources associated with the operation. If asynchronous requests are not successfully completed, the application will suffer a resource leak and MX operations will eventually fail. The usage of these functions is the only way for the application to query for the eventual success or failure of the requests.

VI.2.1 `mx_test()`

The function used to check for the completion of a pending operation in a non-blocking way is `mx_test()`:

```
mx_return_t  
mx_test(mx_endpoint_t endpoint,  
        mx_request_t *request,  
        mx_status_t *status,  
        uint32_t *result);
```

Parameters:

IN	endpoint	The MX endpoint on which the operation is pending.
IN	request	The handle to the pending request.
OUT	status	The status structure to be filled in case of completion.
OUT	result	Non-zero if the request is complete.

If the asynchronous pending request is complete, `mx_test()` returns `MX_SUCCESS`. Otherwise, the function may return one of the following error codes.

Error return codes:

<code>MX_NO_RESOURCES</code>	Shortage of memory or other system resources.
------------------------------	---

The argument *request* is the pointer to the handle referencing the pending MX operation. If the referenced operation is complete, the output parameter *result* is non-zero and the output parameter *status*, a pointer to an `mx_status_t` structure, is filled with information about the completed operation. If the request is not in the complete state, the content of the output parameter *status* is unchanged and meaningless.

The information returned to the application upon completion is organized as a structure of type `mx_status_t`, defined below:

- `mx_status_code_t` code: This code defines the nature of the completion of the operation. It can take one of these values:
 - `MX_STATUS_SUCCESS`: Operation completed successfully.
 - `MX_STATUS_PENDING`: Request still pending.

- `MX_STATUS_BUFFERED`: Request has been buffered, but still pending.
 - `MX_STATUS_REJECTED`: Filter key mismatch, message was rejected by the remote endpoint.
 - `MX_STATUS_TIMEOUT`: Posted operation timed out.
 - `MX_STATUS_TRUNCATED`: Operation completed, but received data was truncated due to undersized buffer (or oversized message).
 - `MX_STATUS_CANCELLED`: Pending operation was cancelled.
 - `MX_STATUS_ENDPOINT_UNKNOWN`: Destination endpoint is unknown on the network fabric.
 - `MX_STATUS_ENDPOINT_CLOSED`: Remote endpoint is closed.
 - `MX_STATUS_ENDPOINT_UNREACHABLE`: Connectivity is broken between the source and the destination.
 - `MX_STATUS_BAD_SESSION`: Bad session (no `mx_connect ()` done?).
 - `MX_STATUS_BAD_KEY`: Connection failed due to bad credentials.
 - `MX_STATUS_BAD_ENDPOINT`: Destination endpoint rank is out of range for the peer.
 - `MX_STATUS_BAD_RDMAWIN`: Invalid RDMA window given to the mcp.
 - `MX_STATUS_ABORTED`: Operation aborted on peer NIC.
 - `MX_STATUS_NO_RESOURCES`: Operation failed due to shortage of memory or other system resources.
- `mx_addr_t source`: This field represent the MX address of the source endpoint, from which the NIC id and the endpoint id can be extracted with `mx_decompose_endpoint_addr()`. It can be used for identification purposes or to reply to the sender.
 - `uint32_t length`: This is the effective length of the received message. It can be smaller than the length of the posted receive but not greater. If the incoming message was larger than the length of the posted receive, this length is set to the length of the posted receive and the status code returned is `MX_STATUS_TRUNCATED`.
 - `void *context`: The user-defined pointer which was passed to MX when posting the original request. It can be used to implement a callback functionality, or simply ignored.

If a *context* argument was specified when the operation was posted, this value will be in the status structure returned. To implement callbacks, *context* could be a pointer to a structure containing a callback function address and an argument that the application code would arrange to call.

VI.2.2 `mx_wait()`

It is sometimes useful to block the current thread when waiting for the completion of a pending operation. The blocked thread should not use any CPU cycles while waiting, thus yielding the processor to other threads. MX provides this capability via `mx_wait()`:

```

mx_return_t
mx_wait(mx_endpoint_t endpoint,
        mx_request_t *request,
        uint32_t timeout,
        mx_status_t *status,
        uint32_t *result);

```

Parameters:

IN	endpoint	The MX endpoint on which the operation is pending.
IN	request	The handle of the pending request.
IN	timeout	The value of the timeout in milliseconds.
OUT	status	A pointer to the status structure to be filled in case of completion, if any.
OUT	result	Non zero if the request is complete.

If the asynchronous pending request is complete, **mx_wait()** returns `MX_SUCCESS`. Otherwise, the function may return one of the following error codes.

Error return codes:

<code>MX_NO_RESOURCES</code>	Shortage of memory or other system resources.
------------------------------	---

This function blocks the current thread of execution in the kernel waiting for an interrupt from the NIC. The arguments to the **mx_wait()** functions are the same as to **mx_test()** with the addition of a timeout. This timeout is the maximum time, in milliseconds, that the function will wait for the completion of the pending request. If the request is not yet completed at the expiration of the timeout, **mx_wait()** will return to the application. If the request is completed before the expiration of the timeout, the function will return at that time and *result* will be non-zero.

VI.3 Querying for Any Completion

It may be required for the application to know if at least one request among all of the posted operations on an endpoint is ready to be completed; **mx_ipeek()** and **mx_peek()** provide this capability. These functions will return the handle of the first request on this endpoint that is ready for completion, i.e., that can be successfully processed by **mx_test()** or **mx_wait()**. If there are no requests posted on the endpoint that can be completed at the time of the call, **mx_peek()** will wait until one is ready for completion, and **mx_ipeek()** will return immediately. If several requests are eligible for completion, the particular one returned by one of the peek functions is non-deterministic.

These functions do not release any resources associated with the request; a call to **mx_test()** or **mx_wait()** is still required to release the resources.

VI.3.1 mx_ipeek()

mx_ipeek() looks for a request ready for completion on the specified endpoint and returns immediately:

```
mx_return_t  
mx_ipeek(mx_endpoint_t endpoint,  
         mx_request_t *request,  
         uint32_t *result);
```

Parameters:

IN	endpoint	The MX endpoint on which the operations are pending.
OUT	request	The handle of the completed operation, if any.
OUT	result	Non-zero if there is a request that can be completed.

If one asynchronous pending request is complete, **mx_ipeek()** returns **MX_SUCCESS**. Otherwise, the function may return one of the following error codes.

Error return codes:

MX_NO_RESOURCES	Shortage of memory or other system resources.
------------------------	---

This function looks for completion of any pending operations on a specific MX endpoint. If multiple pending requests are ready to be completed, the request returned is non-deterministic. The output parameter *request* is only valid if the output parameter *result* is non-zero.

The returned handle must be subsequently passed to **mx_test()** or **mx_wait()** in order to learn the success or failure of the request and to release the resources associated with the request. **mx_test()** is preferred over **mx_wait()** in this case as the specified request is guaranteed to be complete.

VI.3.2 mx_peek()

mx_peek() is the same as **mx_ipeek()** except that it does not return until a complete request is available or the timeout specified in the call expired.

```
mx_return_t  
mx_peek(mx_endpoint_t endpoint,  
        uint32_t timeout,  
        mx_request_t *request,  
        uint32_t *result);
```

Parameters:

IN	endpoint	The MX endpoint on which the operations are pending.
IN	timeout	The value of the timeout in milliseconds.
OUT	request	The handle of the completed operation, if any.
OUT	result	Non-zero if there is a request that can be completed.

If the asynchronous pending request is complete, **mx_peek()** returns **MX_SUCCESS**. Otherwise, the function may return one of the following error codes.

Error return codes:

MX_NO_RESOURCES	Shortage of memory or other system resources.
------------------------	---

This function blocks until at least one pending operation on a specific MX endpoint is ready for completion. If multiple pending requests are ready to be completed, the request returned is non-deterministic. The output parameter *request* is only valid if the output parameter *result* is non-zero.

The returned handle must subsequently be passed to **mx_test()** or **mx_wait()** in order to learn the success or failure of the request and to release the resources associated with the request. **mx_test()** is preferred over **mx_wait()** in this case as the specified request is guaranteed to be complete.

VI.4 Obtaining the context

Functions that generate request handles take a *context* parameter. This parameter is made available to the user when the request is completed by **mx_wait()** or **mx_test()** as part of the *status* output parameter. There can be cases, for example when handling requests returned by **mx_peek()** or **mx_peek()**, where it might be useful to extract the context field before the request is completed. **mx_context()** is the function used to obtain the *context*.

```
mx_return_t  
mx_context(mx_request_t *request,  
           void **context);
```

Parameters:

IN	request	The handle of the request from which the context is to be extracted.
OUT	context	The user-defined pointer specified when the request was created.

The current implementation of **mx_context()** always returns **MX_SUCCESS**.

VI.5 Combined Query and Completion

It is possible to combine **mx_peek()** or **mx_peek()** with the associated **mx_test()** by using **mx_test_any()** or **mx_wait_any()**. Moreover, these functions enable looking for a completed request in only a subset of requests defined by their matching information.

VI.5.1 mx_test_any()

mx_test_any() looks for a request ready for completion on the specified endpoint and returns immediately:

```
mx_return_t  
mx_test_any(mx_endpoint_t endpoint,  
            uint64_t match_info,  
            uint64_t match_mask,  
            mx_status_t *status,  
            uint32_t *result);
```

Parameters:

IN	endpoint	The MX endpoint on which the operations are pending.
IN	match_info	The matching information to be matched by the completed request after masking by the match mask.
IN	match_mask	The mask applied to the matching information of the completed request to match the match info.
OUT	status	The status structure to be filled in case of success.
OUT	result	Non-zero if there is a request that can be completed.

If one asynchronous pending request is complete, **mx_test_any()** returns **MX_SUCCESS**. Otherwise, the function may return one of the following error codes.

Error return codes:

MX_NO_RESOURCES	Shortage of memory or other system resources.
MX_BAD_MATCH_MASK	The matching information is not masked by the matching mask.

This function looks for completion of any pending operations on a specific MX endpoint that matches the *match_info* with the mask *match_mask*. If multiple pending requests are ready to be completed, the request completed is non-deterministic. The output parameter *status* is only valid if the output parameter *result* is non-zero.

The completed request is freed and all associated resources are released. No call to **mx_test()** or **mx_wait()** is required.

VI.5.2 mx_wait_any()

mx_wait_any() is the same as **mx_test_any()** except that it does not return until a complete request is available or the timeout specified in the call expired.

```
mx_return_t  
mx_wait_any(mx_endpoint_t endpoint,  
            uint32_t timeout,  
            uint64_t match_info,  
            uint64_t match_mask,  
            mx_status_t *status,  
            uint32_t *result);
```

Parameters:

IN	endpoint	The MX endpoint on which the operations are pending.
IN	timeout	The value of the timeout in milliseconds.
IN	match_info	The matching information to be matched by the completed request after masking by the match_mask.
IN	match_mask	The mask applied to the matching information of the completed request to match the match_info.
OUT	status	The status structure to be filled in case of success.
OUT	result	Non-zero if there is a request that can be completed.

If the asynchronous pending request is complete, **mx_wait_any()** returns **MX_SUCCESS**. Otherwise, the function may return one of the following error codes.

Error return codes:

MX_NO_RESOURCES	Shortage of memory or other system resources.
MX_BAD_MATCH_MASK	The matching information is not masked by the matching mask.

This function blocks until at least one pending operation on a specific MX endpoint is ready for completion and matches the *match_info* with the mask *match_mask*. If multiple matching pending requests are ready to be completed, the request completed is non-deterministic. The output parameter *status* is only valid if the output parameter *result* is non-zero.

The completed request is freed and all associated resources are released. No call to **mx_test()** or **mx_wait()** is required.

VII Probing

The functions **mx_iprobe()** and **mx_probe()** can check for incoming messages without actually receiving them. If a message is ready to be received and matches the specified matching information, the probe functions return a status structure updated with information about the message, including match data, message source, and message length.

mx_probe() blocks until a matching message is available; **mx_iprobe()** returns immediately, indicating whether a matching incoming message is available or not.

MX also provides an active message model through the **mx_register_unexp_callback()** function. It enables the execution of a user-defined callback for each unexpected message. This callback gives a last chance to the user to post a corresponding receive before the message is really moved to the unexpected queue.

VII.1.1 mx_iprobe()

```
mx_return_t
mx_iprobe(mx_endpoint_t endpoint,
          uint64_t match_rcv,
          uint64_t match_mask,
          mx_status_t *status,
          uint32_t *result);
```

Parameters:

IN	endpoint	The MX endpoint on which to probe for incoming messages.
IN	match_rcv	The matching information to be matched by the incoming message after masking it by the match mask.
IN	match_mask	The mask applied to the matching information of the incoming message to match the match_rcv argument.
OUT	status	The status structure to be filled in case of a matching incoming message is available.
OUT	result	Non-zero if there is a message ready to be received

If an incoming message matches the matching information, the status structure has been updated and **mx_iprobe()** returns **MX_SUCCESS**. Otherwise, the function may return one of the following error codes.

Error return codes:

MX_BAD_MATCH_MAS K	The matching information is not masked by the matching mask.
MX_NO_RESOURCES	Shortage of memory or other system resources.

If the output parameter *result* is non-zero, the status structure has been updated with the information related to the incoming message that matches the matching information. The incoming message is not received yet; a call to **mx_irecv()** is required to allow delivery of the message.

One current application of the probe function is to allocate the exact amount of memory needed to receive a message before receiving it.

VII.1.2 mx_probe()

mx_probe() is the blocking counterpart of **mx_iprobe()**:

```

mx_return_t
mx_probe(mx_endpoint_t endpoint,
         uint32_t timeout,
         uint64_t match_rcv,
         uint64_t match_mask,
         mx_status_t *status,
         uint32_t *result);

```

Parameters:

IN	endpoint	The MX endpoint on which to probe for incoming messages.
IN	timeout	The value of the timeout in milliseconds.
IN	match_rcv	The matching information to be matched by the incoming message after masking it by the match mask.
IN	match_mask	The mask applied to the matching information of the incoming message to match the match_rcv argument.
OUT	status	The status structure to be filled in case a matching incoming message is available.
OUT	result	Non-zero if there is a message ready to be received

If an incoming message matches the matching information, the status structure has been updated and **mx_probe()** returns MX_SUCCESS. Otherwise, the function may return one of the following error codes.

Error return codes:

MX_BAD_MATCH_MAS K	The matching information is not masked by the matching mask.
MX_NO_RESOURCES	Shortage of memory or other system resources.

If multiple threads are blocked in **mx_probe()** on the same endpoint, only one of them will return with success in case of a matching incoming message.

VII.1.3 mx_register_unexp_callback()

*This function might be deprecated by **mx_register_unexp_handler()** which provides more flexibility.*

```

mx_return_t
mx_register_unexp_callback(mx_endpoint_t endpoint,
                           mx_matching_callback_t cb,
                           void * context);

```

Parameters:

IN	endpoint	The MX endpoint on whom the callback is registered to.
IN	cb	The callback function to be called each time an unexpected message is received.
IN	context	The context to pass to the callback.

If an incoming message is unexpected, the callback is immediately called, even before any call to **mx_probe()** or **mx_iprobe()** completes. If the user posts a matching receive in the callback, the message is immediately matched. If not, the message is moved to the unexpected queue.

The **mx_matching_callback_t** type of the callback requires the user to define a function of the following prototype:

```

void
callback(void * context,
          uint64_t match_info,
          int length);

```

Parameters:

IN	context	The context that was passed to mx_register_unexp_callback() .
IN	match_info	The matching info of the unexpected incoming message that caused the callback to be called.
IN	length	The length of the incoming message.

As the callback is called on behalf of the progression, it should be as short as possible. The most usual case is to check the matching info and length of the unexpected message and post the corresponding receive. Blocking (for instance with **mx_wait()**) is allowed in the callback but should be avoided as much as possible since no progression occur during the execution of the callback.

If a matching receive is posted in the callback, it might not complete immediately since the message may have been matched on the receiver's side without actually being entirely received.

VII.1.4 mx_register_unexp_handler()

This function is only available in mx_extensions.h for now.

mx_register_unexp_callback() is somehow limited since it does not return all the informations that are available about the incoming message. **mx_register_unexp_handler()** is an extended version. Any of these functions will override the effect of the previous one if they are both used on the same endpoint.

```
mx_return_t
mx_register_unexp_handler(mx_endpoint_t endpoint,
                          mx_unexp_handler_t handler,
                          void * context);
```

Parameters:

IN	endpoint	The MX endpoint on whom the callback is registered to.
IN	handler	The callback handler to be callback each time an unexpected message is received.
IN	context	The context to pass to the callback.

The **mx_unexp_handler_t** type of the handler requires the user to define a function of the following prototype:

```
mx_unexp_handler_action_t
handler(void * context,
        mx_endpoint_addr_t source,
        uint64_t match_info,
        uint32_t length,
        void * data_if_available);
```

Parameters:

IN	context	The context that was passed to mx_register_unexp_callback() .
IN	source	The address of the sender.
IN	match_info	The matching info of the unexpected incoming message that caused the callback to be called.
IN	length	The length of the incoming message.

IN	data_if_available	A pointer to the location of the unexpected data if it has been entirely received.
----	-------------------	--

The handler must return a value to indicate what MX should do with the unexpected message afterwards. The common value is **MX_RECV_CONTINUE** which means that MX should continue to process the unexpected message, either by matching it if the application posted a receive, or by moving it to the unexpected queue.

If the application does not want to receive this unexpected message, the handler should return **MX_RECV_FINISHED** to tell MX to drop the message immediately after the handler is done.

The argument **data_if_available** is passed as non-NULL to the handler in case the message has already been received entirely. In this case, the application might choose to process the data in place instead of posting a receive and processing the data in the associated receive buffer later. Once the data has been processed in place, the handler may return **MX_RECV_FINISHED** since the application does not want to this message to be matched anymore.

data_if_available is guaranteed to be non-NULL for message whose length is less than 1kB. For larger messages, the application should be ready to process the unexpected as usual, i.e. post a receive in the handler, return **MX_RECV_CONTINUE** and get the completion later.

VIII Controlling the progression

VIII.1.1 `mx_disable_progression()` and `mx_reenable_progression()`

The unexpected handler or callback might run as any time because MX communications may be progressing in background. To avoid possible race conditions, it is sometime important to prevent the handler or callback to be called. **`mx_disable_progression()`** prevents any incoming events to be processed, any queued requests to be sent, and any callback or handler to be called. **`mx_reable_progression()`** reenables the progression as usual.

```
mx_return_t
mx_disable_progression(mx_endpoint_t endpoint);

mx_return_t
mx_reenable_progression(mx_endpoint_t endpoint);
```

Parameters:

IN	endpoint	The MX endpoint on whom the progression is disabled or reenabled.
----	----------	---

VIII.1.2 `mx_progress()`

This function is only available in `mx_extensions.h` for now.

The progression occurs either during any MX function, or while somebody is sleeping in a MX function (such as **`mx_wait()`**, **`mx_peek()`**, **`mx_probe()`** or **`mx_wait_any()`**). When the application wants to ensure that the progression occurs, it may use **`mx_progress()`**.

```
mx_return_t
mx_progress(mx_endpoint_t endpoint);
```

Parameters:

IN	endpoint	The MX endpoint on whom the progression is enforced.
----	----------	--

IX Canceling MX requests

IX.1.1 mx_cancel()

Pending receive operations may be cancelled via **mx_cancel()**. This function is required for cleanup. Posting a receive ties up user resources (receive buffers) and MX resources (in the library or in the NIC), and a cancel may be needed to free these resources gracefully.

```
mx_return_t  
mx_cancel(mx_endpoint_t endpoint,  
          mx_request_t *request,  
          uint32_t *result);
```

Parameters:

IN	endpoint	The MX endpoint on which the operation is pending.
IN	request	The pointer to the handle of the pending request.
OUT	result	Non-zero if the request was really cancelled.

mx_cancel() returns `MX_SUCCESS` when called on a receive request. Other possible return codes include:

Error return codes:

<code>MX_CANCEL_NOT_SUPPORTED</code>	Called on a request whose cancellation is not supported.
--------------------------------------	--

This function always returns immediately. If (`*result == 1`), then this request was cancelled successfully. If (`*result == 0`), then it was too late to cancel this request because the receive has already been matched. Thus, after a call to **mx_cancel()**, the request has either been cancelled (and the resources freed), or the request has been matched and a subsequent call to **mx_test()** or **mx_wait()** is guaranteed to complete quickly. In either case, **mx_cancel()** provides a way for the application to safely free receive requests.

IX.1.2 mx_forget()

This function is only available in `mx_extensions.h` for now.

Pending requests may be forgotten via **mx_forget()**. When completing, the request will be automatically freed without returning any completion event to the application. Once **mx_forget()** has returned, the request handle is invalid, the application cannot use it anymore.

```
mx_return_t
```

```
mx_forget(mx_endpoint_t endpoint,  
          mx_request_t *request);
```

Parameters:

IN	endpoint	The MX endpoint on which the operation is pending.
IN	request	The pointer to the handle of the pending request.

mx_forget() always returns `MX_SUCCESS`.

X Application Programming Notes

This section discusses important points for which application programmers writing to the Myrinet Express API should be aware.

X.1 Completing Requests

It is important to remember that every request posted must have a matching call to either **mx_test()** or **mx_wait()** to free the resources allocated for handling the request. These resources are not released until a call, made with the handle for the request, to **mx_test()** or **mx_wait()** returns successfully.

Remember also that calling **mx_cancel()** on a request only releases its resources if (**result == 1*). Otherwise, the call to **mx_test()** or **mx_wait()** is still needed to confirm the completion of the request (and the release of the resources).

X.2 Multi-threaded Applications

Thread safety in MX imposes special considerations:

If one thread is already blocked in a blocking state function, such as **mx_wait()**, for a single pending request then no other threads can block on the same handle. It is an application error to have several threads waiting on the same operation. However, it is allowed to have several threads blocking on a whole MX endpoint through calls to **mx_peek()**. In this case, a request on this endpoint reaching the complete state will awaken one of the blocked threads.

The user must not mix polling and blocking on the same handle. Concurrently calling **mx_test()** and **mx_wait()**, for example, or on the same endpoint concurrently calling **mx_peek()** and **mx_peek()** is not allowed. Such a mix would introduce race conditions and the result would be undefined. However, it is safe to poll and block on different handles or endpoints at the same time.

X.3 Stringified Errors and Statuses

All MX functions return a **mx_return_t**. All request completion routines return a **mx_status_t** structure which contains a **mx_status_code_t** describing the status of the completed request. As those values types do not have an obvious meaning, the application should not report them to the user as integer values, but translate them to strings first.

X.3.1 mx_strerror()

mx_strerror() converts a **mx_return_t** into a descriptive string:

```
char *  
mx_strerror(mx_return_t return_value);
```

X.3.2 mx_strstatus()

mx_strstatus() converts a **mx_status_code_t** into a descriptive string:

```
char *  
mx_strstatus(mx_status_code_t status_code);
```

XI Error Handling

Each MX program has an error handler (either the default one, or one explicitly given by the application). This handler is invoked each time a MX function is unable to complete successfully.

The error handler may terminate the application, or if it returns, the error code is simply passed back to the application as the return value of the MX function call.

The default error handler will print some details about the error and terminate the application. Consequently, unless the application installs a specific error handler, MX functions will always return `MX_SUCCESS`, never an error code. This is a behavior similar to the MPI default error handling. Most applications that would abort upon a fatal network error or memory exhaustion can rely on this default behavior and do not need to check the return value of MX primitives.

Applications can change the error handler with `mx_set_error_handler()`.

```
mx_error_handler_t  
mx_set_error_handler(mx_error_handler_t handler);
```

IN	handler	The error handler chosen by the application.
----	---------	--

It is the only function allowed to be called before `mx_init()`. (It would be necessary to do so if the application wanted to handle `mx_init()` errors itself). An application can change the error handler at any point in the course of the application. The `mx_set_error_handler()` function returns the previous error handler that was installed.

An application can either install its own handler (of type `mx_error_handler_t`), or it can install the predefined `MX_ERRORS_RETURN` handler. This predefined error handler does nothing and returns immediately. This is the handler to use to have all errors passed back as the return value of MX functions; the application then has the responsibility of checking the return value of MX functions and handling any error condition.

An application can restore the default error handler at any time by using `MX_ERRORS_ARE_FATAL` as the error parameter.

All MX functions return `MX_SUCCESS` when no error occurs. A list of possible errors (if a non-aborting error handler is used) is given with each function description. For compatibility with future revisions, applications should not assume that this list is exhaustive, and should always have a default case for unknown errors (`mx_strerror()` can give a string describing the error in this case).

MX behavior in the case of programming errors is undefined (examples of programming errors are passing an invalid endpoint/request or pointer to any MX functions, or calling

any MX primitive without having called **mx_init()** first, waiting for the same request twice,, etc.. Undefined behavior includes the possibility of generating an undocumented error code (with explicative text given by **mx_strerror()**), the MX implementation might use such undocumented error codes as a way to report some programmings errors that are easy to detect.

XII Disconnection Handling

XII.1.1 mx_disconnect()

This function is only available in mx_extensions.h for now.

If an application detects that a peer has died, or has closed and reopened its endpoint, the connection is no longer valid. Some requests might be pending, for instance a large send that has not been matched on the remote peer, or a medium receive that did not arrive entirely. All these resources might have to be cleaned up, but **mx_cancel()** cannot help since they are already matched or partially processed. **mx_disconnect()** has been designed to clean up all the requests associated with a peer. When the application detects that the peer is gone, while MX did not detect it (see below).

```
mx_return_t  
mx_disconnect(mx_endpoint_t endpoint,  
              mx_endpoint_addr_t address);
```

IN	endpoint	The endpoint whose connection to the peer must be shutdown.
IN	address	The mx_endpoint_addr_t of the peer to disconnect from.

Once **mx_disconnect()** returns, all pending requests associated with the peer will complete with an error status. The connection to the peer is then invalid. **mx_connect()** or **mx_ireconnect()** must be called to reenable the communications.

It is important to note that MX will internally call **mx_disconnect()** when it will detect that a remote peer is gone:

- If a request is resent to many times without being acked
- If the peer connects again to this endpoint with a different session identifier, proving that the peer endpoint has been closed and reopened in the meantime.

But, there are some cases where MX cannot detect that the peer is gone. In these cases, the application might want to disconnect by calling **mx_disconnect()** explicitly.

XII.1.2 mx_set_request_timeout()

This function is only available in mx_extensions.h for now.

Send requests are resent as long as the destination peer does not acknowledge them. When a timeout is reached (1000 seconds by default), the request completes with an error status and the remote peer is disconnected automatically. A receive request involved in a rendez-vous might face the same problem. Connection requests too.

mx_set_request_timeout() might be used to change this timeout.

```
mx_return_t  
mx_set_request_timeout(mx_endpoint_t endpoint,  
                      mx_request_t request,  
                      uint32_t seconds);
```

Parameters:

IN	endpoint	The MX endpoint on which the operation is pending.
IN	request	The request handle whose the timeout will be changed, or NULL.
IN	seconds	The number of seconds to wait before stopping resending a request.

When called with a request handle, **mx_set_request_timeout()** will change the timeout of this single request. If the request handle is NULL, the endpoint timeout is changed, and will thus be used for all future request (not the ones that already exists).

It is worth noting that reducing the timeout of a request might lead to MX detecting the disconnection of a peer earlier, making it call **mx_disconnect()** earlier, making the other request complete with an error status earlier too.

XIII Using MX in the Kernel

XIII.1 Enabling the Kernel Library

The MX API may be exported in the kernel by specifying `–enable-kernel-lib` at configure time. This option is only available on Linux and FreeBSD for now, and is only fully supported on Linux. To compile a kernel module using the MX kernel API, the include file, `myriexpress.h`, must be included after `MX_KERNEL` has been defined to `1`.

XIII.2 Kernel-specific Behavior

Due to shared resources between all processes in the kernel, the kernel library is always compiled with multithreading support.

Self-communication and shared-memory communication are not implemented in the kernel. Thus, any messages sent to the same endpoint or another endpoint on the same node will be processed by the NIC instead of being directly written to the destination memory buffer.

Communications between a user-space and a kernel endpoint on the same node are possible as long as the user-space application does not use shared-memory communications (you may for instance pass `MX_DISABLE_SHMEM=1` as an environment variable).

XIII.3 Exceptions to the generic MX API

The entire user-space MX API may be used in a kernel environment with the following exceptions:

- `mx_set_error_handler()` is not available. Errors are always returned (as if `MX_ERRORS_RETURN` has been passed).
- To avoid system resource starvation, the unexpected message queue is limited (default is 2MB) as if `MX_PARAM_UNEXP_QUEUE_MAX` has been passed to `mx_open_endpoint()`.
- Blocking functions (such as `mx_wait()`, `mx_probe()` or `mx_peek()`) are interruptible. Thus they may return before the completion and before the timeout has expired.
- The segment structure type is `mx_ksegment_t` instead of `mx_segment_t`. Its `segment_ptr` field must be filled with a kernel virtual address through the `MX_KVA_TO_U64()` macro.

XIII.4 Extended Kernel API

The MX Kernel API provides extended memory addressing features so that it is possible to send and receive data from/to memory zones whose addresses are kernel virtual addresses, user/virtual addresses, and physical addresses. Unlike the classic communication primitives (**mx_isend()**, **mx_issend()** and **mx_irecv()**) which assume that the segment pointer is a *kernel virtual address*, the extended primitives to describe the type of memory addressing that is used in segments:

```
mx_return_t
mx_isend(mx_endpoint_t endpoint,
         mx_ksegment_t *segments_list,
         uint32_t segments_count,
         mx_pin_type_t pin_type,
         mx_endpoint_addr_t destination,
         uint64_t match_send,
         void *context,
         mx_request_t *request);
```

```
mx_return_t
mx_issend(mx_endpoint_t endpoint,
          mx_ksegment_t *segments_list,
          uint32_t segments_count,
          mx_pin_type_t pin_type,
          mx_endpoint_addr_t destination,
          uint64_t match_send,
          void *context,
          mx_request_t *request);
```

```
mx_return_t
mx_irecv(mx_endpoint_t endpoint,
         mx_segment_t *segments_list,
         uint32_t segments_count,
         mx_pin_type_t pin_type,
         uint64_t match_recv,
         uint64_t match_mask,
         void *context,
         mx_request_t *request);
```

Possible values of the **pin_type** parameter:

MX_PIN_KERNEL	MX assumes that segment pointers are <i>kernel virtual addresses</i> . This is equivalent to using the classic MX API without the pin_type parameter.
MX_PIN_PHYSICAL	MX assumes that segment pointers are <i>physical addresses (valid on the processor, not DMA addresses)</i> . On Linux, such an address might for instance be retrieved by passing a struct page * to page_to_phys() .

MX_PIN_KERNEL	MX assumes that segment pointers are <i>kernel virtual addresses</i> . This is equivalent to using the classic MX API without the pin_type parameter.
MX_PIN_USER	MX assumes that segment pointers are <i>user addresses of the current process</i> .
Any pointer to an address space descriptor	MX assumes that segment pointers are <i>user addresses</i> in the process whose address space is described by the pin_type pointer. This feature is only supported on Linux. The descriptor must be a struct mm_struct .

Due to the various type of addresses that may be used in segment pointers, the MX Kernel API provides several macros to properly set the **segment_ptr** field of the **mx_ksegment_t** with those addresses:

MX_KVA_TO_U640	Converts a <i>Kernel Virtual Address</i> to fit the segment_ptr field of the mx_ksegment_t structure. This macro must be used in the generic API in the Kernel, and in the extended API with MX_PIN_KERNEL .
MX_PA_TO_U640	Converts a <i>Physical Address</i> . This macro must be used to set segment_ptr in the extended API with MX_PIN_PHYSICAL .
MX_UVA_TO_U640	Converts a <i>User Virtual Address</i> . This macro must be used to set segment_ptr in the extended API either with MX_PIN_USER or with a pointer to an address space descriptor.