The GM Message Passing System

$Date: 2001/10/18 23:55:09 $

1 Copyright Notice

Myricom, Inc.
325 N. Santa Anita Ave.
Arcadia, CA 91024

Email: info@myri.com
World Wide Web: http://www.myri.com/

Portions of this program are subject to the following copyright:

2 About This Document

This document describes the GM message passing system. The document describes the GM-1.1 API, which is both simpler to use and more powerful that the GM-1.0 API. The 1.0 API will continue to be supported by the GM libraries for the foreseeable future and GM-1.0 programs actually run significantly faster under GM-1.1 than under GM-1.0, but new programs should use the GM API as described in this document.

This document exists in the following formats:

Adobe Acrobat (`gm.pdf`)
: best suited for printing (using Adobe Acrobat Reader (`http://www.adobe.com/prodindex/acrobat/readstep.html`)). This format produces publication quality output. It can also be read online if you have installed the Adobe Acrobat Reader plug-in for your browser, but startup time can be quite large when doing so over a slow network connection.

ASCII (`gm.txt`)
: best suited for email and ASCII-only environments. All graphical figures are represented as "ASCII art".

Gnu info format (`gm.info`)
: best suited for interactive Unix online use with searching and indexing. Viewing this version of the documentation requires the Gnu '`info`' program, available as source from the Free Software Foundation's ftp server (`ftp://prep.ai.mit.edu/pub/gnu/texinfo`). All graphical figures are represented as "ASCII art".

Hypertext Markup Language (`gm_toc.html`)
: best suited for online interactive viewing with your favorite browser. All graphical figures are included as inline images.

Monolithic Hypertext Markup Language (`gm.html`)
: Second-best suited for printing if you can't print the Adobe Acrobat version.

The following typeface conventions are used in this document:

- Text like '`this`' represents user input.
- Text like `this` represents code.
- Text like *this* represents variables.
- Text like '`this`' represents files.
- Text like `this()` represents a function name, with the return type and parameters unspecified. Such references should not be interpreted to necessarily represent a function with no parameters and/or no return value. The absence of a return value or parameters is indicated only by the use of the keyword `void`, as in "`void function(void)`", which indicates that `function()` returns nothing and requires no parameter.

Numerical constants are represented in this document using the `C` language conventions.

3 GM Overview

GM is a message-based communication system for Myrinet. Like many messaging systems, GM's design objectives included low CPU overhead, portability, low latency, and high bandwidth. Additionally, GM has several distinguishing characteristics:

- GM has extremely low overhead of about 1 microsecond per packet on all architectures.
- GM can provide simultaneous memory-protected user-level OS-bypass network interface access to several user-level applications simultaneously. (On systems that do not support memory protection, such as VxWorks, no memory protection is provided.)
- GM provides reliable ordered delivery between hosts in the presence of network faults. GM will detect and retransmit lost and corrupted packets. GM will also reroute packets around network faults when alternate routes exist. Catastrophic network errors, such as crashed hosts or disconnected links, are nonfatal; the undeliverable packets are returned to the client with an error indication, although most client programs are unable to adapt in the presence of such severe errors.
- GM supports clusters of over 10,000 nodes.
- GM provides two levels of message priority to allow efficient deadlock-free bounded-memory forwarding.
- GM allows clients to send messages up to $2^{31} - 1$ bytes long, under operating systems that support sufficient amounts of DMAable memory to be allocated.
- GM automatically maps Myrinet networks.

GM is a light-weight communication layer, and as such has limitations that can be addressed by layering a heavier-weight interface over GM. Some such limitations are the following:

- GM is unable to send messages from or receive messages into nonDMAable memory.
- The GM API does not yet support any gather or scatter operations directly.

From the client's point of view, GM consists of a library, '`libgm.a`', and a header file, '`gm.h`'. All externally visible GM identifiers in these files match the regular expression '`^_*[Gg][Mm]_`' to minimize name space pollution.

Additionally, GM has other parts that system administrators need to be concerned about:

'`gm`' The GM driver provides systems services. It is called ''`gm`'' under Unix, and is the '`Myricom Myrinet Adapter`' driver implemented in ''`gm.sys`'' under Windows NT.

'`mapper`' The Myrinet mapper daemon maps the network. It is called ''`sbin/mapper`'' under Unix, and is the '`Myricom Myrinet Mapper Daemon`' service implemented in ''`gm_mapper_service.exe`'' under Windows NT.

4 Definitions

This document attaches special meaning to a few commonly used words. The meaning of each of these words in the context of this document is defined here. *In particular, please note the special meanings of the words "size" and "length." Understanding the special meaning of these terms is critical to understanding this document.*

aligned A value is said to be aligned if it is a multiple of the required GM alignment. The required GM alignment is 1 on LANai7 hardware, 4 on LANai4 hardware, and 8 on LANai5 hardware. Pointers to memory allocated by GM are always automatically aligned.

client software

client The *client software* or simply *client* is the non-GM software that uses GM to provide a reliable ordered message delivery service. It can be an application, or a higher level networking layer, such as the MPICH over GM implementation provided by Myricom.

message A *message* is an aligned array of bytes in DMAable memory.

buffer A *buffer* is a contiguous region of DMAable memory into which a message may be copied. All GM buffers must be aligned.

length The *length* of a message is the number of bytes of data that comprise the message. There is no alignment restriction on the length of any GM message.

The *length* of a receive buffer is the number of bytes that may be safely copied into the buffer.

packet A packet is an aggregation of bytes sent over the network. Packet lengths are limited to just over `gm_mtu(port)` (usually 4096 bytes) to bound the time any packet can monopolize network resource. Note that multiple packets are required to send large messages over the network, but the segmentation of messages into packets and reassembly of packets into messages is performed automatically by GM.

size The *size* of the message is any integer greater than or equal to $\log_2(length + 8)$ where *length* is the length of the message.

The *size* of a receive buffer is any positive integer *less* than or equal to $\log_2(length + 8)$ where *length* is the length of the buffer. Consequently, a buffer of size *size* must have a *length* of at least $2^{size} - 8$. A buffer having a longer length serves no useful purpose in GM, but is allowed.

The function `gm_min_size_for_length(`*length*`)` can be used to compute the minimum size for any length, and the function `gm_max_length_for_size(`*size*`)` can be used to compute the maximum length for any size.

port A *port* is a GM communication endpoint, and serves as the interface between the client software and the network.

user A human using an application that uses GM.

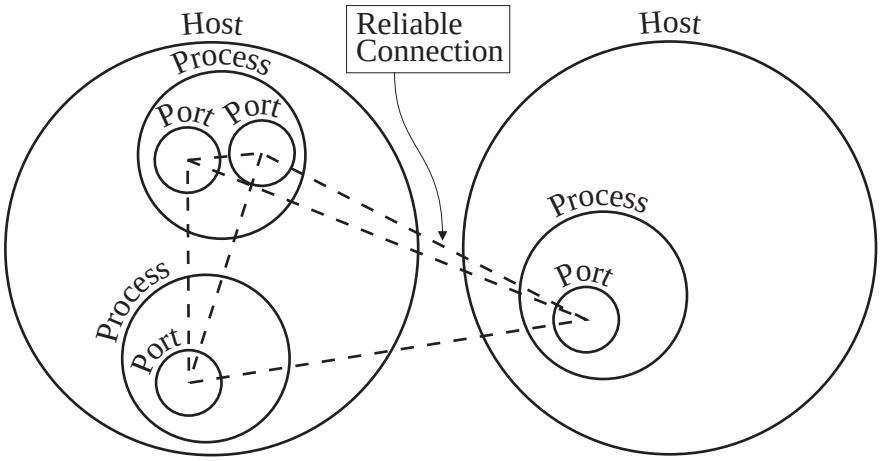
user virtual memory

Memory directly accessible by software running in a user application.

kernel virtual memory
: Memory directly accessible by the GM driver.

5 Programming Model

The GM communication system provides reliable, ordered delivery between communication endpoints, called "ports," with two levels of priority. This model is "connectionless" in that there is no need for client software to establish a connection with a remote port in order to communicate with it: the client software simply builds a message and sends it to any port in the network. (This apparently paradoxical "connectionless reliability" is achieved by GM maintaining reliable connections between each pair of hosts in the network and multiplexing the traffic between ports over these reliable connections.)



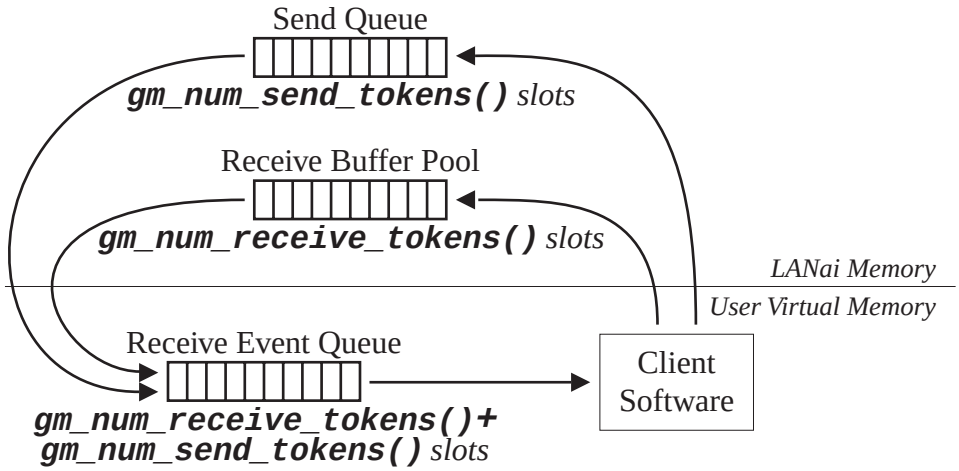
GM Endpoints (Ports)

Under operating systems that provide memory protection, GM provides memory protected network access. It should be impossible for any non-privileged GM client application to use GM to access any memory other than the application's own memory, except as explicitly allowed by the GM API. The unforgeable source of each received message is available to the receiver, allowing the receiver to discard messages from untrusted sources.

The largest message GM can send or receive is limited to $2^{31} - 1$ bytes. However, because send and receive buffers must reside in DMAable memory, the maximum message size is limited by the amount of DMAable memory the GM driver is allowed to allocate by the operating system. Most GM applications obtain DMAable memory using the straightforward `gm_dma_malloc()` and `gm_dma_free()` calls, but sophisticated applications with large memory requirements may perform DMA memory management using `gm_register_memory()` and `gm_deregister_memory()` to pin and unpin memory on operating systems that support memory registration.

Message order is preserved only for messages of the same priority, from the same sending port, and directed to the same receiving port. Messages with differing priority never block each other. Consequently, low priority messages may pass high priority messages, unlike in some other communication systems. Typical GM applications will either use only one GM priority, or use the high priority channel for control messages (such as client-to-client acks) or for single-hop message forwarding.

Both sends and receives in GM are regulated by implicit tokens, representing space allocated to the client in various internal GM queues, as depicted in the following figure. At initialization, the client implicitly possesses `gm_num_send_tokens()` send tokens, and `gm_num_receive_tokens()` receive tokens. The client may call certain functions only when possessing an implicit send or receive token, and in calling that function, the client implicitly relinquishes the token[1]. The client program is responsible for tracking the number of tokens of each type that it possesses, and must not call any GM function requiring a token when the client does not possess the appropriate token. Calling a GM API function without the required tokens has undefined results, but GM usually reports such errors, and such errors will not cause system security to be violated.

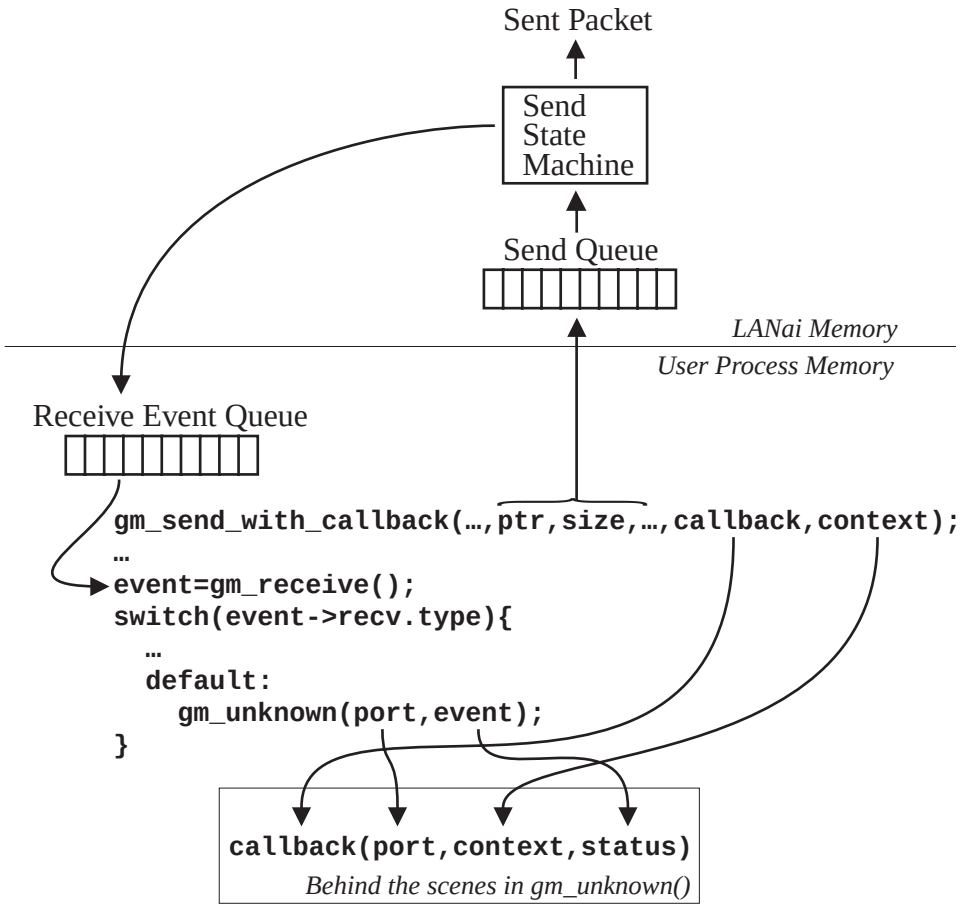


User Token Flow

As stated above, sends are token regulated. A client of a port may send a message only when it possesses a send token for that port. By calling a GM API send functions, the client implicitly relinquishes that send token. The client passes a `callback` and `context` pointer to the send function. When the send completes, GM calls `callback`, passing a pointer to the GM port, the client-supplied `context` pointer, and status code indicating if the send completed successfully or with an error. When GM calls the client's `callback` function, the send token is implicitly passed back to the client. Most GM programs, which rely on GM's fault tolerance to handle transient network faults, should consider a send completing with a status other than `GM_SUCCESS` to be a fatal error. However, more sophisticated programs may use the GM fault tolerance API extensions to handle such non-transient errors. These extensions are described in an appendix. It is important to note that the client-supplied `callback` function will be called only within a client's call to `gm_unknown()`, the GM

[1] Appendix C [Token Reference], page 42 for details.

unknown event handler function that the client must call when it receives an unrecognized event. The `gm_unknown()` function is described in more detail below.



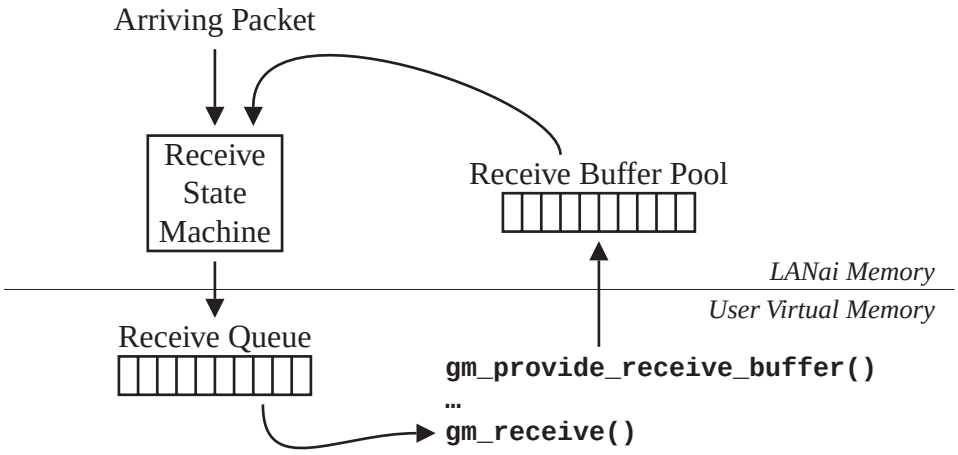
Sending

GM receives are also token regulated. After a port is opened, the client implicitly possesses `gm_num_receive_tokens()` receive tokens, allowing it to provide GM with up to this many receive buffers using `gm_provide_receive_buffer()`. With each call to `gm_provide_receive_buffer()`, the client implicitly relinquishes a receive token. With each buffer passed to `gm_provide_receive_buffer()`, the client passes a corresponding integer *size* indicating that the length of the receive buffer is at least `gm_max_length_for_size()` bytes.

Before a client of a port can receive a message of a particular size and priority, the client software must provide GM with a receive token of matching size and priority. The receive token specifies the buffer in which to store the matching receive. When a message of matching size and priority is received, that message will be transferred into the receive buffer specified in the receive token. Note that multiple receive tokens of the same size and priority **may** be provided to the port.

After providing receive buffers with sizes matching the sizes of all packets that potentially could be received, the client must poll for receive events using a `gm_*receive*()` function.

(Most developers who think polling is unacceptable in their application find that polling is fine as long as they do it in a separate thread.) The `gm_*receive*()` function will return a `gm_receive_event`. The receipt of events of type `GM_RECV_EVENT` and `GM_HIGH_RECV_EVENT` describe received packets of low and high priority, respectively. All other events should be simply passed to `gm_unknown()`. Such events are used internally by GM for sundry purposes, and the client need not be concerned with the contents of unrecognized receive events unless otherwise stated in this document.



Receiving

To avoid deadlock of the port, the client software must ensure that the port is never without a receive token for any acceptable combination of size and priority for more than a bounded amount of time, that the port is informed which combinations of size and priority are not acceptable for receives, and that the client not send to any remote port that does not do likewise.

By convention, when a port runs out of **low** priority receive tokens for any combination of sizes, the client may defer replacing the receive tokens pending the completion of a bounded number of **high** priority sends, but must always replace exhausted types of high priority receive tokens without waiting for any sends to complete. Using this technique, reliable, deadlock-free, single-hop forwarding can be achieved.

6 Initialization

Before calling any other GM function, `gm_init()` should be called. `gm_finalize()` should be called after all other GM calls and before your program exits. Each call to `gm_init()` should be balanced by a call to `gm_finalize()` before the program exits. Although GM automatically handles ungraceful program termination without such balanced calls on operating systems with memory protection, developers are strongly discouraged from relying on this feature because on some systems, such as those using the VxWorks embedded runtime system, the calls to `gm_finalize()` are required for proper shutdown of GM to allow ports to be reused without rebooting VxWorks.

A GM port is initialized by calling `gm_open(struct gm_port**`*port*`, unsigned int` *unit*`, unsigned int` *port_id*`, char *`*port_name*`, enum gm_api_version` *version*`)` to open port number *port_id* of Myrinet interface number *unit*. The pointer returned at `*`*port* must be passed to subsequent GM API calls. *port_name* is an character string of up to `gm_max_port_name_length()` bytes describing the client. The name is currently used for debugging purposes only, but this information will eventually be available to all GM clients on the network through a mechanism TBD. *version* should be `GM_API_VERSION_1_1`.

Note that while the GM API uses "`struct gm_port *`" pointers throughout, these pointers are opaque to the client. The client should not attempt to dereference these pointers.

After opening a port, the client implicitly possesses `gm_num_send_tokens()` send tokens and `gm_num_receive_tokens()` receive tokens. Most GM programs will use most or all of the `gm_num_receive_tokens()` immediately after opening a port to pass receive buffers to GM using `gm_provide_receive_buffer()`.

After the client has provided all receive buffers that it will provide during port initialization, the client should call `gm_set_acceptable_sizes()` for each priority (`GM_LOW_PRIORITY` and `GM_HIGH_PRIORITY`) to indicate what GM receive sizes the client expects to receive on the port. While this call is not strictly required, calling it allows GM to immediately reject any contradictory sends, immediately generating a send error at the sender. If these calls to `gm_set_acceptable_sizes()` are not made, then the error will not be reported until the sender experiences a GM long-period timeout, which takes about a minute to be generated by default. Therefore, calling `gm_set_acceptable_sizes()` can save much time during application development.

7 Memory Setup

GM will only send messages from memory allocated with a `gm_dma_*alloc()` function, or memory that has been registered for DMA transfers using `gm_register_memory()`. If the client attempts to send data from nonDMAable memory, GM will send bytes of value `0xaa` instead. If the client attempts to receive data into nonDMAable memory, the data will be silently discarded.

Note that some operating systems (e.g.: Solaris) do not support `gm_register_memory()` due to operating system limitations, so the `gm_dma_*alloc()` functions must be used instead to obtain DMA memory.

Unless explicitly enabled using `gm_allow_remote_memory_access(`*port*`)`, GM will not allow remote processes to use `gm_directed_send()` to modify the memory of the process. If remote memory access has been enabled, then this protection is disabled, and `any` remote GM port may modify the contents of `any` DMAable memory associated with that port. GM developers should be aware of this potential security risk, although it is usually not a concern.

8 Sending Messages

In GM, message sends are regulated by a simple token-passing mechanism to prevent GM's bounded-size internal queues from overflowing. The client software must possess a send token before calling `gm_send_with_callback()`. After initialization, the client software implicitly possesses all `gm_num_send_tokens()` send tokens, and implicitly passes one token to the GM library with each call to `gm_send_with_callback()` or `gm_send_to_peer_with_callback()`. The token is retained by GM until the send completes, at which time GM calls the client-supplied callback, implicitly returning the send token to the client. The contents of the send message should not be modified in the interval between the call to `gm_send()` and the send completion, because doing so will cause undefined data to be delivered to the receiver.

The `gm_send_with_callback()` call requires the following parameters:

port a pointer to the GM port over which the message is to be sent

message a pointer to the data to be sent

size the size receive buffer in which to store the message on the remote node

len the number of bytes to send

priority the priority with which to send the message (`GM_HIGH_PRIORITY` or `GM_LOW_PRIORITY`

target_node_id
the ID of the node to which the message should be sent

target_port_id
the ID of the GM port to which the message should be sent

callback the client function to call when the send completes

context a pointer to pass to the *callback* function when it is called

The order of messages with different priorities or with different destination ports is not preserved. Only the order of messages with the same priority and to the same destination port is preserved.

In the special case that the *target_port_id* is the same as the sending port ID (as is often the case), the streamlined `gm_send_to_peer_with_callback()` function may be used instead of `gm_send_with_callback()`, allowing the *target_port_id* parameter to be omitted, and slightly improving small-message performance on 32-bit Myrinet interfaces.

9 Receiving Messages

Similarly to message sends, message receives in GM are regulated by a simple token-passing mechanism: Before a message can be received, the client software must provide GM a receive token that allows the message to be received and specifies a buffer to hold the received data.

After initialization, the client implicitly possesses all `gm_num_receive_tokens()` receive tokens. The client software grants receive tokens to GM by calling `gm_provide_receive_buffer(`*port*`,` *buffer*`,` *size*`,` *priority*`)`, indicating that GM may receive any message into *buffer* as long as the `size` and `priority` fields of the received message exactly match the *size* and *priority* fields passed to `gm_provide_receive_buffer()`. Eventually, GM will use the buffer indicated by *message* and *size* to receive a message of the indicated *size* and *priority*. Unlike some messaging systems, GM requires that the *size* of the received message match the token size exactly. GM will *not* use the next larger sized receive buffer when a receive buffer of the correct size is not available. All receive buffers passed to `gm_provide_receive_buffer` must DMAable. They must also be aligned or be within memory allocated using `gm_dma_*alloc()` to ensure that messages can be DMAed into the buffer, and must be at least `gm_max_length_for_size(`*size*`)` bytes long.

Typical GM clients will provide at least 2 receive buffers for each size and priority of message that might be received to maximize performance by allowing one buffer to be processed and replaced while the network is filling the other. However, 1 receive buffer for each size-priority combination is sufficient for correct operation. Additionally, it is almost always a good idea to provide additional buffers for the smallest sizes, so that many small messages may be received while the host is busy computing. There is no need to provide tokens for receives smaller than `gm_min_message_size()`.

After providing receive tokens, code may poll for pending events using `gm_receive_pending(port)`, which returns a nonzero value if a receive is pending or zero if no event has been received. `gm_next_event_peek (struct gm_port *`*p*`, gm_u16_t *`*sender*`)` can also be used to peek at the event at the head of the queue. The return value is the event type (zero if no event is pending). The *sender* parameter will be filled with the sender of the message if the event is a message receive event. The client may also poll for receives using `gm_receive(`*port*`)`, which returns a pointer to a event structure of type `gm_event_t`. If no recv event is in the receive queue, a pointer to a fake receive event of `GM_NO_RECV_EVENT` will be returned. The event returned by `gm_receive()` is only guaranteed to be valid until the next call to `gm_receive()`.

There are several variants of `gm_receive()` available, all of which can safely be used in the same program.

`gm_receive()`
: returns the first pending receive event or `GM_NO_RECV_EVENT` if none is pending.

`gm_blocking_receive()`
: returns the first pending receive, blocking if necessary. This function polls for receives for 1 millisecond before sleeping, so it should generally be used only if the polling thread has a dedicated processor.

`gm_blocking_receive_no_spin()`
: returns the first pending receive, blocking if necessary. This function sleeps immediately if no receive is pending. It should be generally used in environments with more than one thread per processor.

Once the client has obtained a receive event from a `gm_*receive*()` function, the client should either process the event if the client recognizes the event, or pass the event to `gm_unknown()` if the event is unrecognized. All fields in the receive event are in network byte order, and must be converted to host byte order as specified in section See Chapter 10 [Endian Conversion], page 20.

The client is not required to handle any receive events, and may simply pass all events to `gm_unknown()`, but any useful GM program will handle `GM_RECV_EVENT`s or `GM_HIGH_RECV_EVENT`s in order to access the received data. The receive event types that the client software may choose to recognize are as follows (GM internal events are not listed):

`GM_ALARM_EVENT`
: `GM_ALARM_EVENT`s should be treated as an unknown event and passed to `gm_unknown()`. However, because client alarm handlers are called within `gm_unknown()` when `gm_unknown()` receives a `GM_ALARM_EVENT`, it can be useful for a program to perform alarm polling only after passing `GM_ALARM_EVENT`s to `gm_unknown()`, as in the '`test/gm_allsize.c`' example program. See the documentation for `gm_set_alarm()` for more information.

`GM_RECV_EVENT`
`GM_HIGH_RECV_EVENT`
: This event indicates that a normal receive has occurred. The following information is available in the `event->recv` structure.

  `length` — the number of bytes of received data

  `size` — the size of the buffer into which the message was received

  `buffer` — a pointer to the buffer passed in a call to `gm_provide_receive_buffer()`, which allowed this receive to occur

  `sender_node_id` — the GM identifier for the node that sent the message

  `sender_port_id` — the GM identifier for the port that sent the message

  `tag` — the tag passed to `gm_provide_receive_buffer_with_tag()` or 0 if `gm_provide_receive_buffer()` was used instead

  `type` — `GM_HIGH_RECV_EVENT` indicates the receipt of a high-priority packet. `GM_RECV_EVENT` indicates the receipt of a low-priority packet.

`GM_PEER_RECV_EVENT`
`GM_HIGH_PEER_RECV_EVENT`
: These events may be safely ignored (passed to `gm_unknown()`), in which case the event will be converted to a normal `GM_RECV_EVENT` and passed to the client in the next call to a `gm_*receive*()` function.

These events are just like the normal GM_RECV_EVENT and GM_HIGH_RECV_EVENT events, but indicate that the sender port id is the same as the receiver port id. Most GM programs should handle these events directly just like they handle normal receive events.

length — the number of bytes of received data

size — the size of the buffer into which the message was received

buffer — a pointer to the buffer passed in a call to gm_provide_receive_buffer(), which allowed this receive to occur

sender_node_id — the GM identifier for the node that sent the message

sender_port_id — the GM identifier for the port that sent the message

tag — the tag passed to gm_provide_receive_buffer_with_tag() or 0 if gm_provide_receive_buffer() was used instead.

type — The PEER event types indicate that the sender port number is the same as the port number. The HIGH event types indicate that the message was sent with high priority.

GM_FAST_RECV_EVENT
GM_FAST_HIGH_RECV_EVENT
GM_FAST_PEER_RECV_EVENT
GM_FAST_HIGH_PEER_RECV_EVENT

These events may be safely ignored (passed to gm_unknown()), in which case the event will be converted to a normal GM_RECV_EVENT and passed to the client in the next call to a gm_*receive*() function. The conversion process will copy the receive message from the receive queue into the receive buffer.

These types indicate that a small-message receive occurred with the small message stored in the receive queue for improved small-message performance. The PEER event types indicate that the sender port number is the same as the port number. The HIGH event types indicate that the message was sent with high priority.

If your program uses any small messages that are immediately processed and discarded upon receipt, then your program can improve performance by processing these messages directly. If after examining the message your program determines that it needs the data copied into the buffer, it can either call gm_memorize_message() to do so or can pass the event to gm_unknown().

message — a pointer to the received message, which is stored in the receive queue and is only guaranteed to be valid until the next call to gm_receive()

length — the number of bytes of received data

size — the size of the buffer into which the message was received

buffer — a pointer to the buffer passed in a call to gm_provide_receive_buffer(), which allowed this receive to occur

`sender_node_id`
: the GM identifier for the node that sent the message

`sender_port_id`
: the GM identifier for the port that sent the message

`tag`
: the tag passed to `gm_provide_receive_buffer_with_tag()` or 0 if `gm_provide_receive_buffer()` was used instead.

`type`
: The `PEER` types indicate that the sender port number is the same as the port number. The `HIGH` types indicate that the message was sent with high priority.

Note that although the receive data is in the receive queue and no receive buffer was used to store the received message, the client must have provided an appropriate receive buffer before the receive could take place, and this buffer is passed back to the client in the fast receive event. If the client needs to store the data `*message` past the next call to `gm_receive()`, then the client should copy `*message` into `*buffer` using `gm_memorize_message()`, which is simply a version of `bcopy()` optimized for copying aligned messages. After calling `gm_memorize_message()`, the fast receive event becomes equivalent to a normal receive event.

`GM_NO_EVENT`
: No event is in the event queue.

`GM_RAW_RECV_EVENT`
: This type is for internal use by the GM mapper process and will never be received by normal GM clients. It provides the following information in the `event->recv` structure:

`length`
: the number of bytes received

`buffer`
: the location of the received bytes

`GM_SENT_EVENT`
: This type indicates that one or more sends completed. **Developers using the GM-1.1 API should never see this event type**, as it is generated only if the client calls the GM-1.0 `gm_send()` function, which is deprecated in favor of the superior `gm_send_with_callback()` functions.

`event->sent.message_list` points to a null-terminated array of `void` pointers, which are message pointers from earlier `gm_send()` calls that have completed successfully. For each pointer in this array, a send token is implicitly returned to the client.

Although the number of receive events may seem daunting at first glance, almost all of the event types can be ignored. The following receive dispatch loop is fully functional for a nontrivial application that accepts messages ports, accepts only small control messages sent with high priority, and accepts low priority messages of any size:

```
{
  struct gm_port *my_port;
  gm_recv_event_t *e;
```

```
  void *some_buffer;
  ...
  while (1) {
    e = gm_receive (my_port);
    switch (gm_htohc (e->recv.type))
      {
      case GM_HIGH_RECV_EVENT:
   /* Handle high-priority control messages here in bounded time */
   gm_provide_recv_buffer (my_port,
   gm_ntohp (e->recv.buffer),
   gm_ntohc (e->recv.size),
   GM_HIGH_PRIORITY);
   break;

      case GM_RECV_EVENT:
   /* Handle data messages here in bounded time */
   gm_provide_recv_buffer (my_port, some_buffer,
   gm_ntohc (e->recv.size),
   GM_LOW_PRIORITY);
   break;

      case GM_NO_RECV_EVENT:
   /* Do bounded-time processing here, if desired. */
   break;

      default:
   gm_unknown (my_port, e);
      }
  }
}
```

However, the following implementation is slightly faster because it handles control messages without copying them into the receive buffer:

```
{
  struct gm_port *my_port;
  gm_recv_event_t *e;
  void *some_buffer;
  ...
  while (1) {
    e = gm_receive (my_port);
    switch (gm_ntohc (e->recv.type))
      {
      case GM_FAST_HIGH_PEER_RECV_EVENT:
      case GM_FAST_HIGH_RECV_EVENT:
   /* Handle high-priority control messages here in bounded time */
   gm_provide_recv_buffer (my_port,
   gm_ntohp (e->recv.buffer),
   gm_ntohc (e->recv.size),
   GM_HIGH_PRIORITY);
   break;
```

```
        case GM_FAST_PEER_RECV_EVENT:
        case GM_FAST_RECV_EVENT:
      gm_memorize_message (gm_ntohp (e->recv.buffer),
           gm_ntohp (e->recv.message),
           gm_ntohl (e->recv.length));
        case GM_PEER_RECV_EVENT:
      /* Handle data messages here in bounded time */
      gm_provide_recv_buffer (my_port, some_buffer,
      gm_ntohc (e->recv.size),
      GM_LOW_PRIORITY);
      break;

        case GM_NO_RECV_EVENT:
      /* Do bounded-time processing here, if desired. */
      break;

        default:
      gm_unknown (my_port, e);
        }
    }
  }
```

Any receive event not recognized by an application must be passed immediately to `gm_unknown()`, as in the example above. The function `gm_unknown()` will free any resources associated with the event that the client application would normally be expected to free if it recognized the type. Also, additional, undocumented event types will be received by an application and are handled by `gm_unknown()`. These messages can be used for supporting features such as GM alarms and blocking receives.

The motivation for putting small messages in the receive queue despite the fact that doing so might require a receive-side copy is the following set of observations:

- A large fraction of small receive messages are control messages that can be processed immediately upon reception, and consequently do not need to be copied into the more permanent buffer to survive calls to `gm_receive()`.
- The cost of performing an additional DMA to place the message in the buffer, rather than in the receive queue, is actually more expensive for very small messages than having the host perform the copy.

Therefore, placing small received messages in the receive command queue rather than in the more permanent receive buffer enhances performance and is worth the added complexity.

To prevent program deadlock, the client software must ensure that GM is never without a receive token (buffer) for any potentially received message for more than a bounded amount of time. Generally, except for the case of message '`forwarding`' described in the next chapter, this means that after each successful call to `gm_receive()` the client will call `gm_provide_receive_buffer()` to replace the receive token (buffer) with one of the same *size* and *priority* before the next call to `gm_receive()` or `gm_send()`. If such a deadlock condition exists for too long (on the order of a minute) or too often (a significant fraction of a one-minute interval), then remote sends directed at the receiving port will time out.

10 Endian Conversion

GM receive events are delivered to the user in network byte order. This enhances the performance of GM programs, but is a minor inconveniece to developers using the GM API. The client must call a special function to convert each field read from the `gm_recv_event_t` union to host byte order. Neglecting this conversion will result in undefined program behaviour in most cases.

In the absense of automatic checks, endian conversion is typically an error-prone programming task. Therefore, support has been added to GM-1.4 '`gm.h`' to ensure that no conversion is missing. Note, however, the support is incompatible with the deprecated `gm_send()/GM_SENT_EVENT` mechanism in GM. All you need to do to activate the checking is add the line

```
#define GM_STRONG_TYPES 1
```

before your the line

```
#include "gm.h"
```

in your source code to activate this feature[1]. Once the feature is activated, the compiler will report errors if any type conversion is missing. The error messages can be a bit cryptic and are platform specific, but they generally indicate some sort of type mismatch.

Endian conversion of fields in receive events from network to host order is achieved with the following functions:

`gm_ntohc()`
: converts 8-bit fields

`gm_ntohs()`
: converts 16-bit fields

`gm_ntohc()`
: converts 32-bit fields

`gm_ntohp()`
: converts pointer-sized fields

These 4 functions should be sufficient to convert all the types you will encounter in gm receive events.

[1] On 64-bit Solaris machines, the GM_STRONG_TYPES feature can be used during compilation to check for missing conversion, but it the resulting programs will not run and must be recompiled without this feature.

11 Alarms

GM provides the following simple alarm API. The alarm API allows the GM client to schedule a callback function to be called after a delay, specified in microseconds. An unbounded number of alarms may be set, although alarm overhead increases linearly in the number of set alarms, and the client must provide storage for each set alarm.

`void` **gm_initialize_alarm** (`gm_alarm_t *`*alarm*)
: Initialize a client-allocated `gm_alarm_t` structure for use with `gm_set_alarm()`. This function should be called after the structure is allocated but before a pointer to it is passed to `gm_set_alarm()` or `gm_cancel_alarm()`.

`void` **gm_cancel_alarm** (`gm_alarm_t *`*alarm*)
: Cancel a scheduled alarm, or do nothing if the alarm is not scheduled.

`void` **gm_set_alarm** (`struct gm_port *`*port*`, gm_alarm_t *`*alarm*`, unsigned int` *usec*`, void (*`*callback*`)(void *), void *`*context*)
: Schedule *callback*(*context*) to be called after *usec* microseconds (or later), or reschedule the alarm if it has already been scheduled and has not yet triggered. *callback* must be non-`NULL`. *context* is treated as an opaque pointer by GM, and will be passed as the single parameter to the client-supplied *callback* function.

GM clients will also be able to take advantage of the fact that an application is guaranteed to receive a single `GM_ALARM_EVENT` for each call to a client-supplied callback, with the corresponding callback occurring during the call to `gm_unknown()` that processes that alarm. This means that a case statement like the following in the client's event loops can be used to significantly reduce the overhead of polling for any effect of a client supplied alarm callback:

```
case GM_ALARM_EVENT:
  gm_unknown (event);
  /* poll for effect of alarm callbacks only here */
  break;
```

12 High Availability Extensions

While GM automatically handles transient network errors such as dropped, corrupted, or misrouted packets, and while the GM mapper automatically reconfigures the network if links or nodes appear or disappear, GM cannot automatically handle catastrophic errors such as crashed hosts or loss of network connectivity without the cooperation of the client program.

When GM detects a catastrophic error, it temporarily disables the delivery of all messages with the same sender port, target port, and priority as the message that experienced the error, and GM informs the client of catastrophic network errors by passing a status other than `GM_SUCCESS` to the client's send completion callback routine. The client program is then expected to call either `gm_resume_sending()` or `gm_drop_sends()`, which reenable the delivery of messages with the same sender port, target port, and priority. This mechanism preserves the message order over the prioritized connection between the sending and receiving ports, while allowing the client to decide if the other packets that it has already enqueued over the same connection should be transmitted or dropped.

Simpler GM programs, such as MPI programs, will typically consider GM send errors to be fatal and will typically exit when they see a send error. This is reasonable for applications running on small or physically robust clusters where errors are rare and when users can tolerate restarting jobs in the rare event of a network error. Poorly written GM programs may simply ignore the error codes, which will cause the program to eventually hang with no error indication when catastrophic errors are encountered. This poor programming practice is strongly discouraged: Developers should always check the send completion status. More sophisticated applications, such as high availability database applications, will respond to the network faults, which appear to the client as send completion status codes other than `GM_SUCCESS`.

The send completion status codes are as follows:

`GM_SUCCESS`
: The send succeeded. This status code does not indicate an error.

`GM_SEND_TIMED_OUT`
: The target port is open and responsive and the message is of an acceptable size, but the receiver failed to provide a matching receive buffer within the timeout period. This error can be caused by the receive neglecting its responsibility to provide receive buffers in a timely fashion or crashing. It can also be caused by severe congestion at the receiving node where many senders are contending for the same receive buffers on the target port for an extended period. This error indicates a programming error in the client software.

`GM_SEND_REJECTED`
: The receiver indicated (in a call to `gm_set_acceptable_sizes()`) the size of the message was unacceptable. This error indicates a programming error in the client software.

`GM_SEND_TARGET_PORT_CLOSED`
: The message cannot be delivered because the destination port has been closed.

`GM_SEND_TARGET_NODE_UNREACHABLE`
: The target node could not be reached over the Myrinet. This error can be caused by the network becoming disconnected for too long, the remote node being powered off, or by network links being rearranged when the Myrinet mapper is not running.

`GM_SEND_DROPPED`
: The send was dropped at the client's request. (The client called `gm_drop_sends()`.) This status code does not indicate an error.

`GM_SEND_PORT_CLOSED`
: Clients should never see this internal error code.

When the send completion status code indicates an error a sophisticated client program may respond by calling `gm_resume_sending()` or `gm_drop_sends()`. Calling `gm_resume_sending()` causes GM to simply reenable delivery of subsequent messages over the connection, including those that have already been enqueued. This would be the typical response of a distributed database that assumes the underlying network is unreliable and layers its own reliability protocol over GM. Calling `gm_drop_sends()` causes GM to drop all enqueued sends over the disabled connection, return them to the client with status `GM_SEND_DROPPED`, and reenable the connection. This would be the typical response of a program that wishes to reorder subsequent communication over the connection in response to the error.

Note that each of the fault response functions (`gm_drop_sends()` and `gm_resume_sending()`) requires a send token. This send token is implicitly returned to the caller when the callback function passed to `gm_drop_sends()` or `gm_resume_sending()` is called by GM.

13 Utility Modules

Some of GM's internal modules may be useful to GM developers, so their APIs are exposed. These modules include the following:

13.1 CRC Functions

GM provides the following functions, which compute 32-bit CRCs on the contents of memory. These functions are not guaranteed to perform any particular variant of the CRC-32, but these functions are useful for creating robust hashing functions.

`unsigned long` **gm_crc** (`void *`*ptr*, `unsigned long` *len*)
: computes a CRC-32 of the indicated range of memory.

`unsigned long` **gm_crc_str** (`char *`*ptr*)
: computes a CRC-32 for the indicated string.

13.2 Hash Table

13.2.1 Introduction

GM implements a generic hash table with a flexible interface. This module can automatically manage storage of fixed-size keys and/or data, or can allow the client to manage storage for keys and/or data. It allows the client to specify arbitrary hashing and comparison functions.

For example,

```
hash = gm_create_hash (gm_hash_compare_strings, gm_hash_hash_string,
                       0, 0, 0, 0);
```

creates a hash table that uses null-terminated character string keys residing in client-managed storage, and returns pointers to data in client-managed storage. In this case, all pointers to hash keys and data passed by GM to the client will be the same as the pointers passed by the client to GM.

As another example,

```
hash = gm_create_hash (gm_hash_compare_ints, gm_hash_hash_int,
                       sizeof (int), sizeof (struct my_big_struct),
                       100, 0);
```

creates a hash table that uses `ints` as keys and returns pointers to copies of the inserted structures. All storage for the keys and data is automatically managed by the hash table. In this case, all pointers to hash keys and data passed by GM to the client will point to GM-managed buffers. This function also preallocates enough storage for 100 hash entries, guaranteeing that at least 100 key/data pairs can be inserted in the table if the hash table creation succeeds.

The automatic storage management option of GM not only is convenient, but also is extremely space efficient for keys and data no larger than a pointer, because when keys and data are no larger than a pointer, GM automatically stores them in the space reserved for the pointer to the key or data, rather than allocating a separate buffer.

Note that all keys and data buffers are referred to by pointers, not by value. This allows keys and data buffers of arbitrary size to be used. As a special (but common) case, however, one may wish to use pointers as keys directly, rather than use what they point to. In this special case, use the following initialization, and pass the keys (pointers) directly to the API, rather than the usual references to the keys.

```
hash = gm_create_hash (gm_hash_compare_ptrs, gm_hash_hash_ptr,
                       0, data_len, min_cnt, flags);
```

While it is possible to specify a *key_len* of `sizeof (void *)` during initialization and treat pointer keys just like any other keys, the API above is more efficient, more convenient, and completely architecture independent.

13.2.2 Hash Table API

Some day the GM hash table API may be extended, but the current API is as follows:

`struct gm_hash *` **gm_create_hash** (`long (*`*gm_client_compare*`) (void *key1, void *key2), gm_u32_t (*`*gm_client_hash*`) (void *key1), unsigned long` *key_len*`, unsigned long` *data_len*`, unsigned long` *min_cnt*`, int` *flags*)

returns a newly-created `gm_hash` structure or `0` if the hash table could not be created. The parameters are as follows:

gm_client_compare
: the function used to compare keys and may be any of

  `gm_hash_compare_ints`

  `gm_hash_compare_longs`

  `gm_hash_compare_ptrs`

  `gm_hash_compare_strings`

  or may be a client-defined function.

gm_client_hash
: the function to be used to hash keys and may be any of

  `gm_hash_hash_int`

  `gm_hash_hash_long`

  `gm_hash_hash_ptr`

  `gm_hash_hash_string`

  or may be a client-defined function.

key_len
: specifies the length of the keys to be used for the hash table, or `0` if the keys should not be copied into GM-managed buffers.

data_len
: specifies the length of the data to be stored in the hash table, or `0` if the data should not be copied into GM-managed buffers.

min_cnt
: specifies the number of entries for which storage should be preallocated.

flags
: should be `0` because no flags are currently defined.

`void` **gm_destroy_hash** (`struct gm_hash *`*h*)

frees all resources associated with the hash table, except for any client-allocated buffers.

`void` **gm_hash_rekey** (`struct gm_hash *`*hash*`, void *`*old_key*`, void *`*new_key*)

finds each entry with key *old_key* and changes the key used to store the data to *new_key*. This call is guaranteed to succeed.

`void *` **gm_hash_remove** (`struct gm_hash *`*hash*`, void *`*key*)

removes an entry associated with *key* from the hash table *hash* and returns a pointer to the data associated with the key, or `0` if no match exists. If the data resides in a GM-managed buffer, it is only guaranteed to be valid until the next operation on the hash table.

`void *` **gm_hash_find** (`struct gm_hash *`*hash*`, void *`*key_ptr*)
: finds an entry associated with *key* from the hash table *hash* and returns a pointer to the data associated with the key, or `0` if no match exists.

`gm_status_t` **gm_hash_insert** (`struct gm_hash *`*hash*`, void *`*key*`, void *`*data*)
: stores the association of *key* and *data* in the hash table *hash*. The key `*`*key* (or data `*`*data*) is copied into the hash table unless the table was initialized with a *key_len* (or *data_len*) of 0.

13.3 Lookaside List

GM implements a lookaside list, which may be used to manage small fixed-length blocks more efficiently than `gm_malloc()` and `gm_free()`. Lookaside lists can also be used to ensure that at least a minimum number of blocks are available for allocation at all times.

GM lookaside lists have the following API:

`struct gm_lookaside *` **gm_create_lookaside** (`unsigned long` *entry_len*, `unsigned long` *min_entry_cnt*)
: returns a newly created lookaside list to be used to allocate blocks of *entry_len* bytes. *min_entry_cnt* entries are preallocated.

`void` **gm_destroy_lookaside** (`struct gm_lookaside *`*l*)
: frees a lookaside list and all associated resources, including any buffers currently allocated from the lookaside list.

`void *` **gm_lookaside_alloc** (`struct gm_lookaside *`*l*)
: returns a buffer of size *entry_len* specified when the entry list *l* was created, or `0` if the buffer could not be allocated.

`void` **gm_lookaside_free** (`void *`*ptr*)
: frees a block of memory previously allocated by a call to `gm_lookaside_alloc()`. The contents of the block of memory are guaranteed to be unchanged until the next operation is performed on the lookaside list.

13.4 Marks

13.4.1 Introduction to Marks

The GM "mark" API is new to GM-1.4. It allows the creation and destruction of mark sets, which allow mark addition, mark removal, and test for mark in mark set operations to be performed in constant time. Marks may be members of only one mark set at a time. Marks have the very unusual property that they need not be initialized before use.

All operations on marks are extremely efficient. Mark initialization requires zero time. Removing a mark from a mark set and testing for mark inclusion in a mark set take constant time. Addition of a mark to a mark set takes O(constant) time, assuming the marks set was created with support for a sufficient number of marks; otherwise, it requires O(constant) average time. Finally, creation and destruction of a mark set take time comperable to the time required for a single call to `malloc()` and `free()`, respectively.

Because marks need not be initialized before use, they can actually be used to determine if other objects have been initialized. This is done by putting a mark in the object, and adding the mark to a "mark set of marks in initialized objects" once the object has been initialized. This is similar to one common use of "magic numbers" for debugging purposes, except that it is immune to the possibility that the uninitialized magic number contained the magic number before initialization, so such marks can be used for non-debugging purposes. Therefore, marks can be used in ways that magic numbers cannot. For example, they may be used to solve the following exercise:

Exercise for the Reader:

> Given a mark set with preallocated storage for N marks and an *uninitialized*[1] block of memory large enough to hold an array of N elements and and array of N marks, write code to initialize the array, insert an element in the array, remove an element from the array, and test for presense of an element in the array, each in constant (bounded) time. No amortization of time is allowed.

Marks have a nice set of properties that each mark in a mark set has a unique value and if this value is corrupted, then the mark is implicitly removed from the mark set. This makes marks useful for detecting memory corruption, and are less prone to false negatives than are magic numbers, which proliferate copies of a single value.

Finally, marks are location-dependent. This means that if a mark is copied, the copy will not be a member of the mark set.

[1] containing unknown values

13.4.2 The Mark API

`gm_status_t` **gm_create_mark_set** (`struct gm_mark_set **`*set*`, unsigned long` *init_count*)
Return a pointer to a new mark set at *set* with enough preallocated resources to support *init_count*. Return `GM_SUCCESS` on success. Requires time comperable to `malloc()`.

`void` **gm_destroy_mark_set** (`struct gm_mark_set *`*set*)
Free all resources associated with mark set `*`*set*. Requires time comperable to `free()`.

`gm_status_t` **gm_mark** (`struct gm_mark_set *`*set*`, gm_mark_t *`*m*)
Add `*`*mark* to *set*. Requires O(constant) time if the mark set has preallocated resources for the mark. Otherwise, requires O(constant) average time.

`int` **gm_mark_is_valid** (`struct gm_mark_set *`*set*`, gm_mark_t *`*m*)
Return nonzero value if `*`*mark* is in *set*. Requires O(constant) time.

`void` **gm_unmark** (`struct gm_mark_set *`*set*`, gm_mark_t *`*m*)
Remove `*`*mark* from *set*. Requires O(constant) time.

`void` **gm_unmark_all** (`struct gm_mark_set *`*set*)
Remove all marks from *set*. Requires O(constant) time.

13.5 Page Allocation

The following GM API allows pages to be allocated and freed.

`void *` **gm_page_alloc** ()
: allocate a page-aligned buffer of length `GM_PAGE_LEN`.

`void` **gm_page_free** (`void *`*addr*)
: frees the page at *addr* previously allocated by `gm_page_alloc()`.

14 Additional Features

The function `(char *)_gm_get_route(`*port*`,` *node_id*`,` *length_p*`)` returns a pointer to the route to the network host with GM ID *node_id* and stores the length of the route in **length_p*[1].

In the future, a standard mechanism for determining the type of a remote node will be specified. It will involve sending a request to the node to return the type to avoid storing types for all remote nodes in local memory.

[1] The syntax of this call may change.

Appendix A GM Constants and Macros

GM_HIGH_PRIORITY Enum

The priority of high priority messages (1).

GM_LOW_PRIORITY Enum

The priority of low priority messages (0).

Appendix B Function Summary

The following miscellaneous library functions are provided. Several are simply cover functions for standard Unix library functions, but are provided to simplify the creation of portable GM programs, or to provide the ANSI functionality on non-ANSI systems, such as Windows NT.

`void` **gm_abort** ()
Cover function for `abort()`. Aborts the current process.

`void *` **gm_alloc_pages** (`unsigned long` *len*)
Allocate *len* bytes, aligned on a page boundary. Any fractional page following the buffer is wasted.

`int` **gm_alloc_send_token** (`struct gm_port *`*port*`, unsigned int` *priority*)
This trivial utility function allocates a send token of priority `priority` previously freed with `gm_free_send_token()` or returns `0` if no token is available. Clients may choose to maintain their own send token counts without using this utility function.

`void` **gm_allow_remote_memory_access** (`struct gm_port *`*port*)
Allow *any* remote GM port to modify the contents of *any* GM DMAable memory using the `gm_directed_send()` function. This is a significant security hole, but is very useful on tightly coupled clusters on trusted networks.

`void` **gm_bcopy** (`void *`*from*`, void *`*to*`, unsigned long` *len*)
Copy *len* bytes starting at *from* to location *to*. Does not handle overlapping regions.

`union gm_receive_event *` **gm_blocking_receive** (`struct gm_port *`*port*)
Block until there is a receive event and then return a pointer to the event. If no send is immediately available, this call suspends the current process until a receive event is available. As an optimization for applications with one CPU per CPU-intensive thread, this function polls for receives for one millisecond before sleeping the process, so it is not suited for machines running more than one performance critical process or thread on the machine.

`union gm_receive_event *` **gm_blocking_receive_no_spin** (`struct gm_port *`*port*)
This function behaves just like `gm_blocking_receive()`, only it sleeps the current thread immediately if no receive is pending. It is well suited to applications with more than one CPU-intensive thread per processor.

`void` **gm_bzero** (`void *`*ptr*`, int` *len*)
Clear the *len* bytes of memory starting at *ptr*.

`void *` **gm_calloc** (`unsigned long` *len*`, unsigned long` *cnt*)
Allocate and clear an array of *cnt* elements of length *len*.

`void` **gm_close** (`struct gm_port *`*port*)
Close a previously opened port, and free all resources associated with the port.

`void` **gm_datagram_send** (`struct gm_port *`*port*, `void *`*message*, `unsigned int` *size*, `unsigned int` *len*, `unsigned int` *priority*, `unsigned int` *target_node_id*, `unsigned int` *target_port_id*, `gm_send_completion_callback_t` *callback*, `void *`*context*)
Queue *message* of length *length* to be sent unreliably to a buffer of size *size* at *target_port_id* on *target_node_id*. *length* must be no larger than `gm_mtu(port)` If any network error is encountered while sending the packet, the packet is silently and immediately dropped. After the packet has been DMA'd from host memory, *callback*(*port*,*context*,*status*) is called inside a user invokation of `gm_unknown()`, reporting the *status* of the attempted send.

`gm_status_t` **gm_deregister_memory** (`struct gm_port *`*port*, `void *`*ptr*, `unsigned` *len*)
Deregister *len* bytes of user virtual memory starting at *ptr* that were previously registered for DMA transfers with a matching call to `gm_register_memory()`.

`void` **gm_directed_send** (`struct gm_port *`*p*, `void *`*source_buffer*, `gm_remote_ptr_t` *target_buffer*, `unsigned long` *len*, `enum gm_priority` *priority*, `unsigned int` *target_node_id*, `unsigned int` *target_port_id*)
This function is deprecated. Use `gm_directed_send_with_callback()` *instead.*
Transfer the *len* bytes at *source_buffer* to *target_port_id* on *target_node_id* with priority *priority* and store the data at the remote virtual memory address *target_buffer*. The order of the transfer is preserved relative to messages of the same priority sent using `gm_send()` or `gm_send_to_peer()`.

`void` **gm_directed_send_with_callback** (`struct gm_port *`*p*, `void *`*source_buffer*, `gm_remote_ptr_t` *target_buffer*, `unsigned long` *len*, `enum gm_priority` *priority*, `unsigned int` *target_node_id*, `unsigned int` *target_port_id*, `gm_send_completion_callback_t` *callback*, `void *`*context*)
Transfer the *len* bytes at *source_buffer* to *target_port_id* on *target_node_id* with priority *priority* and store the data at the remote virtual memory address *target_buffer*. Call *callback*(*port*,*context*,*status*) when the send completes or fails, with *status* indicating the status of the send. The order of the transfer is preserved relative to messages of the same priority sent using `gm_send()` or `gm_send_to_peer()`.

`void *` **gm_dma_calloc** (`struct gm_port *`*port*, `unsigned` *count*, `unsigned` *length*)
Allocates and clears *count***length* bytes of DMAable memory aligned on a 4-byte boundary.

`void` **gm_dma_free** (`struct gm_port *`*port*, `void *`*ptr*)
Frees *ptr*, which was allocated by a call to `gm_dma_calloc()` or `gm_dma_malloc()`. Note that the memory is not necessarily unlocked and returned to the operating system, but may be reused in future calls to `gm_dma_calloc()` and `gm_dma_malloc()`.

`void *` **gm_dma_malloc** (`struct gm_port *`*port*, `unsigned` *count*)
Allocates *length* bytes of DMAable memory aligned on a 4-byte boundary.

`void` **gm_drop_sends** (`struct gm_port *`*port*, `unsigned int` *priority*, `unsigned int` *target_node_id*, `unsigned int` *target_port_id*, `gm_send_completion_callback_t` *callback*, `void *`*context*);
Drop all enqueued sends for *port* of priority *priority* destined for *target_port_id* of *target_node_id* to be dropped, and reenable packet transmission on that connection. This function should only be called after an error is reported to a send completion callback routine, and only with parameters matching those of the failed send. It should be called only once per reported error. The dropped sends will be returned to the client with a status of `GM_SEND_DROPPED`. This function requires a send token, which is implicitly returned to the caller when the callback is called.

`void` **gm_exit** (`gm_status_t` *status*)
Cause the current process to exit with a status appropriate to the GM status code `status`.

`void` **gm_finalize** ()
Decrement the GM initialization counter, and if it becomes zero, free all resources associated with GM in the current process. Each call to `gm_finalize()` should be matched by a call to `gm_init()`.

`void` **gm_free** (`void *`*ptr*)
Free the memory buffer at *ptr*, which was previously allocated by `gm_malloc()`, or `gm_calloc()`.

`void` **gm_free_pages** (`void *`*ptr*)
Free the pages at *ptr*, which were previously allocated with `gm_alloc_pages()`.

`void` **gm_free_send_token** (`struct gm_port *`*port*, `unsigned int` *priority*)
This trivial utility function increments a count of free send tokens of `priority` for `port` so that it can later be allocated using `gm_alloc_send_token()`. Clients may choose to maintain their own count of send tokens in the client's possession instead of using this utility function.

`void` **gm_free_send_tokens** (`struct gm_port *`*port*, `unsigned int` *priority*, `unsigned int` *count*)
Like `gm_free_send_token()`, but can be used to free zero or more tokens.

`gm_status_t` **gm_get_host_name** (`struct gm_port *`*port*, `char` *name*`[GM_MAX_HOST_NAME_LEN]`)
Copy the host name of the local node to *name*.

`gm_status_t` **gm_get_node_id** (`struct gm_port *`*port*, `unsigned int *`*n*)
Copy the GM ID of the network interface card associated with *port* to `*`*n*.

`gm_status_t` **gm_get_unique_board_id** (`struct gm_port *`*port*, `char` *unique*[6])

Copy the 6-byte MAC address of the network interface card associated with *port* to the buffer at *unique*.

`void` **gm_handle_directed_send_notification** (`struct gm_port *port, gm_recv_event_t *`*event*)

Deprecated. When a `GM_DIRECTED_SEND_NOTIFICATION` event is received, programs may optionally call this function instead of `gm_unknown()` as an optimization for handling send completions more efficiently. This function does not improve performance on newer hardware, which does not generate `GM_DIRECTED_SEND_NOTIFICATION` events.

`void` **gm_handle_sent_tokens** (`struct gm_port *`*port*, `gm_recv_event_t *`*event*);

Deprecated. When a `GM_DIRECTED_SEND_NOTIFICATION` event is received, programs may optionally call this function instead of `gm_unknown()` as an optimization for handling send completions more efficiently. This function does not improve performance on newer hardware, which does not generate `GM_DIRECTED_SEND_NOTIFICATION` events.

`unsigned int` **gm_host_name_to_node_id** (`struct gm_port *`*port*, `char *`*host_name*)

Return the GM ID associated with *host_name* or `GM_NO_SUCH_NODE_ID` in case of an error.

`gm_status_t` **gm_init** ()

Increment the GM initialization counter and ianitialize GM if it was uninitialized. This call must be performed before any other GM call and before any reference to a GM global variable (e.g.: `GM_PAGE_LEN`). Each call to `gm_init()` should be matched by a call to `gm_finalize()`.[1]

`unsigned long` **gm_log2_roundup** (`unsigned long` *n*)

Returns the logarithm, base 2, of *n*, rounded up to the next integer.

`void *` **gm_malloc** (`unsigned long` *len*)

Allocates and returns a pointer to *len* bytes of uninitialized memory. The memory should be freed with `gm_free()`.

`unsigned long` **gm_max_length_for_size** (`unsigned int` *size*)

Return the maximum length of a message that will fit in a GM buffer of size *size*.

`gm_status_t` **gm_max_node_id** (`struct gm_port *port, unsigned int *n`)

Store the maximum GM node ID supported by the the network interface card corresponding to *port* at `*`*n*.

[1] Currently, `gm_open()` implicitly calls `gm_init()` for the caller and `gm_close()` implicitly calls `gm_finalize()`, but developers should not depend on this.

`gm_status_t` **gm_max_node_id_in_use** (`struct gm_port *port, unsigned int *n`)
Store the maximum in-use node ID for the network attached to *port* at **n*.

`int` **gm_memcmp** (`void *`*a*, `void *`*b*, `unsigned long` *len*)
Emulate the ANSI `memcmp()` function.

`void *` **gm_memorize_message** (`void *`*message*, `void *`*buffer*, `unsigned int` *len*)
Function for optionally optimizing the handling "`FAST`" receive messages as described in See Chapter 9 [Receiving Messages], page 14. If *message* and *buffer* differ, `gm_memorize_message(`*port*, *message*, *buffer*`)` copies the message pointed to by *message* into the buffer pointed to by *buffer*. `gm_memorize_message()` returns *buffer*. This function is optimized for performing such aligned copies.

`unsigned int` **gm_min_message_size** (`struct gm_port *`*port*)
Returns the minimum message size supported by this GM implementation.

`unsigned int` **gm_min_size_for_length** (`unsigned long` *length*)
Returns the minimum GM message buffer size required to store a message of length *length*.

`char *` **gm_node_id_to_host_name** (`struct gm_port *`*port*, `unsigned int` *node_id*)
Return a pointer to the host name of the host containing the network interface card with GM node id *node_id*. The data referenced by the returned pointer is only valid until the next GM API call.

`gm_status_t` **gm_node_id_to_unique_id** (`struct gm_port *`*port*, `unsigned int` *n*, `char` *unique*[6])
Store the MAC address for the interface with GM ID *n* at *unique*.

`unsigned int` **gm_num_ports** (`struct gm_port *`*p*)
Return the number of ports supported by the the network interface card associated with port *p*.

`unsigned int` **gm_num_receive_tokens** (`struct gm_port *`*p*)
Return the number of receive tokens implicitly owned by the port's client after the port is opened.

`unsigned int` **gm_num_send_tokens** (`struct gm_port *`*p*)
Return the number of send tokens implicitly owned by the port's client after the port is opened.

`gm_status_t` **gm_on_exit** (`gm_on_exit_callback_t` *callback*, `void *arg`)

Much like Linux `on_exit()`, this functions registers a callback so that *callback*(*status*,*arg*) is called when the program exits. Callbacks are called in the reverse of the order of registration. This function is also somewhat similar to BSD `atexit()`.

`gm_status_t` **gm_open** (`struct gm_port **`*p*, `int` *device*, `int` *port*, `char *`*port_name*)

Opens GM port *port* for LANai interface *device*, a pointer to the port's state at **p*. This pointer must be passed to all subsequent functions that operate on the opened port. *port_name* is a null-terminated ASCII string that is used to identify the port client for debugging (and potentially other) purposes; pass in the name of your program.

Note that unit and port numbers start at 0, and that ports 0 and 1 reserved, so clients will usually open ports 2 and higher.

`void *` **gm_page_alloc** ()

Returns a pointer to a newly allocated aligned uninitialized page of memory.

`gm_page_free` (**void** **addr*) void

Frees the page of memory at *addr*, an address returned by `gm_page_alloc()`.

`void` **gm_perror** (`char *`*message*, `gm_status_t` *errno*)

Similar to ANSI standard `perror`, but takes the error code as a parameter to allow thread safety in future implementations, and only supports GM error numbers. Prints *message* followed by a description of *errno*.

`int` **gm_printf** (`char *`*format*, ...)

Emulate or invoke the ANSI standard `printf()` function.

`void` **gm_provide_receive_buffer** (`struct gm_port *`*port*, `void *`*message*, `unsigned` *size*, `unsigned` *priority*)

Equivalent to calling `gm_provide_receive_buffer_with_tag(...,0)`, and no faster than doing so. It is included for backwards compatibility. Many new clients will want to use `gm_provide_receive_buffer_with_tag()` instead.

`void` **gm_provide_receive_buffer_with_tag** (`struct gm_port *`*port*, `void *`*message*, `unsigned` *size*, `unsigned` *priority*, `unsigned int` *tag*)

This function provides GM with a buffer into which it can receive messages with matching `size` and `priority` fields. It is the client software's responsibility to provide buffers of each *size* and *priority* that might be received; not doing so can cause program deadlock, which will eventually result in the port being closed after a timeout[2].

[2] For a future version of GM, an exception notification mechanism will report this exception, instead.

The client software may provide up to `gm_num_receive_tokens()` different receive buffers into which messages may be received.

Each buffer provided by the client software to GM via this function will be used only once to receive a message. In other words, calling `gm_provide_receive_buffer(`*port*`,` *buffer*`,` *size*`,` *priority*`)` provides GM a token to receive a single message of size *size* and priority *priority* into the receive buffer *buffer*. When a message is eventually received into this buffer, `gm_receive(`*port*`)` stores the buffer pointer *buffer* and *tag* in the returned event, returning control of the buffer (token) to the client software. If the client software wishes for the buffer to be reused for a similar receive, it must call `gm_provide_receive_buffer()` again with the same or similar parameters.

Once a buffer has been provided to GM, its content should not be changed until control of the buffer has been returned to the client software via `gm_receive()`.

The *tag* parameter must be in the range [0,255], and is returned in the receive event describing a receive into the buffer. It may be used in any way the client desires, and need not be unique.

`int` **gm_rand** ()

Returns a pseudo-random integer, using a poor but fast random number generator.

`int` **gm_rand_mod** (`int` *modulus*).

Returns a pseudo-random number modulo *modulus*, using a poor but fast random number generator.

`union gm_recv_event *` **gm_receive** (`struct gm_port *`*p*)

Returns a receive event. If no significant receive event is pending, then an event of type `GM_NO_RECV_EVENT` is immediately returned.

`int` **gm_receive_pending** (`struct gm_port *`*port*)

Returns nonzero if a receive event is pending. If a receive event is pending A call to any `gm_receive*)()` function will return the event immediately, although `gm_receive()` is preferred in this case for efficiency.

`int` **gm_next_event_peek** (`struct gm_port *`*p*`, gm_u16_t *`*sender*);

Returns the nonzero event type if an event is pending. If the event is a message receive event, then the sender parameter will be filled with the gmID of the message sender. If an event is pending a call to any `gm_receive*)()` function will return the event immediately, although `gm_receive()` is preferred in this case for efficiency.

`void *` **gm_register_memory** (`struct gm_port *`*port*`, void *`*ptr*`, unsigned` *len*)

Register *len* bytes of user virtual memory starting at *ptr* for DMA transfers. The memory is locked down (made nonpageable) and DMAs on the region of memory are enabled. Memory may be registered multiple times. Memory may be deregistered using matching calls to `gm_deregister_memory()`. Note that memory registration is an

expensive operation relative to sending and receiving packets, so you should use persistent memory registrations wherever possible. Also note that memory registration is not supported on Solaris due to operating system limitations.

`void` **gm_resume_sending** (`struct gm_port *`*port*, `unsigned int` *priority*, `unsigned int` *target_node_id*, `unsigned int` *target_port_id*, `gm_send_completion_callback_t` *callback*, `void *`*context*);

Reenable packet transmission of messages from *port* of priority *priority* destined for *target_port_id* of *target_node_id*. This function should only be called after an error is reported to a send completion callback routine, and only with parameters matching those of the failed send. It should be called only once per reported error. This function requires a send token, which is implicitly returned to the caller when the callback is called.

`void` **gm_send** (`struct gm_port *`*port*, `void *`*message*, `unsigned int` *size*, `unsigned int` *len*, `unsigned int` *priority*, `unsigned int` *target_node_id*, `unsigned int` *target_port_id*)

This function is *deprecated* and included only for backwards compatibility with GM-1.0.

Queues the *message* of length *len* to be sent with priority *priority* to node *target_node_id*. Before calling `gm_send()`, client software must first possess a send token of the same priority, and by calling `gm_send ()` the client implicitly relinquishes this send token. After a call to `gm_send(...,` *message*, *len*, `...)`, the memory specified by *message* and *len* must not be modified until the send completes. The buffer pointed to by *message* is not modified by GM between the time `gm_send(`*port*, *message*, `...)` is called and the time that the sent message pointer appears in a `GM_SENT_EVENT`.

`void` **gm_send_with_callback** (`struct gm_port *`*port*, `void *`*message*, `unsigned int` *size*, `unsigned int` *len*, `unsigned int` *priority*, `unsigned int` *target_node_id*, `unsigned int` *target_port_id*, `gm_send_completion_callback_t` *callback*, `void *`*context*)

Queues the *message* of length *len* to be sent with priority *priority* to node *target_node_id*. Before calling `gm_send()`, client software must first possess a send token of the same priority, and by calling `gm_send ()` the client implicitly relinquishes this send token. After a call to `gm_send(...,` *message*, *len*, `...)`, the memory specified by *message* and *len* must not be modified until the send completes. After the send completes, *callback*(*port*,*context*,*status*) will be called inside `gm_unknown()`, with *status* indicating the status of the completed send. The buffer pointed to by *message* should not be modified by GM between the time `gm_send(`*port*, *message*, `...)` is called and the time that the sent completes.

`void` **gm_send_to_peer** (`struct gm_port *`*port*, `void *`*message*, `unsigned int` *len*, `unsigned int` *priority*, `unsigned int` *target_node_id*)

This function is *deprecated* and is included only for backwards compatibility with GM-1.0.

Like `gm_send()`, only with the *target_node_id* implicitly set to the same ID as *port*. This function is marginally faster than `gm_send()`.

`void` **gm_send_to_peer_with_callback** (`struct gm_port *`*port*, `void *`*message*, `unsigned int` *len*, `unsigned int` *priority*, `unsigned int` *target_node_id*, `gm_send_completion_callback_t` *callback*, `void *`*context*)

Like `gm_send_with_callback()`, only with the *target_node_id* implicitly set to the same ID as *port*. This function is marginally faster than `gm_send_with_callback()`.

`int` **gm_send_token_available** (`struct gm_port *`*port*, `unsigned` *priority*)

Returns the value `gm_alloc_send_token(`*port*, *priority*`)` would return, without actually allocating a send token. This function allows client software to test for the availability of a send token without actually allocating the send token.

`gm_status_t` **gm_set_acceptable_sizes** (`struct gm_port *`*p*, `enum gm_priority` *priority*, `unsigned long` *mask*)

Inform GM of the acceptable sizes of GM messages received on port *p* with priority *priority*. Each set bit of *mask* represents indicates an acceptable size. While calling this function is not required, clients should call it during program initialization to errors involving the reception of badly sized messages to be reported nearly instantaneously, rather than after a substantial delay of 30 seconds or more.

`int` **gm_sleep** (`unsigned` *seconds*)

Emulate ANSI standard `sleep()`, sleeping the entire process for *seconds* seconds.

`char *` **gm_strerror** (`gm_status_t` *errno*)

Return a pointer to a constant string describing GM error code *errno*. This is somewhat similar to Unix strerror(), except *errno* must be a GM error code; it must not be a system error code.

`gm_status_t` **gm_node_id_to_unique_id** (`struct gm_port *`*port*, `unsigned int` *n*, `char` *unique*[6])

Store the MAC address for the interface with GM ID *n* at *unique*.

`void` **gm_unknown** (`struct gm_port *`*p*, `union gm_recv_event *`*e*)

This functions handles all GM events not recognized or processed by the client software, allowing the GM library and network interface card firmware to interact. This functions also catches and reports several common client program errors, and converts some unrecognizable events into recognizable form for the client.

Appendix C Token Reference

The following functions require a send tokens:

`gm_datagram_send()`

`gm_directed_send()`

`gm_directed_send_with_callback()`

`gm_drop_sends()`

`gm_resume_sending()`

`gm_send()`

`gm_send_to_peer()`

`gm_send_to_peer_with_callback()`

`gm_send_with_callback()`

The send token is implicitly returned to the client when the function's callback is called or, for the GM-1.0 functions `gm_send()` and `gm_send_to_peer()`, a send token is implicitly passed to the client with each pointer returned in a `GM_SENT_EVENT`. (The legacy `GM_SENT_EVENT`s are generated if and only if the legacy `gm_send()` and `gm_send_to_peer()` functions are called.)

The following functions require a receive token:

`gm_provide_receive_buffer()`

`gm_provide_receive_buffer_with_tag()`

A single receive token is passed to the client with each of the following events:

`GM_RAW_RECV_EVENT`

`GM_RECV_EVENT`

`GM_HIGH_RECV_EVENT`

`GM_HIGH_PEER_RECV_EVENT`

`GM_FAST_HIGH_RECV_EVENT`

`GM_FAST_HIGH_PEER_RECV_EVENT`

(However, if the client passes these events to `gm_unknown()`, then the token is implicitly returned to GM.) Any of the GM receive functions can generate these types of events. These functions are:

`gm_receive()`

`gm_blocking_receive()`

`gm_blocking_receive_no_spin()`

Appendix D Glossary

The following terms abbreviations are used in the GM documentation and source code. Some of these abbreviations are obvious to speakers of English, but are included for speakers of other languages. This section does not include architecture-specific abbreviations used in the architecture-specific GM driver code, as those are documented by the architecture's vendor and are not of interest to most GM developers.

accum accumulator

addr address

alloc allocate

arch(s) architecture(s)

buf
buff buffer

cnt count

create allocate and then initialize

destroy finalize and then free

dma direct memory access

hash hash table

hp host pointer (a pointer of the appropriate size for the host architecture in question)

insn(s) instruction(s)

intr interrupt

KVMA kernel virtual memory address

lookaside lookaside list

LSB(s) least significant byte(s)

lsb('s) least significant bit(s)

MAC Media Access Control. This is a commonly referred to sublayer of the datalink layer in the ISO network reference model.

MAC Address
A 6-byte address unique to a Myrinet interface. It is equivalent to an ethernet address.

minor device minor number

num number

phys physical

pre prefetch or precompute

PTE page table entry

recv	receive
ref	reference
seg	segment
sema	semaphore
UVMA	user virtual memory address
virt	virtual
VM	virtual memory
VMA	virtual memory address
zalloc	allocate and clear

Variable Index

G

GM_HIGH_PRIORITY 32
GM_LOW_PRIORITY 32

Concept Index

(

(void 38

C

Copyright Notice 1

G

gm_abort 33
gm_alloc_pages 33
gm_alloc_send_token 33
gm_allow_remote_memory_access 33
gm_bcopy 33
gm_blocking_receive 33
gm_blocking_receive_no_spin 33
gm_bzero 33
gm_calloc 33
gm_cancel_alarm 21
gm_close 34
gm_crc 24
gm_crc_str 24
gm_create_hash 26
gm_create_lookaside 27
gm_create_mark_set 29
gm_datagram_send 34
gm_deregister_memory 34
gm_destroy_hash 26
gm_destroy_lookaside 27
gm_destroy_mark_set 29
gm_directed_send 34
gm_directed_send_with_callback 34
gm_dma_calloc 34
gm_dma_free 34
gm_dma_malloc 35
gm_drop_sends 35
gm_exit 35
gm_finalize 35
gm_free 35
gm_free_pages 35
gm_free_send_token 35
gm_free_send_tokens 35
gm_get_host_name 35
gm_get_node_id 35
gm_get_unique_board_id 36
gm_handle_directed_send_notification 36
gm_handle_sent_tokens 36
gm_hash_find 27
gm_hash_insert 27
gm_hash_rekey 26
gm_hash_remove 26
gm_host_name_to_node_id 36
gm_init 36
gm_initialize_alarm 21
gm_log2_roundup 36
gm_lookaside_alloc 27
gm_lookaside_free 27
gm_malloc 36
gm_mark 29
gm_mark_is_valid 29
gm_max_length_for_size 36
gm_max_node_id 36
gm_max_node_id_in_use 37
gm_memcmp 37
gm_memorize_message 37
gm_min_message_size 37
gm_min_size_for_length 37
gm_next_event_peek 39
gm_node_id_to_host_name 37
gm_node_id_to_unique_id 37, 41
gm_num_ports 37
gm_num_receive_tokens 37
gm_num_send_tokens 37
gm_on_exit 38
gm_open 38
gm_page_alloc 30, 38
gm_page_free 30
gm_perror 38
gm_printf 38
gm_provide_receive_buffer 38
gm_provide_receive_buffer_with_tag 38
gm_rand 39
gm_rand_mod 39
gm_receive 39
gm_receive_pending 39
gm_register_memory 39
gm_resume_sending 40
gm_send 40
gm_send_to_peer 40
gm_send_to_peer_with_callback 41
gm_send_token_available 41
gm_send_with_callback 40
gm_set_acceptable_sizes 41
gm_set_alarm 21
gm_sleep 41
gm_strerror 41
gm_unknown 41
gm_unmark 29
gm_unmark_all 29

Table of Contents

1 Copyright Notice 1

2 About This Document 3

3 GM Overview 4

4 Definitions 5

5 Programming Model 7

6 Initialization 11

7 Memory Setup 12

8 Sending Messages 13

9 Receiving Messages 14

10 Endian Conversion 20

11 Alarms 21

12 High Availability Extensions 22

13 Utility Modules 24

13.1 CRC Functions 24
13.2 Hash Table 25
13.2.1 Introduction 25
13.2.2 Hash Table API 26
13.3 Lookaside List 27
13.4 Marks 28
13.4.1 Introduction to Marks 28
13.4.2 The Mark API 29
13.5 Page Allocation 30

14 Additional Features 31

Appendix A GM Constants and Macros 32

Appendix B Function Summary 33

Appendix C Token Reference 42

Appendix D Glossary 43

Variable Index 45

Concept Index 46