GM Reference Manual

2.0.6

Generated by Doxygen 1.2.15

Tue Sep 9 08:26:21 2003

Contents

Chapter 1

GM: A message-passing system for Myrinet networks

1.1 Table of Contents:

If difficulties are encountered, please consult the FAQ and all technical support questions should be directed to help@myri.com.

Chapter 2

GM Module Index

2.1 GM Modules

Here is a list of all modules:

Chapter 3

GM Data Structure Index

3.1 GM Data Structures

Here are the data structures with brief descriptions:

Chapter 4

GM File Index

4.1 GM File List

Here is a list of all documented files with brief descriptions:

Chapter 5

GM Page Index

5.1 GM Related Pages

Here is a list of all related documentation pages:

Chapter 6

GM Module Documentation

6.1 Deprecated GM API functions

gm_directed_send() is deprecated and included only for backward compatibility. Use gm_directed_send_with_callback() instead.

See also:
gm_directed_send_with_callback

gm_handle_sent_tokens() is deprecated and included only for backward compatibility. Use gm_unknown() instead.

See also:
gm_unknown

gm_provide_receive_buffer() is deprecated and included only for backward compatibility. Customers should instead use gm_provide_receive_buffer_with_tag(...,0).

See also:
gm_provide_receive_buffer_with_tag

gm_send() is deprecated. Use gm_send_with_callback() instead.

See also:
gm_send_with_callback

gm_send_to_peer() is deprecated and included only for backward compatibility. Use gm_send_to_peer_with_callback() instead.

See also:
gm_send_to_peer_with_callback

Chapter 7

GM Data Structure Documentation

7.1 gm_free_mdebug Union Reference

7.1.1 Detailed Description

Union to allow chaining of free mdebug records, without requiring an extra field in the mdebug record.

The documentation for this union was generated from the following file:

- gm_malloc_debug.c

7.2 gm_free_page Struct Reference

7.2.1 Detailed Description

A structure representing a free page. This structure is placed at the start of each free page, and used to link these free pages in a list.

The documentation for this struct was generated from the following file:

- gm_page_alloc.c

7.3 gm_hash Struct Reference

7.3.1 Detailed Description

The state of a GM hash table, referenced by the client only using opaque pointers.

The documentation for this struct was generated from the following file:

- gm_hash.c

7.4 gm_hash_entry Struct Reference

7.4.1 Detailed Description

A hash table entry.

The documentation for this struct was generated from the following file:

- gm_hash.c

7.5 gm_hash_segment Struct Reference

7.5.1 Detailed Description

Structure representing a segment of allocated hash table bins. To double the size of the hash table, we allocate a new segment with just enough bins to double the number of bins in the hash table an prepend it to the list of hash segments. This way, we don't have to double-buffer the hash table while growing it, and we can grow the table closer to the limits of available memory. While we sometimes have to walk the O(log(N))-segment list to find a bin, the average lookup only looks at O(2) segments, so operations are still constant-average-time as expected for hash tables.

The documentation for this struct was generated from the following file:

- gm_hash.c

7.6 gm_lookaside Struct Reference

Data Fields

- gm_lookaside::gm_lookaside_segment_list segment_list

7.6.1 Detailed Description

State of a lookaside list, which is an very fast and space efficient memory allocator for fixed-sized buffers.

7.6.2 Field Documentation

7.6.2.1 struct gm_lookaside::gm_lookaside_segment_list gm_lookaside::segment_list

List of segments. Segments with free entries are at the front.

The documentation for this struct was generated from the following file:

- gm_lookaside.c

7.7 gm_lookaside::gm_lookaside_segment_list Struct Reference

7.7.1 Detailed Description

List of segments. Segments with free entries are at the front.

The documentation for this struct was generated from the following file:

- gm_lookaside.c

7.8 gm_lookaside_segment Struct Reference

7.8.1 Detailed Description

Structure holding a group of allocated lookaside entries. It is used primarily when entries are smaller than a page to allocate a page worth of entries at a time. This is important since some kernel memory allocators (Windows) round all allocations up to a page length, but this implementation allows us to be efficient for small allocations despite this.

The documentation for this struct was generated from the following file:

- gm_lookaside.c

7.9 gm_mark_reference Union Reference

7.9.1 Detailed Description

a reference to a mark, or a free entry. The reference is used to validate the mark. That mark is considered valid if and only if (set->reference[mark->tag] == mark).

The documentation for this union was generated from the following file:

- gm_mark.c

7.10 gm_mark_set Struct Reference

7.10.1 Detailed Description

The state of a mark set, the database that tracks the location and validity of marks.

The documentation for this struct was generated from the following file:

- gm_mark.c

7.11 gm_mdebug Struct Reference

7.11.1 Detailed Description

A record holding a pointer free function that an allocation should be used to free the allocation, and the top of the call stack when the memory was allocated. This allows us to report when the wrong function is used to free the memory, and to report the call stack of the allocation in this case or if a memory leak is detected.

The documentation for this struct was generated from the following file:

- gm_malloc_debug.c

7.12 gm_on_exit_record Struct Reference

7.12.1 Detailed Description

List element storing the details of a callback that should be called upon exit.

The documentation for this struct was generated from the following file:

- gm_on_exit.c

7.13 gm_page_allocation_record Struct Reference

7.13.1 Detailed Description

A record of a large buffer allocation used to allocate a bunch of pages. (We allocate pages in blocks of many pages, in case the OS's memory allocator does not give us a page-aligned allocation, and use only the aligned pages within this large buffer.

The documentation for this struct was generated from the following file:

- gm_page_alloc.c

7.14 gm_remote_ptr_n_t Struct Reference

`#include <gm.h>`

7.14.1 Detailed Description

A pointer to memory on a (potentially) remote machine. Such pointers are always 64-bits, since we don't know if the remote pointer is 32- or 64-bits.

The documentation for this struct was generated from the following file:

- gm.h

7.15 gm_s16_n_t Struct Reference

`#include <gm.h>`

7.15.1 Detailed Description

A 16-bit signed value in network byte order.

The documentation for this struct was generated from the following file:

- gm.h

7.16 gm_s32_n_t Struct Reference

`#include <gm.h>`

7.16.1 Detailed Description

A 32-bit signed value in network byte order.

The documentation for this struct was generated from the following file:

- gm.h

7.17 gm_s64_n_t Struct Reference

`#include <gm.h>`

7.17.1 Detailed Description

A 64-bit signed value in network byte order.

The documentation for this struct was generated from the following file:

- gm.h

7.18 gm_s8_n_t Struct Reference

`#include <gm.h>`

7.18.1 Detailed Description

A 8-bit signed value in network byte order. (Silly, I know.)

The documentation for this struct was generated from the following file:

- gm.h

7.19 gm_s_e_context_t Struct Reference

7.19.1 Detailed Description

A structure used to store the state of this simple example program.

The documentation for this struct was generated from the following file:

- gm_simple_example_recv.c

7.20 gm_u16_n_t Struct Reference

`#include <gm.h>`

7.20.1 Detailed Description

A 16-bit unsigned value in network byte order.

The documentation for this struct was generated from the following file:

- gm.h

7.21 gm_u32_n_t Struct Reference

`#include <gm.h>`

7.21.1 Detailed Description

A 32-bit unsigned value in network byte order.

The documentation for this struct was generated from the following file:

- gm.h

7.22 gm_u64_n_t Struct Reference

`#include <gm.h>`

7.22.1 Detailed Description

A 64-bit unsigned value in network byte order.

The documentation for this struct was generated from the following file:

- gm.h

7.23 gm_u8_n_t Struct Reference

`#include <gm.h>`

7.23.1 Detailed Description

An byte value in network byte order. (Silly, I know.)

The documentation for this struct was generated from the following file:

- gm.h

7.24 gm_up_n_t Struct Reference

`#include <gm.h>`

7.24.1 Detailed Description

A user-space pointer in network byte order. (On systems supporting multiple pointer sizes, this is large enough to store the largest pointer size, even if the process is using smaller pointers.)

The documentation for this struct was generated from the following file:

- gm.h

7.25 gm_zone Struct Reference

7.25.1 Detailed Description

State of a zone, which is a region of memory from which one can allocate memory using the zone allocation functions.

The documentation for this struct was generated from the following file:

- gm_zone.c

7.26 gm_zone_area Struct Reference

7.26.1 Detailed Description

Zones are divided into managed buffers called ”areas”, which may either represent free buffers or buffers holding user data.

The documentation for this struct was generated from the following file:

- gm_zone.c

7.27 hash_entry Struct Reference

7.27.1 Detailed Description

An entry in a custom hash table used to track memory allocations.

The documentation for this struct was generated from the following file:

- gm_malloc_debug.c

7.28 preallocated_record_chunk Struct Reference

7.28.1 Detailed Description

Holds a groups of preallocated malloc debugging records. We allocated these records in chunks to reduce the average allocation overhead, and for storage density. We allocate 2^N-1 records at a time because many malloc() implementations use memory most efficiently for allocations just smaller than 2^M bytes long.

The documentation for this struct was generated from the following file:

- gm_malloc_debug.c

Chapter 8

GM File Documentation

8.1 gm.h File Reference

Data Structures

- struct gm_remote_ptr_n_t
- struct gm_s16_n_t
- struct gm_s32_n_t
- struct gm_s64_n_t
- struct gm_s8_n_t
- struct gm_u16_n_t
- struct gm_u32_n_t
- struct gm_u64_n_t
- struct gm_u8_n_t
- struct gm_up_n_t

Defines

- #define GM_API_VERSION_1_0 0x100
- #define GM_API_VERSION_1_1 0x101
- #define GM_API_VERSION_1_2 0x102
- #define GM_API_VERSION_1_3 0x103
- #define GM_API_VERSION_1_4 0x104
- #define GM_API_VERSION_1_5 0x105
- #define GM_API_VERSION_1_6 0x106

- #define GM_API_VERSION_2_0 0x200
- #define GM_API_VERSION_2_0_6 0x20006
- #define GM_API_VERSION GM_API_VERSION_2_0_6
- #define GM_MAX_HOST_NAME_LEN 128
- #define GM_MAX_PORT_NAME_LEN 32
- #define GM_NO_SUCH_NODE_ID 0
- #define GM_CPU_alpha 0
- #define GM_RDMA_GRANULARITY 64
- #define GM_MAX_DMA_GRANULARITY 8
- #define GM_STRUCT_CONTAINING(type, field, field_instance) ((type *)((char *)(field_instance) - GM_OFFSETOF (type, field)))
- #define GM_NUM_ELEM(ar) (sizeof (ar) / sizeof (*ar))
- #define GM_POWER_OF_TWO(n) (!((n)&((n)-1)))
- #define GM_STRUCT_CONTAINING(type, field, field_instance) ((type *)((char *)(field_instance) - GM_OFFSETOF (type, field)))
- #define GM_NUM_ELEM(ar) (sizeof (ar) / sizeof (*ar))
- #define GM_POWER_OF_TWO(n) (!((n)&((n)-1)))

Typedefs

- typedef gm_u64_t gm_remote_ptr_t
- typedef enum gm_status gm_status_t
- typedef void(* gm_send_completion_callback_t)(struct gm_port *p, void *context, gm_status_t status)
- typedef gm_u32_t gm_pid_t

Enumerations

- enum gm_status { GM_SUCCESS = 0, GM_FAILURE = 1, GM_INPUT_BUFFER_TOO_SMALL = 2, GM_OUTPUT_BUFFER_TOO_SMALL = 3, GM_TRY_AGAIN = 4, GM_BUSY = 5, GM_MEMORY_FAULT = 6, GM_INTERRUPTED = 7, GM_INVALID_PARAMETER = 8, GM_OUT_OF_MEMORY = 9, GM_INVALID_COMMAND = 10, GM_PERMISSION_DENIED = 11, GM_INTERNAL_ERROR = 12, GM_UNATTACHED = 13, GM_UNSUPPORTED_DEVICE = 14, GM_SEND_TIMED_OUT = 15, GM_SEND_REJECTED = 16, GM_SEND_TARGET_PORT_CLOSED = 17, GM_SEND_TARGET_NODE_UNREACHABLE = 18, GM_SEND_DROPPED = 19, GM_SEND_PORT_CLOSED = 20, GM_NODE_ID_NOT_YET_SET = 21, GM_STILL_SHUTTING_DOWN = 22, GM_CLONE_BUSY = 23, GM_NO_SUCH_DEVICE = 24, GM_ABORTED = 25, GM_INCOMPATIBLE_LIB_AND_DRIVER = 26, GM_UNTRANSLATED_SYSTEM_ERROR = 27, GM_ACCESS_DENIED =

28, GM_NO_DRIVER_SUPPORT = 29, GM_PTE_REF_CNT_OVERFLOW = 30, GM_NOT_SUPPORTED_IN_KERNEL = 31, GM_NOT_SUPPORTED_ON_ARCH = 32, GM_NO_MATCH = 33, GM_USER_ERROR = 34, GM_TIMED_OUT = 35, GM_DATA_CORRUPTED = 36, GM_HARDWARE_FAULT = 37, GM_SEND_ORPHANED = 38, GM_MINOR_OVERFLOW = 39, GM_PAGE_TABLE_FULL = 40, GM_UC_ERROR = 41, GM_INVALID_PORT_NUMBER = 42, GM_DEV_NOT_FOUND = 43, GM_FIRMWARE_NOT_RUNNING = 44, GM_YP_NO_MATCH = 45 }

- enum gm_priority { GM_LOW_PRIORITY = 0, GM_HIGH_PRIORITY = 1, GM_NUM_PRIORITIES = 2 }
- enum gm_recv_event_type { GM_NO_RECV_EVENT = 0, GM_SENDS_FAILED_EVENT = 1, GM_ALARM_EVENT = 2 , GM_RECV_EVENT = 11, GM_HIGH_RECV_EVENT = 12, GM_PEER_RECV_EVENT = 13, GM_HIGH_PEER_RECV_EVENT = 14, GM_FAST_RECV_EVENT = 15, GM_FAST_HIGH_RECV_EVENT = 16, GM_FAST_PEER_RECV_EVENT = 17, GM_FAST_HIGH_PEER_RECV_EVENT = 18 , GM_NEW_SENDS_FAILED_EVENT = 129 , GM_NEW_RECV_EVENT = 139 , GM_NEW_FAST_RECV_EVENT = 143 , _GM_NEW_PUT_NOTIFICATION_EVENT = 149 , GM_NUM_RECV_EVENT_TYPES }

Functions

- GM_ENTRY_POINT void gm_abort (void)
- GM_ENTRY_POINT int gm_alloc_send_token (struct gm_port *p, unsigned int priority)
- GM_ENTRY_POINT gm_status_t gm_allow_remote_memory_access (struct gm_port *p)
- GM_ENTRY_POINT void gm_bcopy (const void *from, void *to, gm_size_t len)
- GM_ENTRY_POINT union gm_recv_event * gm_blocking_receive (struct gm_port *p)
- GM_ENTRY_POINT union gm_recv_event * gm_blocking_receive_no_spin (struct gm_port *p)
- GM_ENTRY_POINT void gm_bzero (void *ptr, gm_size_t len)
- GM_ENTRY_POINT void * gm_calloc (gm_size_t len, gm_size_t cnt)
- GM_ENTRY_POINT void gm_cancel_alarm (struct gm_alarm *gm_alarm)
- GM_ENTRY_POINT void gm_close (struct gm_port *p)
- GM_ENTRY_POINT void gm_datagram_send (struct gm_port *p, void *message, unsigned int size, gm_size_t len, unsigned int priority, unsigned int target_node_id, unsigned int target_port_id, gm_send_completion_callback_t callback, void *context)

- GM_ENTRY_POINT void gm_datagram_send_4 (struct gm_port *p, gm_u32_t message, unsigned int size, gm_size_t len, unsigned int priority, unsigned int target_node_id, unsigned int target_port_id, gm_send_completion_callback_t callback, void *context)
- GM_ENTRY_POINT gm_status_t gm_deregister_memory (struct gm_port *p, void *ptr, gm_size_t length)
- GM_ENTRY_POINT void gm_directed_send_with_callback (struct gm_port *p, void *source_buffer, gm_remote_ptr_t target_buffer, gm_size_t len, enum gm_priority priority, unsigned int target_node_id, unsigned int target_port_id, gm_send_completion_callback_t callback, void *context)
- GM_ENTRY_POINT void * gm_dma_calloc (struct gm_port *p, gm_size_t count, gm_size_t length)
- GM_ENTRY_POINT void gm_dma_free (struct gm_port *p, void *addr)
- GM_ENTRY_POINT void * gm_dma_malloc (struct gm_port *p, gm_size_t length)
- GM_ENTRY_POINT void gm_flush_alarm (struct gm_port *p)
- GM_ENTRY_POINT void gm_free (void *ptr)
- GM_ENTRY_POINT void gm_free_send_token (struct gm_port *p, unsigned int priority)
- GM_ENTRY_POINT void gm_free_send_tokens (struct gm_port *p, unsigned int priority, unsigned int count)
- GM_ENTRY_POINT gm_status_t gm_get_host_name (struct gm_port *port, char name[GM_MAX_HOST_NAME_LEN])
- GM_ENTRY_POINT gm_status_t gm_get_node_type (struct gm_port *port, int *node_type)
- GM_ENTRY_POINT gm_status_t gm_get_node_id (struct gm_port *port, unsigned int *n)
- GM_ENTRY_POINT gm_status_t gm_get_unique_board_id (struct gm_port *port, char unique[6])
- GM_ENTRY_POINT gm_status_t gm_get_mapper_unique_id (struct gm_port *port, char unique[6])
- GM_ENTRY_POINT void gm_hex_dump (const void *ptr, gm_size_t len)
- GM_ENTRY_POINT unsigned int gm_host_name_to_node_id (struct gm_port *port, char *_host_name)
- GM_ENTRY_POINT void gm_initialize_alarm (struct gm_alarm *my_alarm)
- GM_ENTRY_POINT int gm_isprint (int c)
- GM_ENTRY_POINT void * gm_malloc (gm_size_t len)
- GM_ENTRY_POINT void * gm_page_alloc (void)
- GM_ENTRY_POINT void gm_page_free (void *addr)
- GM_ENTRY_POINT void * gm_alloc_pages (gm_size_t len)
- GM_ENTRY_POINT void gm_free_pages (void *addr, gm_size_t len)
- GM_ENTRY_POINT gm_size_t gm_max_length_for_size (unsigned int size)
- GM_ENTRY_POINT gm_status_t gm_max_node_id (struct gm_port *port, unsigned int *n)

- GM_ENTRY_POINT int gm_memcmp (const void *a, const void *b, gm_size_t len)
- GM_ENTRY_POINT void * gm_memorize_message (void *message, void *buffer, unsigned int len)
- GM_ENTRY_POINT unsigned int gm_min_message_size (struct gm_port *port)
- GM_ENTRY_POINT unsigned int gm_min_size_for_length (gm_size_t length)
- GM_ENTRY_POINT unsigned int gm_mtu (struct gm_port *port)
- GM_ENTRY_POINT char * gm_node_id_to_host_name (struct gm_port *port, unsigned int node_id)
- GM_ENTRY_POINT gm_status_t gm_node_id_to_unique_id (struct gm_port *port, unsigned int n, char unique[6])
- GM_ENTRY_POINT unsigned int gm_num_ports (struct gm_port *p)
- GM_ENTRY_POINT unsigned int gm_num_send_tokens (struct gm_port *p)
- GM_ENTRY_POINT unsigned int gm_num_receive_tokens (struct gm_port *p)
- GM_ENTRY_POINT unsigned int gm_get_port_id (struct gm_port *p)
- GM_ENTRY_POINT gm_status_t gm_open (struct gm_port **p, unsigned int unit, unsigned int port, const char *port_name, enum gm_api_version version)
- GM_ENTRY_POINT void gm_provide_receive_buffer_with_tag (struct gm_port *p, void *ptr, unsigned int size, unsigned int priority, unsigned int tag)
- GM_ENTRY_POINT int gm_receive_pending (struct gm_port *p)
- GM_ENTRY_POINT int gm_next_event_peek (struct gm_port *p, gm_u16_t *sender)
- GM_ENTRY_POINT union gm_recv_event * gm_receive (struct gm_port *p)
- GM_ENTRY_POINT gm_status_t gm_register_memory (struct gm_port *p, void *ptr, gm_size_t length)
- GM_ENTRY_POINT int gm_send_token_available (struct gm_port *p, unsigned int priority)
- GM_ENTRY_POINT void gm_send_with_callback (struct gm_port *p, void *message, unsigned int size, gm_size_t len, unsigned int priority, unsigned int target_node_id, unsigned int target_port_id, gm_send_completion_callback_t callback, void *context)
- GM_ENTRY_POINT void gm_send_to_peer_with_callback (struct gm_port *p, void *message, unsigned int size, gm_size_t len, unsigned int priority, unsigned int target_node_id, gm_send_completion_callback_t callback, void *context)
- GM_ENTRY_POINT gm_status_t gm_set_acceptable_sizes (struct gm_port *p, enum gm_priority priority, gm_size_t mask)
- GM_ENTRY_POINT void gm_set_alarm (struct gm_port *p, struct gm_alarm *my_alarm, gm_u64_t usecs, void(*callback)(void *), void *context)
- GM_ENTRY_POINT gm_size_t gm_strlen (const char *cptr)
- GM_ENTRY_POINT char * gm_strncpy (char *to, const char *from, int len)
- GM_ENTRY_POINT int gm_strcmp (const char *a, const char *b)
- GM_ENTRY_POINT int gm_strncmp (const char *a, const char *b, int len)
- GM_ENTRY_POINT int gm_strncasecmp (const char *a, const char *b, int len)
- GM_ENTRY_POINT gm_u64_t gm_ticks (struct gm_port *port)

- GM_ENTRY_POINT gm_status_t gm_unique_id (struct gm_port *port, char unique[6])
- GM_ENTRY_POINT gm_status_t gm_unique_id_to_node_id (struct gm_port *port, char unique[6], unsigned int *node_id)
- GM_ENTRY_POINT void gm_unknown (struct gm_port *p, union gm_recv_event *e)
- GM_ENTRY_POINT gm_status_t _gm_get_route (struct gm_port *p, unsigned int node_id, char *route, unsigned int *len)
- GM_ENTRY_POINT void gm_dump_buffers (void)
- GM_ENTRY_POINT void gm_register_buffer (void *addr, int size)
- GM_ENTRY_POINT int gm_unregister_buffer (void *addr, int size)
- GM_ENTRY_POINT struct gm_lookaside * gm_create_lookaside (gm_size_t entry_len, gm_size_t min_entry_cnt)
- GM_ENTRY_POINT void gm_destroy_lookaside (struct gm_lookaside *l)
- GM_ENTRY_POINT void * gm_lookaside_alloc (struct gm_lookaside *l)
- GM_ENTRY_POINT void * gm_lookaside_zalloc (struct gm_lookaside *l)
- GM_ENTRY_POINT void gm_lookaside_free (void *ptr)
- GM_ENTRY_POINT struct gm_hash * gm_create_hash (long(*gm_user_compare)(void *key1, void *key2), unsigned long(*gm_user_hash)(void *key1), gm_size_t key_len, gm_size_t data_len, gm_size_t gm_min_entries, int flags)
- GM_ENTRY_POINT void gm_destroy_hash (struct gm_hash *h)
- GM_ENTRY_POINT void * gm_hash_remove (struct gm_hash *hash, void *key)
- GM_ENTRY_POINT void * gm_hash_find (struct gm_hash *hash, void *key)
- GM_ENTRY_POINT gm_status_t gm_hash_insert (struct gm_hash *hash, void *key, void *datum)
- GM_ENTRY_POINT void gm_hash_rekey (struct gm_hash *hash, void *old_key, void *new_key)
- GM_ENTRY_POINT long gm_hash_compare_strings (void *key1, void *key2)
- GM_ENTRY_POINT unsigned long gm_hash_hash_string (void *key)
- GM_ENTRY_POINT long gm_hash_compare_longs (void *key1, void *key2)
- GM_ENTRY_POINT unsigned long gm_hash_hash_long (void *key)
- GM_ENTRY_POINT long gm_hash_compare_ints (void *key1, void *key2)
- GM_ENTRY_POINT unsigned long gm_hash_hash_int (void *key)
- GM_ENTRY_POINT long gm_hash_compare_ptrs (void *key1, void *key2)
- GM_ENTRY_POINT unsigned long gm_hash_hash_ptr (void *key)
- GM_ENTRY_POINT unsigned long gm_crc (void *ptr, gm_size_t len)
- GM_ENTRY_POINT unsigned long gm_crc_str (const char *ptr)
- GM_ENTRY_POINT int gm_rand (void)
- GM_ENTRY_POINT void gm_srand (int seed)
- GM_ENTRY_POINT unsigned int gm_rand_mod (unsigned int modulus)
- GM_ENTRY_POINT gm_status_t gm_init (void)
- GM_ENTRY_POINT void gm_finalize (void)
- GM_ENTRY_POINT unsigned long gm_log2_roundup (unsigned long n)

- GM_ENTRY_POINT struct gm_mutex * gm_create_mutex (void)
- GM_ENTRY_POINT void gm_destroy_mutex (struct gm_mutex *mu)
- GM_ENTRY_POINT void gm_mutex_enter (struct gm_mutex *mu)
- GM_ENTRY_POINT void gm_mutex_exit (struct gm_mutex *mu)
- GM_ENTRY_POINT struct gm_zone * gm_zone_create_zone (void *base, gm_size_t len)
- GM_ENTRY_POINT void gm_zone_destroy_zone (struct gm_zone *zone)
- GM_ENTRY_POINT void * gm_zone_free (struct gm_zone *zone, void *a)
- GM_ENTRY_POINT void * gm_zone_malloc (struct gm_zone *zone, gm_size_t length)
- GM_ENTRY_POINT void * gm_zone_calloc (struct gm_zone *zone, gm_size_t count, gm_size_t length)
- GM_ENTRY_POINT int gm_zone_addr_in_zone (struct gm_zone *zone, void *p)
- GM_ENTRY_POINT void gm_resume_sending (struct gm_port *p, unsigned int priority, unsigned int target_node_id, unsigned int target_port_id, gm_send_completion_callback_t callback, void *context)
- GM_ENTRY_POINT 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)
- GM_ENTRY_POINT gm_pid_t gm_getpid (void)
- GM_ENTRY_POINT gm_status_t gm_directcopy_get (struct gm_port *p, void *source_addr, void *target_addr, gm_size_t length, unsigned int source_instance_id, unsigned int source_port_id)
- GM_ENTRY_POINT void gm_perror (const char *message, gm_status_t error)
- GM_ENTRY_POINT int gm_sleep (unsigned seconds)
- GM_ENTRY_POINT void gm_exit (gm_status_t status)
- GM_ENTRY_POINT int gm_printf (const char *format,...)
- GM_ENTRY_POINT char * gm_strerror (gm_status_t error)
- GM_ENTRY_POINT gm_status_t gm_set_enable_nack_down (struct gm_port *port, int flag)
- GM_ENTRY_POINT gm_status_t gm_max_node_id_in_use (struct gm_port *port, unsigned int *n)
- GM_ENTRY_POINT int gm_eprintf (const char *format,...)
- GM_ENTRY_POINT void * gm_memset (void *s, int c, gm_size_t n)
- GM_ENTRY_POINT char * gm_strdup (const char *)
- GM_ENTRY_POINT gm_status_t gm_mark (struct gm_mark_set *set, gm_mark_t *m)
- GM_ENTRY_POINT int gm_mark_is_valid (struct gm_mark_set *set, gm_mark_t *m)
- GM_ENTRY_POINT gm_status_t gm_create_mark_set (struct gm_mark_set **set, unsigned long init_count)
- GM_ENTRY_POINT void gm_destroy_mark_set (struct gm_mark_set *set)
- GM_ENTRY_POINT void gm_unmark (struct gm_mark_set *set, gm_mark_t *m)

- GM_ENTRY_POINT void gm_unmark_all (struct gm_mark_set *set)
- GM_ENTRY_POINT gm_status_t gm_on_exit (gm_on_exit_callback_t, void *arg)
- GM_ENTRY_POINT void gm_put (struct gm_port *p, void *local_buffer, gm_remote_ptr_t remote_buffer, gm_size_t len, enum gm_priority priority, unsigned int target_node_id, unsigned int target_port_id, gm_send_completion_callback_t callback, void *context)
- GM_ENTRY_POINT gm_status_t gm_global_id_to_node_id (struct gm_port *port, unsigned int global_id, unsigned int *node_id)
- GM_ENTRY_POINT gm_status_t gm_node_id_to_global_id (struct gm_port *port, unsigned int node_id, unsigned int *global_id)
- GM_ENTRY_POINT gm_status_t gm_node_id_to_host_name_ex (struct gm_port *port, unsigned int timeout_usecs, unsigned int node_id, char(*name)[GM_MAX_HOST_NAME_LEN+1])
- GM_ENTRY_POINT gm_status_t gm_host_name_to_node_id_ex (struct gm_port *port, unsigned int timeout_usecs, const char *host_name, unsigned int *node_id)

Variables

- GM_ENTRY_POINT const unsigned char gm_log2_roundup_table [257]

8.1.1 Detailed Description

The official GM API include file.

author: glenn@myri.com

8.1.2 Define Documentation

8.1.2.1 #define GM_API_VERSION_1_0 0x100

Hex equivalent of GM_API_VERSION_1_0

8.1.2.2 #define GM_API_VERSION_1_1 0x101

Hex equivalent of GM_API_VERSION_1_1

8.1.2.3 #define GM_API_VERSION_1_2 0x102

Hex equivalent of GM_API_VERSION_1_2

8.1.2.4 #define GM_API_VERSION_1_3 0x103

Hex equivalent of GM_API_VERSION_1_3

8.1.2.5 #define GM_API_VERSION_1_4 0x104

Hex equivalent of GM_API_VERSION_1_4

8.1.2.6 #define GM_API_VERSION_1_5 0x105

Hex equivalent of GM_API_VERSION_1_5

8.1.2.7 #define GM_API_VERSION_1_6 0x106

Hex equivalent of GM_API_VERSION_1_6

8.1.2.8 #define GM_API_VERSION_2_0 0x200

Hex equivalent of GM_API_VERSION_2_0

8.1.2.9 #define GM_API_VERSION_2_0_6 0x20006

Hex equivalent of GM_API_VERSION_2_0_6

8.1.2.10 #define GM_API_VERSION GM_API_VERSION_2_0_6

Set the default API version used in this file.

8.1.2.11 #define GM_MAX_HOST_NAME_LEN 128

Maximum length of GM host name

8.1.2.12 #define GM_MAX_PORT_NAME_LEN 32

Maximum length of GM port name

8.1.2.13 #define GM_NO_SUCH_NODE_ID 0

No such GM node id

8.1.2.14 #define GM_CPU_alpha 0

Define all undefined GM_CPU switches to 0 to prevent problems with "gcc -Wundef"

8.1.2.15 #define GM_RDMA_GRANULARITY 64

GM RDMA GRANULARITY

8.1.2.16 #define GM_MAX_DMA_GRANULARITY 8

GM MAX DMA GRANULARITY

8.1.2.17 #define GM_STRUCT_CONTAINING(type, field, field_instance) ((type *)((char *)(field_instance) - GM_OFFSETOF (type, field)))

Given a pointer to an instance of a field in a structure of a certain type, return a pointer to the containing structure.

8.1.2.18 #define GM_NUM_ELEM(ar) (sizeof (ar) / sizeof (*ar))

Given an array, return the number of elements in the array.

8.1.2.19 #define GM_POWER_OF_TWO(n) (!((n)&((n)-1)))

Return nonzero if the input is neither a power of two nor zero. Otherwise, return zero.

8.1.2.20 #define GM_STRUCT_CONTAINING(type, field, field_instance) ((type *)((char *)(field_instance) - GM_OFFSETOF (type, field)))

Given a pointer to an instance of a field in a structure of a certain type, return a pointer to the containing structure.

8.1.2.21 #define GM_NUM_ELEM(ar) (sizeof (ar) / sizeof (*ar))

Given an array, return the number of elements in the array.

8.1.2.22 #define GM_POWER_OF_TWO(n) (!((n)&((n)-1)))

Return nonzero if the input is neither a power of two nor zero. Otherwise, return zero.

8.1.3 Typedef Documentation

8.1.3.1 typedef gm_u64_t gm_remote_ptr_t

The "gm_up_t" is a LANai-compatible representation of a user virtual memory address. If the host has 32-bit pointers, then a gm_up_t has 32 bits. If the host has 64-bit pointers, but only 32-bits of these pointers are used, then a gm_up_t still has 32 bits (for performance reasons). Finally, if the host has 64-bit pointers and more than 32 bits of the pointer is used, then a gm_up_t has 64-bits.

8.1.3.2 typedef enum gm_status gm_status_t

GM Send Completion Status codes

8.1.3.3 typedef void(* gm_send_completion_callback_t)(struct gm_port * p, void *context, gm_status_t status)

gm_send_completion_callback_t typedef function.

8.1.3.4 typedef gm_u32_t gm_pid_t

typedef for gm_pid_t.

8.1.4 Enumeration Type Documentation

8.1.4.1 enum gm_status

GM Send Completion Status codes

Enumeration values:

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

GM_FAILURE Operation Failed

GM_INPUT_BUFFER_TOO_SMALL Input buffer is too small

GM_OUTPUT_BUFFER_TOO_SMALL Output buffer is too small

GM_TRY_AGAIN Try Again

GM_BUSY GM Port is Busy

GM_MEMORY_FAULT Memory Fault

GM_INTERRUPTED Interrupted

GM_INVALID_PARAMETER Invalid input parameter

GM_OUT_OF_MEMORY Out of Memory

GM_INVALID_COMMAND Invalid Command

GM_PERMISSION_DENIED Permission Denied

GM_INTERNAL_ERROR Internal Error

GM_UNATTACHED Unattached

GM_UNSUPPORTED_DEVICE Unsupported Device

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.

GM_NODE_ID_NOT_YET_SET Node ID is not yet set

GM_STILL_SHUTTING_DOWN GM Port is still shutting down

GM_CLONE_BUSY GM Clone Busy

GM_NO_SUCH_DEVICE No such device

GM_ABORTED Aborted.

GM_INCOMPATIBLE_LIB_AND_DRIVER Incompatible GM library and driver

GM_UNTRANSLATED_SYSTEM_ERROR Untranslated System Error

GM_ACCESS_DENIED Access Denied
GM_NO_DRIVER_SUPPORT No Driver Support
GM_PTE_REF_CNT_OVERFLOW PTE Ref Cnt Overflow
GM_NOT_SUPPORTED_IN_KERNEL Not supported in the kernel
GM_NOT_SUPPORTED_ON_ARCH Not supported for this architecture
GM_NO_MATCH No match
GM_USER_ERROR User error
GM_TIMED_OUT Timed out
GM_DATA_CORRUPTED Data has been corrupted
GM_HARDWARE_FAULT Hardware fault
GM_SEND_ORPHANED Send orphaned
GM_MINOR_OVERFLOW Minor overflow
GM_PAGE_TABLE_FULL Page Table is Full
GM_UC_ERROR UC Error
GM_INVALID_PORT_NUMBER Invalid Port Number
GM_DEV_NOT_FOUND No device files found
GM_FIRMWARE_NOT_RUNNING Lanai not running
GM_YP_NO_MATCH No match for yellow pages query.

8.1.4.2 enum gm_priority

Priority Levels

Enumeration values:
GM_LOW_PRIORITY Low priority message
GM_HIGH_PRIORITY High priority message
GM_NUM_PRIORITIES Number of priority types

8.1.4.3 enum gm_recv_event_type

Receive Event Types

Enumeration values:
GM_NO_RECV_EVENT No significant receive event is pending.

GM_SENDS_FAILED_EVENT deprecated

GM_ALARM_EVENT This event should be treated as an unknown event (passed to gm_unknown())

GM_RECV_EVENT This event indicates that a normal receive (GM_LOW_PRIORITY) has occurred.

GM_HIGH_RECV_EVENT This event indicates that a normal receive (GM_HIGH_PRIORITY) has occurred.

GM_PEER_RECV_EVENT This event indicates that a normal receive (GM_LOW_PRIORITY) has occurred, and the PEER indicates that the sender/receiver ports are the same.

GM_HIGH_PEER_RECV_EVENT This event indicates that a normal receive (GM_HIGH_PRIORITY) has occurred, and the PEER indicates that the sender/receiver ports are the same.

GM_FAST_RECV_EVENT A small-message receive occurred (GM_LOW_PRIORITY) with the small message stored in the receive queue for improved small-message performance.

GM_FAST_HIGH_RECV_EVENT A small-message receive occurred (GM_HIGH_PRIORITY) with the small message stored in the receive queue for improved small-message performance.

GM_FAST_PEER_RECV_EVENT A small-message receive occurred (GM_LOW_PRIORITY) with the small message stored in the receive queue for improved small-message performance. The PEER indicates that the sender/receiver ports are the same.

GM_FAST_HIGH_PEER_RECV_EVENT A small-message receive occurred (GM_HIGH_PRIORITY) with the small message stored in the receive queue for improved small-message performance. The PEER indicates that the sender/receiver ports are the same.

GM_NEW_SENDS_FAILED_EVENT deprecated

GM_NEW_RECV_EVENT normal receives

GM_NEW_FAST_RECV_EVENT streamlined small message receives

_GM_NEW_PUT_NOTIFICATION_EVENT Directed send notification

GM_NUM_RECV_EVENT_TYPES DO NOT add new types here.

8.1.5 Function Documentation

8.1.5.1 GM_ENTRY_POINT void gm_abort (void)

gm_abort() aborts the current process, and is a wrapper around the system function abort().

See also:
gm_init gm_open gm_close gm_exit gm_finalize

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.2 GM_ENTRY_POINT int gm_alloc_send_token (struct gm_port * *p*, unsigned int *priority*)

Allocates a send token (Details).

8.1.5.3 GM_ENTRY_POINT gm_status_t gm_allow_remote_memory_access (struct gm_port * *port*)

gm_allow_remote_memory_access() allows 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.

Return values:
GM_SUCCESS Operation completed successfully.

Parameters:
port **(IN)** Handle to the GM port.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.4 GM_ENTRY_POINT void gm_bcopy (const void * *from*, void * *to*, gm_size_t *len*)

gm_bcopy() copies len bytes starting at **from** to location **to**. This function does not handle overlapping regions.

Parameters:

from **(IN)** The starting location for the region to be copied.

to **(IN)** The ending location for the region to be copied.

len **(IN)** The length in bytes of the region to be copied.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.5 GM_ENTRY_POINT union gm_recv_event* gm_blocking_receive (struct gm_port * *p*)

Blocks until there is a receive event and then returns a pointer to the event. (Details).

8.1.5.6 GM_ENTRY_POINT union gm_recv_event* gm_blocking_receive_no_spin (struct gm_port * *p*)

Like gm_blocking_receive(), except it sleeps the current thread immediately if no receive is pending. (gm_blocking_receive_no_spin "Details").

8.1.5.7 GM_ENTRY_POINT void gm_bzero (void * *ptr*, gm_size_t *len*)

gm_bzero() clears the **len** bytes of memory starting at **ptr**. This function does not use partword I/O unless it must (for speed, especially when doing PIO), and does not rely on the system bzero() functionality, which may not be safe for PIO mapped memory.

Parameters:

ptr **(IN)** The pointer to the memory location.

len **(IN)** The number of bytes of memory to be bzero'ed.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.8 GM_ENTRY_POINT void* gm_calloc (gm_size_t *len*, gm_size_t *cnt*)

gm_calloc() allocates and clears an array of **cnt** elements of length **len**.

Parameters:

len **(IN)** The number of bytes in each element.

cnt **(IN)** The number of elements in the array.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.9 GM_ENTRY_POINT void gm_cancel_alarm (struct gm_alarm * *gm_alarm*)

Cancels alarm (Details).

8.1.5.10 GM_ENTRY_POINT void gm_close (struct gm_port * *p*)

Closes a GM port (Details).

8.1.5.11 GM_ENTRY_POINT void gm_datagram_send (struct gm_port * *p*, void * *message*, unsigned int *size*, gm_size_t *len*, unsigned int *priority*, unsigned int *target_node_id*, unsigned int *target_port_id*, gm_send_completion_callback_t *callback*, void * *context*)

Unreliable send (Details).

8.1.5.12 GM_ENTRY_POINT void gm_datagram_send_4 (struct gm_port * *p*, gm_u32_t *message*, unsigned int *size*, gm_size_t *len*, unsigned int *priority*, unsigned int *target_node_id*, unsigned int *target_port_id*, gm_send_completion_callback_t *callback*, void * *context*)

Unreliable send of gm_u32_t messages (Details).

8.1.5.13 GM_ENTRY_POINT gm_status_t gm_deregister_memory (struct gm_port * *p*, void * *ptr*, gm_size_t *length*)

Deregisters memory (Details).

8.1.5.14 GM_ENTRY_POINT void gm_directed_send_with_callback (struct gm_port * *p*, void * *source_buffer*, gm_remote_ptr_t *target_buffer*, gm_size_t *len*, enum gm_priority *priority*, unsigned int *target_node_id*, unsigned int *target_port_id*, gm_send_completion_callback_t *callback*, void * *context*)

Directed send (PUT) (Details).

8.1.5.15 GM_ENTRY_POINT void* gm_dma_calloc (struct gm_port * *p*, gm_size_t *count*, gm_size_t *length*)

gm_dma_calloc() allocates and clears **count** * **length** bytes of DMAable memory aligned on a 4-byte boundary. Memory should be freed using gm_dma_free().

Parameters:

p **(IN)** Handle to the GM port.

count **(IN)** The number of elements to be calloc'ed.

length **(IN)** The size of each element to be calloc'ed.

See also:

gm_dma_malloc gm_dma_free

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.16 GM_ENTRY_POINT void gm_dma_free (struct gm_port * *p*, void * *addr*)

Frees a region of DMAable memory (Details).

8.1.5.17 GM_ENTRY_POINT void* gm_dma_malloc (struct gm_port * *p*, gm_size_t *length*)

gm_dma_malloc() allocates **length** bytes of DMAable memory aligned on a 4-byte boundary. Memory should be freed using gm_dma_free().

Parameters:

p **(IN)** Handle to the GM port.

length **(IN)** The number of bytes to be malloc'ed.

See also:
gm_dma_calloc gm_dma_free

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.18 GM_ENTRY_POINT void gm_flush_alarm (struct gm_port * *p*)

Flushes an alarm (Details).

8.1.5.19 GM_ENTRY_POINT void gm_free (void * *ptr*)

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

Parameters:
ptr **(IN)** Address of the memory to be freed.

See also:
gm_malloc gm_calloc

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.20 GM_ENTRY_POINT void gm_free_send_token (struct gm_port * *p*, unsigned int *priority*)

Frees a send token (Details).

8.1.5.21 GM_ENTRY_POINT void gm_free_send_tokens (struct gm_port * *p*, unsigned int *priority*, unsigned int *count*)

Frees multiple send tokens (Details).

8.1.5.22 GM_ENTRY_POINT gm_status_t gm_get_host_name (struct gm_port * *port*, char *name*[GM_MAX_HOST_NAME_LEN])

Copies the host name of the local node (Details).

8.1.5.23 GM_ENTRY_POINT gm_status_t gm_get_node_type (struct gm_port * *port*, int * *node_type*)

Returns GM_GET_NODE_TYPE (Details).

8.1.5.24 GM_ENTRY_POINT gm_status_t gm_get_node_id (struct gm_port * *port*, unsigned int * *n*)

Copies the GM ID of the interface (Details).

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

Copies the board ID of the interface (gm_get_unique_board_id "Details").

8.1.5.26 GM_ENTRY_POINT gm_status_t gm_get_mapper_unique_id (struct gm_port * *port*, char *unique*[6])

Copies copies the 6-byte ethernet address of the interface (Details).

8.1.5.27 GM_ENTRY_POINT void gm_hex_dump (const void * *_ptr*, gm_size_t *len*)

gm_hex_dump() prints the hex equivalent of data at **_ptr**.

Parameters:

_ptr **(IN)** Address of the data.

len **(IN)** The length (in bytes) of the data.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.28 GM_ENTRY_POINT unsigned int gm_host_name_to_node_id (struct gm_port * *port*, char * *_host_name*)

This function is deprectated. Use gm_host_name_to_node_id_ex() instead. Returns the GM ID associated with a host_name (Details).

8.1.5.29 GM_ENTRY_POINT void gm_initialize_alarm (struct gm_alarm * *my_alarm*)

Initializes user-allocated storage for an alarm. (Details).

8.1.5.30 GM_ENTRY_POINT int gm_isprint (int *c*)

gm_isprint() is just like ANSI isprint(), only it works in the kernel and MCP.

Return values:

int The return value is nonzero if the character is a printable character including a space, and a zero value if not.

Parameters:

c **(OUT)** Printable character.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.31 GM_ENTRY_POINT void* gm_malloc (gm_size_t *len*)

gm_malloc() returns a pointer to the specified amount of memory. In the kernel, the memory will be nonpageable.

Return values:

ptr Pointer to the specified amount of memory.

Parameters:

len **(IN)** The length in bytes to be malloc'ed.

Author:

Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.32 GM_ENTRY_POINT void* gm_page_alloc (void)

gm_page_alloc() returns a ptr to a newly allocated page-aligned buffer of length GM_PAGE_LEN.

Return values:
ptr Page-aligned buffer of length GM_PAGE_LEN.

See also:
gm_page_free

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.33 GM_ENTRY_POINT void gm_page_free (void * *ptr*)

gm_page_free() frees the page of memory at **ptr** previously allocated by gm_page_alloc(). If all pages have been freed, free all of the memory allocated for pages.

Parameters:
ptr **(IN)** Address of the memory page to be freed.

See also:
gm_page_alloc

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.34 GM_ENTRY_POINT void* gm_alloc_pages (gm_size_t *alloc_len*)

gm_alloc_pages() allocates a page-aligned buffer of length ALLOC_LEN, where ALLOC_LEN is a multiple of GM_PAGE_LEN. Any fractional page following the buffer is wasted.

Return values:

ptr Pointer to the allocated buffer.

0 Error occurred.

Parameters:

alloc_len **(IN)** The length of buffer to be allocated.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.35 GM_ENTRY_POINT void gm_free_pages (void * *addr*, gm_size_t *alloc_len*)

gm_free_pages() frees the pages at **addr**, which were previously allocated with gm_alloc_pages().

Parameters:

addr **(IN)** The address of the buffer to be freed.

alloc_len **(IN)** The length (in bytes) of the buffer to be freed.

See also:

gm_alloc_pages

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.36 GM_ENTRY_POINT gm_size_t gm_max_length_for_size (unsigned int *size*)

gm_max_length_for_size() returns the maximum length of a message that will fit in a GM buffer of size **size**.

Return values:
gm_size_t

Parameters:
size **(IN)** The size of the GM buffer.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

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

Stores the maximum GM node ID supported by the network interface card (Details).

8.1.5.38 GM_ENTRY_POINT int gm_memcmp (const void * *_a*, const void * *_b*, gm_size_t *len*)

gm_memcmp() emulates the ANSI memcmp() function.

Return values:
int Returns an integer less than, equal to, or greater than zero if the first len bytes of a is found, respectively, to be less than, to match, or be greater than the first len bytes of b.

Parameters:
a (IN) The first memory area for comparison.
b (IN) The second memory area for comparison.
len (IN) The number of bytes to compare.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.39 GM_ENTRY_POINT void* gm_memorize_message (void * *message*, void * *buffer*, unsigned *len*)

gm_memorize_message() is a wrapper around function memcpy(). gm_memorize_message() copies a message into a buffer if needed. 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 optionally optimizes the handling of FAST receive messages as described in "See Chapter 9 [Receiving Messages].

Return values:
something

Parameters:
message **(IN)** Address of the message to be copied.

buffer **(OUT)** Address where the message is to be copied.

len **(IN)** The length in bytes of the message to be copied.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.40 GM_ENTRY_POINT unsigned int gm_min_message_size (struct gm_port * *port*)

Returns the minimum supported message size (Details).

8.1.5.41 GM_ENTRY_POINT unsigned int gm_min_size_for_length (gm_size_t *length*)

gm_min_size_for_length() returns the minimum GM message buffer size required to store a message of length **length**.

Return values:
gm_log2_roundup

Parameters:
length **(IN)** The length of the message.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.42 GM_ENTRY_POINT unsigned int gm_mtu (struct gm_port * *port*)

gm_mtu() returns the value of GM_MTU.

Return values:
GM_MTU

Parameters:
port **(IN)** The handle to the GM port.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.43 GM_ENTRY_POINT char* gm_node_id_to_host_name (struct gm_port * *port*, unsigned int *node_id*)

This function is deprecated. Use gm_node_id_to_host_name_ex() instead. Returns a pointer to the host name of the host containing the network interface card with GM node id node_id. (Details).

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

Stores the MAC address for the interface (gm_node_id_to_unique_id "Details").

8.1.5.45 GM_ENTRY_POINT unsigned int gm_num_ports (struct gm_port * *p*)

Returns the number of ports supported by this build. (Details).

8.1.5.46 GM_ENTRY_POINT unsigned int gm_num_send_tokens (struct gm_port * p)

Returns the number of send tokens for this port. (Details).

8.1.5.47 GM_ENTRY_POINT unsigned int gm_num_receive_tokens (struct gm_port * p)

Returns the number of receive tokens for this port. (Details).

8.1.5.48 GM_ENTRY_POINT unsigned int gm_get_port_id (struct gm_port * p)

Returns the id of the GM port (Details).

8.1.5.49 GM_ENTRY_POINT gm_status_t gm_open (struct gm_port ** p, unsigned int *unit*, unsigned int *port*, const char * *port_name*, enum gm_api_version *version*)

Opens a GM port on an interface (Details).

8.1.5.50 GM_ENTRY_POINT void gm_provide_receive_buffer_with_tag (struct gm_port * p, void * *ptr*, unsigned int *size*, unsigned int *priority*, unsigned int *tag*)

Provides GM with a buffer into which it can receive messages (Details).

8.1.5.51 GM_ENTRY_POINT int gm_receive_pending (struct gm_port * p)

Returns nonzero if a receive event is pending (gm_receive_pending "Details").

8.1.5.52 GM_ENTRY_POINT int gm_next_event_peek (struct gm_port * p, gm_u16_t * *sender*)

Returns the nonzero event type if an event is pending (Details).

8.1.5.53 GM_ENTRY_POINT union gm_recv_event* gm_receive (struct gm_port * p)

Returns a receive event. (Details).

8.1.5.54 GM_ENTRY_POINT gm_status_t gm_register_memory (struct gm_port * *p*, void * *ptr*, gm_size_t *length*)

Registers virtual memory for DMA transfers. (gm_register_memory "Details").

8.1.5.55 GM_ENTRY_POINT int gm_send_token_available (struct gm_port * *p*, unsigned int *priority*)

Tests for the availability of a send token without allocating the send token. (Details).

8.1.5.56 GM_ENTRY_POINT void gm_send_with_callback (struct gm_port * *p*, void * *message*, unsigned int *size*, gm_size_t *len*, unsigned int *priority*, unsigned int *target_node_id*, unsigned int *target_port_id*, gm_send_completion_callback_t *callback*, void * *context*)

A fully asynchronous send. (Details).

8.1.5.57 GM_ENTRY_POINT void gm_send_to_peer_with_callback (struct gm_port * *p*, void * *message*, unsigned int *size*, gm_size_t *len*, unsigned int *priority*, unsigned int *target_node_id*, gm_send_completion_callback_t *callback*, void * *context*)

A fully asychronous send from/to the same GM port on the sending and receiving side. (Details).

8.1.5.58 GM_ENTRY_POINT gm_status_t gm_set_acceptable_sizes (struct gm_port * *p*, enum gm_priority *priority*, gm_size_t *mask*)

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

Note: the MASK is a long to support larger than 2GByte packets (those with size larger than 31).

Return values:

GM_SUCCESS Operation completed successfully.

GM_PERMISSION_DENIED Port number hasn't been set.

GM_INTERNAL_ERROR LANai is not running.

GM_INVALID_PARAMETER The priority has an invalid value.

Parameters:

p **(IN)** The GM port in use.

priority **(IN)** The priority of the message.

mask

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.59 GM_ENTRY_POINT void gm_set_alarm (struct gm_port * *p*, struct gm_alarm * *my_alarm*, gm_u64_t *usecs*, void(* *callback*)(void *), void * *context*)

Sets an alarm, which may already be pending. (gm_set_alarm "Details").

8.1.5.60 GM_ENTRY_POINT gm_size_t gm_strlen (const char * *cptr*)

gm_strlen() calculates the length of a string.

Return values:

gm_size_t The length of the string.

Parameters:

cptr (IN) The string.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.61 GM_ENTRY_POINT char* gm_strncpy (char * *to*, const char * *from*, int *len*)

gm_strncpy() copies exactly n bytes, truncating src or adding null characters to dst if necessary. The result will not be null-terminated if the length of src is n or more.

Return values:
char * Returns a pointer to a destination string.

Parameters:
to (IN) The destination string to be copied.
from (IN) The source string to be copied.
len (IN) The number of bytes to be copied.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.62 GM_ENTRY_POINT int gm_strcmp (const char * *a*, const char * *b*)

gm_strcmp() reimplements strcmp().

Return values:
int Returns an integer less than, equal to, or greater than zero if a is found, respectively, to be less than, to match, or be greater than b.

Parameters:
a The first string to be compared.
b The second string to be compared.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.63 GM_ENTRY_POINT int gm_strncmp (const char * *a*, const char * *b*, int *len*)

gm_strncmp() reimplements strncmp().

Return values:
int Returns an integer less than, equal to, or greater than zero if the first len bytes of a is found, respectively, to be less than, to match, or be greater than b.

Parameters:

a (IN) The first string to be compared.

b (IN) The second string to be compared.

len (IN) The length in bytes.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.64 GM_ENTRY_POINT int gm_strncasecmp (const char * *a*, const char * *b*, int *len*)

gm_strncasecmp() reimplements strncasecmp().

Return values:

int Returns an integer less than, equal to, or greater than zero if the first len bytes of a is found, respectively, to be less than, to match, or be greater than b.

Parameters:

a (IN) The first string to be compared.

b (IN) The second string to be compared.

len (IN) The number of bytes.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.65 GM_ENTRY_POINT gm_u64_t gm_ticks (struct gm_port * *port*)

gm_ticks() returns a 64-bit extended version of the LANai real time clock (RTC). For implementation reasons, the granularity of gm_ticks() is 50 microseconds at the application level.

Return values:

gm_u64_t

Parameters:
port **(IN)** The handle to the GM port.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.66 GM_ENTRY_POINT gm_status_t gm_unique_id (struct gm_port * *port*, char *unique*[6])

Returns the board id number for an interface. (gm_unique_id "Details").

8.1.5.67 GM_ENTRY_POINT gm_status_t gm_unique_id_to_node_id (struct gm_port * *port*, char *unique*[6], unsigned int * *node_id*)

Returns the GM node id for a specific interface. (Details).

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

GM Event Handler. (Details).

8.1.5.69 GM_ENTRY_POINT gm_status_t _gm_get_route (struct gm_port * *p*, unsigned int *node_id*, char * *route*, unsigned int * *len*)

_gm_get_route function. (Details).

8.1.5.70 GM_ENTRY_POINT void gm_dump_buffers (void)

gm_dump_buffers()

Author:
??

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.71 GM_ENTRY_POINT void gm_register_buffer (void * *addr*, int *size*)

gm_register_buffer() registered a GM buffer.

Return values:

1

0

Parameters:

addr **(IN)** Address of data to be registered.

size **(IN)** Size of buffer to be registered.

Author:

??

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.72 GM_ENTRY_POINT int gm_unregister_buffer (void * *addr*, int *size*)

gm_unregister_buffer() unregistered a GM buffer.

Return values:

1

0

Parameters:

addr **(IN)** Address of data to be freed.

size **(IN)** Size of buffer to be freed.

Author:

??

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.73 GM_ENTRY_POINT struct gm_lookaside* gm_create_lookaside (gm_size_t *entry_len*, gm_size_t *min_entry_cnt*)

gm_create_lookaside() returns a newly created lookaside list to be used to allocate blocks of ENTRY_LEN bytes. MIN_ENTRY_CNT entries are preallocated.

Return values:

gm_lookaside Handle to the lookaside table.

Parameters:

entry_len **(IN)**

min_entry_cnt **(IN)**

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.74 GM_ENTRY_POINT void gm_destroy_lookaside (struct gm_lookaside * *l*)

gm_destroy_lookaside() frees a lookaside list and all associated resources, including any buffers currently allocated from the lookaside list.

Parameters:

l **(IN)** Handle to the lookaside table.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.75 GM_ENTRY_POINT void* gm_lookaside_alloc (struct gm_lookaside * *l*)

gm_lookaside_alloc() allocates an entry from the lookaside table, with debugging. It returns a buffer of size ENTRY_LEN specified when the entry list L was created, or '0' if the buffer could not be allocated.

Return values:
ptr Buffer of size ENTRY_LEN.

Parameters:
l **(IN)** Handle to the gm_lookaside list.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.76 GM_ENTRY_POINT void* gm_lookaside_zalloc (struct gm_lookaside * *l*)

gm_lookaside_zalloc() allocates and clear an entry from the lookaside table.

Return values:
ptr

Parameters:
l **(IN)** Handle to the gm_lookaside list.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.77 GM_ENTRY_POINT void gm_lookaside_free (void * *ptr*)

gm_lookaside_free() schedules an allocated entry to be freed, and actually performs any scheduled free. It 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.

Parameters:
ptr **(IN)** Pointer to the entry to be freed.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.78 GM_ENTRY_POINT struct gm_hash* gm_create_hash (long(* *gm_user_compare*)(void *key1, void *key2), unsigned long(* *gm_user_hash*)(void *key1), gm_size_t *key_len*, gm_size_t *data_len*, gm_size_t *gm_min_entries*, int *flags*)

Creates a hash table. (Details).

8.1.5.79 GM_ENTRY_POINT void gm_destroy_hash (struct gm_hash * *hash*)

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

Parameters:
hash **(IN)** Handle to the hash table.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.80 GM_ENTRY_POINT void* gm_hash_remove (struct gm_hash * *hash*, void * *key*)

Removes an entry from the hash table. (Details).

8.1.5.81 GM_ENTRY_POINT void* gm_hash_find (struct gm_hash * *hash*, void * *key*)

Finds an entry in the hash table. (Details).

8.1.5.82 GM_ENTRY_POINT gm_status_t gm_hash_insert (struct gm_hash * *hash*, void * *key*, void * *datum*)

Inserts an entry in the hash table. (Details).

8.1.5.83 GM_ENTRY_POINT void gm_hash_rekey (struct gm_hash * *hash*, void * *old_key*, void * *new_key*)

Replaces a key in the hash table. (Details).

8.1.5.84 GM_ENTRY_POINT long gm_hash_compare_strings (void * *key1*, void * *key2*)

gm_hash_compare_strings() is the function used to compare two strings.

Return values:
long

Parameters:
key1 **(IN)** The key for the first string.
key2 **(IN)** The key for the second string.

See also:
gm_hash_compare_ints gm_hash_compare_longs gm_hash_compare_ptrs

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.85 GM_ENTRY_POINT unsigned long gm_hash_hash_string (void * *key*)

gm_hash_hash_string() is the function used to hash keys.

Return values:
long

Parameters:
key **(IN)** The key for the string in the hash table.

See also:
gm_hash_compare_string gm_hash_hash_int gm_hash_hash_long gm_hash_hash_ptr

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.86 GM_ENTRY_POINT long gm_hash_compare_longs (void * *key1*, void * *key2*)

gm_hash_compare_longs() is the function used to compare two longs.

Return values:
long

Parameters:
key1 **(IN)**

key2 **(IN)**

See also:
gm_hash_compare_ints gm_hash_compare_strings gm_hash_compare_ptrs

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.87 GM_ENTRY_POINT unsigned long gm_hash_hash_long (void * *key*)

gm_hash_hash_long() is the function used to hash keys.

Return values:
long

Parameters:
key **(IN)**

See also:
gm_hash_compare_long gm_hash_hash_int gm_hash_hash_string gm_hash_hash_ptr

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.88 GM_ENTRY_POINT long gm_hash_compare_ints (void * *key1*, void * *key2*)

gm_hash_compare_ints() is the function used to compare two ints.

Return values:
long

Parameters:
key1 **(IN)** The key for the first int.

key2 **(IN)** The key for the second int.

See also:
gm_hash_compare_longs gm_hash_compare_strings gm_hash_compare_ptrs

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.89 GM_ENTRY_POINT unsigned long gm_hash_hash_int (void * *key*)

gm_hash_hash_int() is the function used to hash keys.

Return values:
long

Parameters:
key **(IN)**

See also:
gm_hash_compare_int gm_hash_hash_ptr gm_hash_hash_long gm_hash_hash_string

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.90 GM_ENTRY_POINT long gm_hash_compare_ptrs (void * *key1*, void * *key2*)

gm_hash_compare_ptrs() is the function used to compare two ptrs.

Return values:
long

Parameters:
key1 **(IN)** The key for the first ptr.

key2 **(IN)** The key for the second ptr.

See also:
gm_hash_compare_longs gm_hash_compare_strings gm_hash_compare_ints

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.91 GM_ENTRY_POINT unsigned long gm_hash_hash_ptr (void * *key*)

gm_hash_hash_ptr() is the function used to hash keys.

Return values:
long

Parameters:
key **(IN)**

See also:
gm_hash_compare_ptr gm_hash_hash_int gm_hash_hash_long gm_hash_hash_string

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.92 GM_ENTRY_POINT unsigned long gm_crc (void * *_ptr*, gm_size_t *len*)

gm_crc() computes a CRC-32 of the indicated range of memory.

Return values:
long

Parameters:
_ptr **(IN)** Pointer to a range of memory.

len **(IN)** The length of the indicated range of memory.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.93 GM_ENTRY_POINT unsigned long gm_crc_str (const char * *ptr*)

gm_crc_str() computes a CRC-32 for the indicated string.

Return values:
long

Parameters:
ptr **(IN)** Pointer to a string.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.94 GM_ENTRY_POINT int gm_rand (void)

gm_rand() returns a pseudo-random integer, using a poor but fast random number generator.

Return values:
RANDOM_NUMBER The random number that was generated.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.95 GM_ENTRY_POINT void gm_srand (int *seed*)

gm_srand() returns a pseudo-random integer, and requires a seed for the random number generator.

Parameters:
seed **(IN)** Seed for the random number generator.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.96 GM_ENTRY_POINT unsigned int gm_rand_mod (unsigned int *a*)

gm_rand_mod() returns a pseudo-random number modulo **modulus**, using

a poor but fast random number generator.

Return values:
RANDOM_NUMBER

Parameters:
a **(IN)** The modulus bound.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.97 GM_ENTRY_POINT gm_status_t gm_init (void)

gm_init() initializes GM. It increments the GM initialization counter and initializes 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().

Return values:

GM_SUCCESS Operation completed successfully.

GM_FAILURE Error occurred.

See also:

gm_finalize gm_open gm_close gm_exit gm_abort

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.98 GM_ENTRY_POINT void gm_finalize (void)

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

See also:

gm_init gm_open gm_close gm_abort gm_exit

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.99 GM_ENTRY_POINT unsigned long gm_log2_roundup (unsigned long *n*)

gm_log2_roundup() returns the logarithm, base 2, of **n**, rounding up to the next integer.

Return values:
LOG

Parameters:
n **(IN)** The integer for which the logarithm will be computed.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.100 GM_ENTRY_POINT struct gm_mutex* gm_create_mutex (void)

gm_create_mutex() creates a GM mutex.

Return values:
gm_mutex **(OUT)** Handle to the GM mutex.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.101 GM_ENTRY_POINT void gm_destroy_mutex (struct gm_mutex * *mu*)

gm_destroy_mutex() destroys a GM mutex.

Parameters:
mu **(IN)** The handle to the GM mutex.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.102 GM_ENTRY_POINT void gm_mutex_enter (struct gm_mutex * *mu*)

gm_mutex_enter()

Parameters:
mu **(IN)** The handle to the GM mutex.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.103 GM_ENTRY_POINT void gm_mutex_exit (struct gm_mutex * *mu*)

gm_mutex_exit()

Parameters:
mu **(IN)** The handle to the GM mutex.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.104 GM_ENTRY_POINT struct gm_zone* gm_zone_create_zone (void * *base*, gm_size_t *length*)

gm_zone_create_zone()

Return values:
gm_zone Handle to the GM zone.

Parameters:
base

length

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.105 GM_ENTRY_POINT void gm_zone_destroy_zone (struct gm_zone * *zone*)

gm_zone_destroy_zone()

Parameters:
zone **(IN)** Pointer to the GM zone.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.106 GM_ENTRY_POINT void* gm_zone_free (struct gm_zone * *zone*, void * *a*)

gm_zone_free()

Parameters:
zone **(IN)** Pointer to the GM zone.

a

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.107 GM_ENTRY_POINT void* gm_zone_malloc (struct gm_zone * *zone*, gm_size_t *length*)

gm_zone_malloc() mallocs a GM zone.

Parameters:

zone **(IN)** Pointer to the GM zone.

length

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.108 GM_ENTRY_POINT void* gm_zone_calloc (struct gm_zone * *zone*, gm_size_t *count*, gm_size_t *length*)

gm_zone_calloc() callocs a GM zone.

Parameters:

zone **(IN)** Pointer to the GM zone.

count

length

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.109 GM_ENTRY_POINT int gm_zone_addr_in_zone (struct gm_zone * *zone*, void * *p*)

gm_zone_addr_in_zone()

Parameters:

zone **(IN)** Pointer to the GM zone.

p

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.110 GM_ENTRY_POINT void gm_resume_sending (struct gm_port * *p*, unsigned int *priority*, unsigned int *target_node_id*, unsigned int *target_port_id*, gm_send_completion_callback_t *callback*, void * *context*)

gm_resume_sending(0 reenables 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. The message that generated the error is not resent. The first four parameters must match those of the failed send. It should be called only once per reported error. This function requires a send token, which will be returned to the client in the callback function.

gm_resume_sending() and gm_drop_sends(), as most gm requests, require a send token, and the callback you give to them is just meant to return this token. These gm_requests always succeed (if called in a valid manner), so the callback will **always** be called with **GM_SUCCESS** (which here does not mean at all that something was sent successfully, just that the request has been taken into account, and the token used for that request was recycled).

Parameters:

p **(IN)** The handle to the GM port.

priority **(IN)** The priority of the message being sent.

target_node_id **(IN)** The GM node to which the message is being sent.

target_port_id **(IN)** The GM port on the destination GM node to which the message is being sent.

callback **(IN)** The function called when the send is complete.

context **(IN)** Pointer to an integer or to a structure that is passed to the callback function.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.111 GM_ENTRY_POINT 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*)

gm_drop_sends() drops 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. The first four parameters must match those of the failed

send. It should be called only once per reported error. This function requires a send token, which will be returned to the client in the callback function. The dropped sends will then be returned to the client with a status of GM_SEND_DROPPED.

gm_drop_sends() and gm_resume_sending(), as most gm requests, require a send token, and the callback you give to them is just meant to return this token. These gm_requests always succeed (if called in a valid manner), so the callback will **always** be called with **GM_SUCCESS** (which here does not mean at all that something was sent successfully, just that the request has been taken into account, and the token used for that request was recycled).

Parameters:

port **(IN)** The handle to the GM port.

priority **(IN)** The priority of the message.

target_node_id **(IN)** The GM node to which the message is being sent.

target_port_id **(IN)** The GM port on the destination GM node to which the message was sent.

callback **(IN)** The function called when the send is complete.

context **(IN)** Pointer to an integer or to a structure that is passed to the callback function.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.112 GM_ENTRY_POINT gm_pid_t gm_getpid (void)

gm_getpid() is a cover for the usual Unix getpid() functionality.

Return values:

gm_pid_t The process ID of the parent of the current process.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.113 GM_ENTRY_POINT gm_status_t gm_directcopy_get (struct gm_port * *p*, void * *source_addr*, void * *target_addr*, gm_size_t *length*, unsigned int *source_instance_id*, unsigned int *source_port_id*)

gm_directcopy_get() copies data of length **length** bytes, specified at the address **source_addr** of the local process using the port **port_id** of the board **source_instance_id** to the memory area specified at the address **target_addr** of the current process. This implementation bypasses all of the protection of the operating system to provide a memory copy from one process's memory space to another one. The memory areas must have been registered by GM prior to calling this function in order to lock memory pages at their physical locations. There are no alignment or length constraints but the maximum performance will be reached with aligned addresses on both sides. This function is supported exclusively on Linux.

Return values:

GM_SUCCESS Operation completed successfully.

GM_FAILURE Error occurred.

Parameters:

p **(IN)** Handle to the GM port.

source_addr **(IN)** Address of the data to be copied.

target_addr **(IN)** Target address of the copied data.

length **(IN)** The length (in bytes) of the area to be copied.

source_instance_id **(IN)** The id of the interface.

source_port_id **(IN)** The port id of the interface.

Author:

Patrick Geoffray

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.114 GM_ENTRY_POINT void gm_perror (const char * *message*, gm_status_t *error*)

gm_perror() is similar to ANSI 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**.

Parameters:

message **(OUT)** Textual description of the GM error.

error **(IN)** GM Error code.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.115 GM_ENTRY_POINT int gm_sleep (unsigned int *seconds*)

gm_sleep() emulates the ANSI standard sleep(), sleeping the entire process for **seconds** seconds.

Return values:
SLEEP

Parameters:
seconds **(IN)** The number of seconds for which the process should sleep.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.116 GM_ENTRY_POINT void gm_exit (gm_status_t *status*)

gm_exit() causes the current process to exit with a status appropriate to the GM status code **status**.

Parameters:
status **(IN)** The GM status code, as specified in gm.h.

See also:
gm_init gm_open gm_close gm_finalize gm_abort

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.117 GM_ENTRY_POINT int gm_printf (const char * *format*, ...)

gm_printf() emulates or invokes the ANSI standard printf() function.

Return values:
0 Operation completed successfully.

Parameters:
format Specifies how the arguments are converted for output.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.118 GM_ENTRY_POINT char* gm_strerror (gm_status_t *error*)

gm_strerror() is an error function for GM. The error is only valid until next call to this function.

Return values:
char

Parameters:
error **(IN)** GM status code.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.119 GM_ENTRY_POINT gm_status_t gm_set_enable_nack_down (struct gm_port * *port*, int *flag*)

gm_set_enable_nack_down()

Return values:
GM_SUCCESS Operation completed successfully.

GM_PERMISSION_DENIED

Parameters:

port **(IN)** Handle to the GM port.

flag

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.120 GM_ENTRY_POINT gm_status_t gm_max_node_id_in_use (struct gm_port * *port*, unsigned int * *n*)

Returns the maximum GM node ID that is in use by the network attached to the port. (Details).

8.1.5.121 GM_ENTRY_POINT int gm_eprintf (const char * *format*, ...)

gm_eprintf() emulates or invokes the ANSI standard vprintf() function.

Return values:

0 Operation completed successfully.

Parameters:

format Specifies how the variable-length arguments are converted for output.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.122 GM_ENTRY_POINT void* gm_memset (void * *s*, int *c*, gm_size_t *n*)

gm_memset() reimplements the UNIX function memset().

Return values:

void Returns a pointer to the memory area s.

Parameters:

s (IN) The memory area.

c (IN) The constant byte size.

n (IN) The number of bytes.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.123 GM_ENTRY_POINT char * gm_strdup (const char * *in*)

gm_strdup() reimplements the UNIX function strdup().

Return values:

char * Returns a pointer to the duplicated string, or NULL if insufficient memory was available.

Parameters:

in (IN) The string to be duplicated.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.124 GM_ENTRY_POINT gm_status_t gm_mark (struct gm_mark_set * *set*, gm_mark_t * *m*)

gm_mark() adds '*MARK' to SET. Requires O(constant) time if the mark set has pre-allocated resources for the mark. Otherwise, requires O(constant) average time.

Return values:

GM_SUCCESS Operation completed successfully.

Parameters:

set **(IN)** Handle to the GM mark set.

m

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.125 GM_ENTRY_POINT int gm_mark_is_valid (struct gm_mark_set * *set*, gm_mark_t * *m*)

gm_mark_is_valid() returns nonzero value if '*MARK' is in SET. Requires O(constant) time.

Return values:
int

Parameters:
set **(IN)** Handle to the GM mark set.

m

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.126 GM_ENTRY_POINT gm_status_t gm_create_mark_set (struct gm_mark_set ** *msp*, unsigned long *cnt*)

gm_create_mark_set() returns a pointer to a new mark set at SET with enough preallocated resources to support INIT_COUNT. Returns GM_SUCCESS on success. Requires time comparable to malloc().

Return values:
GM_SUCCESS Operation completed successfully.

Parameters:
msp **(IN)** Handle to the GM mark set.

cnt

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.127 GM_ENTRY_POINT void gm_destroy_mark_set (struct gm_mark_set * *set*)

gm_destroy_mark_set() frees all resources associated with mark set '*SET'. Requires time comparable to free().

Parameters:
set **(IN)** Handle to the GM mark set.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.128 GM_ENTRY_POINT void gm_unmark (struct gm_mark_set * *set*, gm_mark_t * *m*)

gm_unmark() removes '*MARK' from SET. Requires O(constant) time.

Parameters:
set **(IN)** Handle to the GM mark set.

m

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.1.5.129 GM_ENTRY_POINT void gm_unmark_all (struct gm_mark_set * *set*)

gm_unmark_all() unmarks all marks for the mark set, freeing all but the initial number of preallocated mark references.

Removes all marks from SET. Requires O(constant) time.

Parameters:

set **(IN)** Handle to the GM mark set.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.130 GM_ENTRY_POINT gm_status_t gm_on_exit (gm_on_exit_callback_t *callback*, void * *arg*)

gm_on_exit() is like Linux on_exit(). This function 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()'.

Call the callbacks in the reverse order registered inside gm_exit(), passing GM exit status and registered argument to the callback.

Return values:

GM_SUCCESS Operation completed successfully.

GM_OUT_OF_MEMORY

Parameters:

callback

arg

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.1.5.131 GM_ENTRY_POINT void gm_put (struct gm_port * *p*, void * *local_buffer*, gm_remote_ptr_t *remote_buffer*, gm_size_t *len*, enum gm_priority *priority*, unsigned int *target_node_id*, unsigned int *target_port_id*, gm_send_completion_callback_t *callback*, void * *context*)

Directed send (PUT) (Details).

8.1.5.132 GM_ENTRY_POINT gm_status_t gm_global_id_to_node_id (struct gm_port * *port*, unsigned int *global_id*, unsigned int * *node_id*)

Stores at *node_id the local connection ID corresponding to the connection to global_id. (gm_global_id_to_node_id "Details").

8.1.5.133 GM_ENTRY_POINT gm_status_t gm_node_id_to_global_id (struct gm_port * *port*, unsigned int *node_id*, unsigned int * *global_id*)

Stores at *global_id the global node ID corresponding to the connection identified by the local connection ID. (Details).

8.1.5.134 GM_ENTRY_POINT gm_status_t gm_node_id_to_host_name_ex (struct gm_port * *port*, unsigned int *timeout_usecs*, unsigned int *node_id*, char * *name*[GM_MAX_HOST_NAME_LEN+1])

Store at *name the host name of the host containing the network interface card with GM node id node_id. (Details).

8.1.5.135 GM_ENTRY_POINT gm_status_t gm_host_name_to_node_id_ex (struct gm_port * *port*, unsigned int *timeout_usecs*, const char * *host_name*, unsigned int * *node_id*)

Store at *node_id the node ID of the host containing the network interface card with GM host name host_name. (Details).

8.1.6 Variable Documentation

8.1.6.1 GM_ENTRY_POINT const unsigned char gm_log2_roundup_table[257]

Log 2 roundup table. (Details).

8.2 gm_abort.c File Reference

```
#include "gm_config.h"
#include "gm_debug.h"
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT void gm_abort ()

8.2.1 Detailed Description

8.2.2 Function Documentation

8.2.2.1 GM_ENTRY_POINT void gm_abort (void)

gm_abort() aborts the current process, and is a wrapper around the system function abort().

See also:
gm_init gm_open gm_close gm_exit gm_finalize

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.3 gm_alloc_pages.c File Reference

```
#include "gm_call_trace.h"
#include "gm_debug.h"
#include "gm_internal.h"
#include "gm_malloc_debug.h"
```

Functions

- GM_ENTRY_POINT void * gm_alloc_pages (gm_size_t alloc_len)
- GM_ENTRY_POINT void gm_free_pages (void *addr, gm_size_t alloc_len)

8.3.1 Detailed Description

8.3.2 Function Documentation

8.3.2.1 GM_ENTRY_POINT void* gm_alloc_pages (gm_size_t *alloc_len*)

gm_alloc_pages() allocates a page-aligned buffer of length ALLOC_LEN, where ALLOC_LEN is a multiple of GM_PAGE_LEN. Any fractional page following the buffer is wasted.

Return values:

ptr Pointer to the allocated buffer.

0 Error occurred.

Parameters:

alloc_len **(IN)** The length of buffer to be allocated.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.3.2.2 GM_ENTRY_POINT void gm_free_pages (void * *addr*, gm_size_t *alloc_len*)

gm_free_pages() frees the pages at **addr**, which were previously allocated with gm_alloc_pages().

Parameters:

addr **(IN)** The address of the buffer to be freed.

alloc_len **(IN)** The length (in bytes) of the buffer to be freed.

See also:

gm_alloc_pages

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.4 gm_alloc_send_token.c File Reference

```
#include "gm_call_trace.h"
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT int gm_alloc_send_token (gm_port_t *p, unsigned int priority)

8.4.1 Detailed Description

8.4.2 Function Documentation

8.4.2.1 GM_ENTRY_POINT int gm_alloc_send_token (gm_port_t * *p*, unsigned int *priority*)

gm_alloc_send_token() allocates a send token of priority **priority** previously freed with gm_free_send_token() and returns 0 if no token is available. Clients may choose to maintain their own send token counts without using this utility function.

Return values:

send_token_cnt

0 Error occurred.

Parameters:

p **(IN)** The GM port on the source/sender GM node from which the communication will be sent.

priority **(IN)** The priority of the message to be sent.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.5 gm_allow_remote_memory_access.c File Reference

```
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT gm_status_t gm_allow_remote_memory_access (struct gm_port *port)

8.5.1 Detailed Description

8.5.2 Function Documentation

8.5.2.1 GM_ENTRY_POINT gm_status_t gm_allow_remote_memory_access (struct gm_port * *port*)

gm_allow_remote_memory_access() allows 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.

Return values:

GM_SUCCESS Operation completed successfully.

Parameters:

port **(IN)** Handle to the GM port.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.6 gm_bcopy.c File Reference

```
#include "gm_internal.h"
#include "gm_bcopy_up.h"
```

Functions

- GM_ENTRY_POINT void gm_bcopy (const void *from, void *to, gm_size_t len)

8.6.1 Detailed Description

8.6.2 Function Documentation

8.6.2.1 GM_ENTRY_POINT void gm_bcopy (const void * *from*, void * *to*, gm_size_t *len*)

gm_bcopy() copies len bytes starting at **from** to location **to**. This function does not handle overlapping regions.

Parameters:

from **(IN)** The starting location for the region to be copied.

to **(IN)** The ending location for the region to be copied.

len **(IN)** The length in bytes of the region to be copied.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.7 gm_blocking_receive.c File Reference

```
#include "gm_call_trace.h"
#include "gm_cmp64.h"
#include "gm_compiler.h"
#include "gm_debug.h"
#include "gm_internal.h"
#include "gm_set_alarm.h"
```

Functions

- GM_ENTRY_POINT gm_recv_event_t * gm_blocking_receive (gm_port_t *p)

8.7.1 Detailed Description

8.7.2 Function Documentation

8.7.2.1 GM_ENTRY_POINT gm_recv_event_t* gm_blocking_receive (gm_port_t * *p*)

gm_blocking_receive() blocks until there is a receive event and then returns 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.

Return values:

GM_SUCCESS Operation completed successfully.

GM_NO_RECV_EVENT Handle all flushed alarm events, which the user never needs to know about, and which must not cause my_alarm to be cancelled or reset.

GM_FLUSHED_ALARM_EVENT Intercept my_alarm.

GM_ALARM_EVENT If our alarm went off, block. Be careful to not handle any other alarms at this time since we are not passing the GM_ALARM_EVENT to the user and we want to maintain the semantics that user alarms are called only inside gm_unknown() when it is called by the user.

Parameters:

p **(IN)** The GM port in use.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.8 gm_blocking_receive_no_spin.c File Reference

```
#include "gm_call_trace.h"
#include "gm_cmp64.h"
#include "gm_debug.h"
#include "gm_internal.h"
#include "gm_send_queue.h"
```

Functions

- GM_ENTRY_POINT gm_recv_event_t * gm_blocking_receive_no_spin (gm_port_t *p)

8.8.1 Detailed Description

8.8.2 Function Documentation

8.8.2.1 GM_ENTRY_POINT gm_recv_event_t* gm_blocking_receive_no_spin (gm_port_t * *p*)

gm_blocking_receive_no_spin() 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.

Return values:

GM_SUCCESS Operation completed successfully.

GM_NO_RECV_EVENT

GM_WAKE_REQUEST_EVENT

GM_SLEEP_EVENT

Parameters:

p **(IN)** The GM port in use.

Author:

Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.9 gm_bzero.c File Reference

```
#include "gm_config.h"
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT void gm_bzero (void *ptr, gm_size_t len)

8.9.1 Detailed Description

8.9.2 Function Documentation

8.9.2.1 GM_ENTRY_POINT void gm_bzero (void * *ptr*, gm_size_t *len*)

gm_bzero() clears the **len** bytes of memory starting at **ptr**. This function does not use partword I/O unless it must (for speed, especially when doing PIO), and does not rely on the system bzero() functionality, which may not be safe for PIO mapped memory.

Parameters:

ptr **(IN)** The pointer to the memory location.

len **(IN)** The number of bytes of memory to be bzero'ed.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.10 gm_calloc.c File Reference

```
#include <stdlib.h>
#include "gm_call_trace.h"
#include "gm_internal.h"
#include "gm_malloc_debug.h"
```

Functions

- GM_ENTRY_POINT void * gm_calloc (gm_size_t len, gm_size_t cnt)

8.10.1 Detailed Description

8.10.2 Function Documentation

8.10.2.1 GM_ENTRY_POINT void* gm_calloc (gm_size_t *len*, gm_size_t *cnt*)

gm_calloc() allocates and clears an array of **cnt** elements of length **len**.

Parameters:

len **(IN)** The number of bytes in each element.

cnt **(IN)** The number of elements in the array.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.11 gm_close.c File Reference

```
#include "gm_call_trace.h"
#include "gm_debug.h"
#include "gm_debug_open.h"
#include "gm_dma_malloc.h"
#include "gm_enable_trace.h"
#include "gm_internal.h"
#include "gm_ptr_hash.h"
#include "gm_trace.h"
#include <stdio.h>
```

Functions

- GM_ENTRY_POINT void gm_close (gm_port_t *p)

8.11.1 Detailed Description

8.11.2 Function Documentation

8.11.2.1 GM_ENTRY_POINT void gm_close (gm_port_t * *p*)

gm_close() closes a previously opened port **p**, and frees all resources associated with that port.

Parameters:
p **(IN)** The GM port to be closed.

See also:
gm_open gm_init gm_exit gm_finalize gm_abort

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.12 gm_crc.c File Reference

```
#include "gm_internal.h"
#include "gm_crc32.h"
```

Functions

- GM_ENTRY_POINT unsigned long gm_crc (void *_ptr, gm_size_t len)
- GM_ENTRY_POINT unsigned long gm_crc_str (const char *ptr)

8.12.1 Detailed Description

This file contains the GM API functions, gm_crc() and gm_crc_str(), 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.

8.12.2 Function Documentation

8.12.2.1 GM_ENTRY_POINT unsigned long gm_crc (void * *_ptr*, gm_size_t *len*)

gm_crc() computes a CRC-32 of the indicated range of memory.

Return values:
long

Parameters:
_ptr **(IN)** Pointer to a range of memory.
len **(IN)** The length of the indicated range of memory.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.12.2.2 GM_ENTRY_POINT unsigned long gm_crc_str (const char * *ptr*)

gm_crc_str() computes a CRC-32 for the indicated string.

Return values:
long

Parameters:
ptr **(IN)** Pointer to a string.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.13 gm_datagram_send.c File Reference

```
#include "gm_call_trace.h"
#include "gm_debug.h"
#include "gm_enable_fast_small_send.h"
#include "gm_enable_trace.h"
#include "gm_internal.h"
#include "gm_send_queue.h"
```

Functions

- GM_ENTRY_POINT void gm_datagram_send (gm_port_t *p, void *message, unsigned int size, gm_size_t len, unsigned int priority, unsigned int target_node_id, unsigned int target_port_id, gm_send_completion_callback_t callback, void *context)

8.13.1 Detailed Description

This file contains the GM API function gm_datagram_send().

8.13.2 Function Documentation

8.13.2.1 GM_ENTRY_POINT void gm_datagram_send (gm_port_t * *p*, void * *message*, unsigned int *size*, gm_size_t *len*, unsigned int *priority*, unsigned int *target_node_id*, unsigned int *target_port_id*, gm_send_completion_callback_t *callback*, void * *context*)

gm_datagram_send() queues **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**. If any network error is encountered while sending the packet, the packet is silently and immediately dropped. After the packet has been DMA'ed from host memory, callback(port,context,status) is called inside a user invocation of gm_unknown(), reporting the **status** of the attempted send.

Parameters:

p **(IN)** The GM port on the source/sender GM node from which the communication is being sent.

message **(IN)** The pointer to the data to be communicated.

size **(IN)** The size of the buffer.

len **(IN)** The length in bytes of the array.

priority **(IN)** The priority of the message being communicated.

target_node_id **(IN)** The GM node to which the message is being sent.

target_port_id **(IN)** The GM port on the destination GM node to which the message is being sent.

callback **(IN)** The function called when the send is complete.

context **(IN)** Pointer to an integer or to a structure that is passed to the callback function.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.14 gm_datagram_send_4.c File Reference

```
#include "gm_internal.h"
#include "gm_enable_pio_sends.h"
#include "gm_enable_datagrams.h"
#include "gm_enable_trace.h"
#include "gm_send_queue.h"
```

Functions

- GM_ENTRY_POINT void gm_datagram_send_4 (gm_port_t *p, gm_u32_t data, unsigned int size, unsigned long len, unsigned int priority, unsigned int target_node_id, unsigned int target_port_id, gm_send_completion_callback_t callback, void *context)

8.14.1 Detailed Description

This file contains the GM API function gm_datagram_send_4().

8.14.2 Function Documentation

8.14.2.1 GM_ENTRY_POINT void gm_datagram_send_4 (gm_port_t * *p*, gm_u32_t *data*, unsigned int *size*, unsigned long *len*, unsigned int *priority*, unsigned int *target_node_id*, unsigned int *target_port_id*, gm_send_completion_callback_t *callback*, void * *context*)

gm_datagram_send_4() queues **gm_u32_t 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**. If any network error is encountered while sending the packet, the packet is silently and immediately dropped. After the packet has been DMA'ed from host memory, callback(port,context,status) is called inside a user invocation of gm_unknown(), reporting the **status** of the attempted send.

Parameters:

p **(IN)** The GM port on the source/sender GM node from which the communication is being sent.

data **(IN)** The pointer to the gm_u32_t data to be communicated.

size **(IN)** The size of the buffer.

len **(IN)** The length in bytes of the array.

priority **(IN)** The priority of the message being communicated.

target_node_id **(IN)** The GM node to which the message is being sent.

target_port_id **(IN)** The GM port on the destination GM node to which the message is being sent.

callback **(IN)** The function called when the send is complete.

context **(IN)** Pointer to an integer or to a structure that is passed to the callback function.

See also:
gm_datagram_send

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.15 gm_debug_buffers.c File Reference

```
#include "gm_debug.h"
#include "gm_internal.h"
#include <stdio.h>
```

Functions

- GM_ENTRY_POINT int gm_unregister_buffer (void *addr, int size)
- GM_ENTRY_POINT void gm_register_buffer (void *addr, int size)
- GM_ENTRY_POINT void gm_dump_buffers (void)

8.15.1 Detailed Description

This file contains the GM API functions gm_register_buffer(), gm_unregister_buffer(), gm_dump_buffers().

8.15.2 Function Documentation

8.15.2.1 GM_ENTRY_POINT int gm_unregister_buffer (void * *addr*, int *size*)

gm_unregister_buffer() unregistered a GM buffer.

Return values:

1

0

Parameters:

addr **(IN)** Address of data to be freed.

size **(IN)** Size of buffer to be freed.

Author:

??

Version:

GM_API_VERSION (as defined in gm.h)

8.15.2.2 GM_ENTRY_POINT void gm_register_buffer (void * *addr*, int *size*)

gm_register_buffer() registered a GM buffer.

Return values:

1

0

Parameters:

addr **(IN)** Address of data to be registered.

size **(IN)** Size of buffer to be registered.

Author:

??

Version:

GM_API_VERSION (as defined in gm.h)

8.15.2.3 GM_ENTRY_POINT void gm_dump_buffers (void)

gm_dump_buffers()

Author:

??

Version:

GM_API_VERSION (as defined in gm.h)

8.15.3 Variable Documentation

8.15.3.1 GM_ENTRY_POINT char* gm_buf_status_name[ ]

Initial value:

```
 {
  "gm_in_send",
  "gm_in_recv",
  "gm_in_app",
  "gm_invalid_status",
}
```

8.16 gm_deregister.c File Reference

```
#include "gm_call_trace.h"
#include "gm_compiler.h"
#include "gm_debug.h"
#include "gm_debug_mem_register.h"
#include "gm_internal.h"
#include "gm_io.h"
#include "gm_lanai_command.h"
```

Functions

- GM_ENTRY_POINT gm_status_t gm_deregister_memory (gm_port_t *p, void *ptr, gm_size_t length)

8.16.1 Detailed Description

This file contains the GM API function gm_deregister_memory().

8.16.2 Function Documentation

8.16.2.1 GM_ENTRY_POINT gm_status_t gm_deregister_memory (gm_port_t * *p*, void * *ptr*, gm_size_t *length*)

gm_deregister_memory() deregisters **len** bytes of user virtual that were previously registered for DMA transfers with a matching call to gm_register_memory() using **pvma**.

Return values:

GM_SUCCESS Operation completed successfully.

GM_FAILURE

Parameters:

p **(IN)** The GM port in use.

pvma **(IN)** The pvma used to register the memory to now be deregistered.

length **(IN)** The number of bytes in the array to be deregistered.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.17 gm_directcopy.c File Reference

```
#include "gm_debug.h"
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT gm_status_t gm_directcopy_get (struct gm_port *p, void *source_addr, void *target_addr, gm_size_t length, unsigned int source_instance_id, unsigned int source_port_id)

8.17.1 Detailed Description

This file contains the GM API function gm_directcopy_get().

8.17.2 Function Documentation

8.17.2.1 GM_ENTRY_POINT gm_status_t gm_directcopy_get (struct gm_port * *p*, void * *source_addr*, void * *target_addr*, gm_size_t *length*, unsigned int *source_instance_id*, unsigned int *source_port_id*)

gm_directcopy_get() copies data of length **length** bytes, specified at the address **source_addr** of the local process using the port **port_id** of the board **source_instance_id** to the memory area specified at the address **target_addr** of the current process. This implementation bypasses all of the protection of the operating system to provide a memory copy from one process's memory space to another one. The memory areas must have been registered by GM prior to calling this function in order to lock memory pages at their physical locations. There are no alignment or length constraints but the maximum performance will be reached with aligned addresses on both sides. This function is supported exclusively on Linux.

Return values:

GM_SUCCESS Operation completed successfully.

GM_FAILURE Error occurred.

Parameters:

p **(IN)** Handle to the GM port.

source_addr **(IN)** Address of the data to be copied.

target_addr **(IN)** Target address of the copied data.

length **(IN)** The length (in bytes) of the area to be copied.

source_instance_id **(IN)** The id of the interface.

source_port_id **(IN)** The port id of the interface.

Author:
Patrick Geoffray

Version:
GM_API_VERSION (as defined in gm.h)

8.18 gm_directed_send.c File Reference

```
#include "gm.h"
#include "gm_debug.h"
#include "gm_enable_put.h"
#include "gm_internal.h"
#include "gm_send_queue.h"
```

8.18.1 Detailed Description

This file contains the GM API function gm_directed_send().

8.19 gm_dma_calloc.c File Reference

```
#include "gm_internal.h"
#include "gm_call_trace.h"
```

Functions

- GM_ENTRY_POINT void * gm_dma_calloc (struct gm_port *p, gm_size_t count, gm_size_t length)

8.19.1 Detailed Description

This file contains the GM API function gm_dma_calloc().

8.19.2 Function Documentation

8.19.2.1 GM_ENTRY_POINT void* gm_dma_calloc (struct gm_port * *p*, gm_size_t *count*, gm_size_t *length*)

gm_dma_calloc() allocates and clears **count** * **length** bytes of DMAable memory aligned on a 4-byte boundary. Memory should be freed using gm_dma_free().

Parameters:

p **(IN)** Handle to the GM port.

count **(IN)** The number of elements to be calloc'ed.

length **(IN)** The size of each element to be calloc'ed.

See also:

gm_dma_malloc gm_dma_free

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.20 gm_dma_malloc.c File Reference

```
#include "gm_call_trace.h"
#include "gm_compiler.h"
#include "gm_debug.h"
#include "gm_debug_mem_register.h"
#include "gm_internal.h"
#include "gm_malloc_debug.h"
#include "gm_register_recv_queue.h"
```

Functions

- GM_ENTRY_POINT void gm_dma_free (gm_port_t *p, void *addr)
- GM_ENTRY_POINT void * gm_dma_malloc (struct gm_port *p, gm_size_t length)

8.20.1 Detailed Description

This file includes source for the user-level API calls gm_dma_malloc() and gm_dma_free().

8.20.2 Function Documentation

8.20.2.1 GM_ENTRY_POINT void gm_dma_free (gm_port_t * *p*, void * *addr*)

gm_dma_free() frees **p**, 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() or gm_dma_malloc().

Parameters:

p **(IN)** The GM port.

addr **(IN)** The address of the memory to be freed.

See also:
gm_dma_calloc gm_dma_malloc

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.20.2.2 GM_ENTRY_POINT void* gm_dma_malloc (struct gm_port * *p*, gm_size_t *length*)

gm_dma_malloc() allocates **length** bytes of DMAable memory aligned on a 4-byte boundary. Memory should be freed using gm_dma_free().

Parameters:
p **(IN)** Handle to the GM port.

length **(IN)** The number of bytes to be malloc'ed.

See also:
gm_dma_calloc gm_dma_free

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.21 gm_drop_sends.c File Reference

```
#include "gm_internal.h"
#include "_gm_modsend.h"
```

Functions

- GM_ENTRY_POINT 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)

8.21.1 Detailed Description

This file contains the GM API function gm_drop_sends().

8.21.2 Function Documentation

8.21.2.1 GM_ENTRY_POINT 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*)

gm_drop_sends() drops 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. The first four parameters must match those of the failed send. It should be called only once per reported error. This function requires a send token, which will be returned to the client in the callback function. The dropped sends will then be returned to the client with a status of GM_SEND_DROPPED.

gm_drop_sends() and gm_resume_sending(), as most gm requests, require a send token, and the callback you give to them is just meant to return this token. These gm_requests always succeed (if called in a valid manner), so the callback will **always** be called with **GM_SUCCESS** (which here does not mean at all that something was sent successfully, just that the request has been taken into account, and the token used for that request was recycled).

Parameters:

port **(IN)** The handle to the GM port.

priority **(IN)** The priority of the message.

target_node_id **(IN)** The GM node to which the message is being sent.

target_port_id **(IN)** The GM port on the destination GM node to which the message was sent.

callback **(IN)** The function called when the send is complete.

context **(IN)** Pointer to an integer or to a structure that is passed to the callback function.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.22 gm_eprintf.c File Reference

```
#include "gm_config.h"
#include "gm.h"
#include "gm_debug.h"
```

Functions

- GM_ENTRY_POINT int gm_eprintf (const char *format,...)

8.22.1 Detailed Description

8.22.2 Function Documentation

8.22.2.1 GM_ENTRY_POINT int gm_eprintf (const char * *format*, ...)

gm_eprintf() emulates or invokes the ANSI standard vprintf() function.

Return values:
0 Operation completed successfully.

Parameters:
format Specifies how the variable-length arguments are converted for output.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.23 gm_exit.c File Reference

```
#include "gm_call_trace.h"
#include "gm_debug.h"
#include "gm_internal.h"
#include "gm_malloc_debug.h"
```

Functions

- GM_ENTRY_POINT void gm_exit (gm_status_t status)

8.23.1 Detailed Description

8.23.2 Function Documentation

8.23.2.1 GM_ENTRY_POINT void gm_exit (gm_status_t *status*)

gm_exit() causes the current process to exit with a status appropriate to the GM status code **status**.

Parameters:
status **(IN)** The GM status code, as specified in gm.h.

See also:
gm_init gm_open gm_close gm_finalize gm_abort

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.24 gm_flush_alarm.c File Reference

```
#include "gm_call_trace.h"
#include "gm_internal.h"
#include "gm_send_queue.h"
```

Functions

- GM_ENTRY_POINT void gm_flush_alarm (gm_port_t *p)

8.24.1 Detailed Description

This file contains the GM API function gm_flush_alarm().

8.24.2 Function Documentation

8.24.2.1 GM_ENTRY_POINT void gm_flush_alarm (gm_port_t * *p*)

gm_flush_alarm() flushes the alarm queue.

Parameters:
p **(IN)** The GM port.

See also:
gm_set_alarm gm_initialize_alarm gm_cancel_alarm

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.25 gm_free.c File Reference

```
#include <stdlib.h>
#include "gm_call_trace.h"
#include "gm_debug_malloc.h"
#include "gm_internal.h"
#include "gm_malloc_debug.h"
```

Functions

- GM_ENTRY_POINT void gm_free (void *ptr)

8.25.1 Detailed Description

This file contains the GM API function gm_free().

8.25.2 Function Documentation

8.25.2.1 GM_ENTRY_POINT void gm_free (void * *ptr*)

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

Parameters:
ptr **(IN)** Address of the memory to be freed.

See also:
gm_malloc gm_calloc

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.26 gm_free_send_token.c File Reference

```
#include "gm_internal.h"
#include "gm_send_queue.h"
```

Functions

- GM_ENTRY_POINT void gm_free_send_token (gm_port_t *p, unsigned priority)

8.26.1 Detailed Description

This file contains the GM API function gm_free_send_token().

8.26.2 Function Documentation

8.26.2.1 GM_ENTRY_POINT void gm_free_send_token (gm_port_t * *p*, unsigned *priority*)

gm_free_send_token() increments a free send token 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.

Parameters:

p

priority

See also:

gm_alloc_send_token gm_free_send_tokens

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.27 gm_free_send_tokens.c File Reference

```
#include "gm_internal.h"
#include "gm_send_queue.h"
```

Functions

- GM_ENTRY_POINT void gm_free_send_tokens (gm_port_t *p, unsigned priority, unsigned count)

8.27.1 Detailed Description

This file contains the GM API function gm_free_send_tokens().

8.27.2 Function Documentation

8.27.2.1 GM_ENTRY_POINT void gm_free_send_tokens (gm_port_t * *p*, unsigned *priority*, unsigned *count*)

gm_free_send_tokens() 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.

Parameters:

p

priority

count

See also:

gm_alloc_send_token gm_free_send_token

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.28 gm_get.c File Reference

```
#include "gm_call_trace.h"
#include "gm_debug.h"
#include "gm_enable_get.h"
#include "gm_internal.h"
#include "gm_send_queue.h"
```

Functions

- GM_ENTRY_POINT void gm_get (gm_port_t *p, gm_remote_ptr_t remote_buffer, void *local_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)

8.28.1 Detailed Description

This file contains the GM API function gm_get().

8.28.2 Function Documentation

8.28.2.1 GM_ENTRY_POINT void gm_get (gm_port_t * *p*, gm_remote_ptr_t *remote_buffer*, void * *local_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*)

gm_get() performs an RDMA Read operation.

gm_get() transfers the **len** bytes at **remote_buffer** to **target_port_id** on **target_node_id** with priority **priority** and stores the data at the local virtual memory address **local_buffer**. Call callback(port,context,status) when the receive completes or fails, with **status** indicating the status of the receive.

Parameters:

p **(IN)** The GM port on the destination GM node to which the communication is being received.

remote_buffer **(IN)** Address of the remote buffer.

local_buffer **(OUT)** Address of the local buffer.

len **(IN)** The length in bytes of the buffer to be received.

priority **(IN)** The priority of the data being received.

target_node_id **(IN)** The GM node to which the data is being received.

target_port_id **(IN)** The GM port on the destination GM node to which the message is being received.

callback **(IN)** The function called when the receive is complete.

context **(IN)** Pointer to an integer or to a structure that is passed to the callback function.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.29 gm_get_host_name.c File Reference

```
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT gm_status_t gm_get_host_name (gm_port_t *port, char name[GM_MAX_HOST_NAME_LEN])

8.29.1 Detailed Description

This file contains the GM API function gm_get_host_name().

8.29.2 Function Documentation

8.29.2.1 GM_ENTRY_POINT gm_status_t gm_get_host_name (gm_port_t * *port*, char *name*[GM_MAX_HOST_NAME_LEN])

gm_get_host_name() copies the host name of the local node to **name**.

Return values:

GM_SUCCESS Operation completed successfully.

Parameters:

port **(IN)** The GM port.

name **(OUT)** The host name of the GM node.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.30 gm_get_mapper_unique_id.c File Reference

```
#include "gm_internal.h"
```

Functions

- gm_status_t gm_get_mapper_unique_id (struct gm_port *port, char *id)

8.30.1 Detailed Description

This file contains the GM API function gm_get_mapper_unique_id().

8.30.2 Function Documentation

8.30.2.1 gm_status_t gm_get_mapper_unique_id (struct gm_port * *port*, char * *id*)

gm_get_mapper_unique_id() copies the 6-byte ethernet address of the network interface card associated with **port** to the buffer at **id**.

Return values:

GM_SUCCESS Operation completed successfully.

Parameters:

port **(IN)** The GM port associated with a specific network interface card.

id **(OUT)** Buffer to which the 6-byte ethernet address (GM_GET_MAPPER_-UNIQUE_ID) is copied.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.31 gm_get_node_id.c File Reference

```
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT gm_status_t gm_get_node_id (gm_port_t *port, unsigned int *node_id)

8.31.1 Detailed Description

This file contains the GM API function gm_get_node_id.c

8.31.2 Function Documentation

8.31.2.1 GM_ENTRY_POINT gm_status_t gm_get_node_id (gm_port_t * *port*, unsigned int * *node_id*)

gm_get_node_id() copies the GM ID of the network interface card associated with **port** to the address **node_id**.

Return values:

GM_SUCCESS Operation completed successfully

GM_NODE_ID_NOT_YET_SET The GM node ID has not been set.

Parameters:

port **(IN)** The GM port.

node_id **(OUT)** Address to copy the GM ID.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.32 gm_get_node_type.c File Reference

```
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT gm_status_t gm_get_node_type (gm_port_t *port, int *node_type)

8.32.1 Detailed Description

This file contains the GM API function gm_get_node_type().

8.32.2 Function Documentation

8.32.2.1 GM_ENTRY_POINT gm_status_t gm_get_node_type (gm_port_t * *port*, int * *node_type*)

gm_get_node_type() does something.

Return values:

GM_SUCCESS Operation completed successfully.

Parameters:

port **(IN)** The GM port.

node_type **(OUT)** GM_GET_NODE_TYPE, the Big Endian, Little Endianness???.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.33 gm_get_port_id.c File Reference

```
#include "gm_compiler.h"
#include "gm_debug.h"
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT unsigned int gm_get_port_id (gm_port_t *p)

8.33.1 Detailed Description

8.33.2 Function Documentation

8.33.2.1 GM_ENTRY_POINT unsigned int gm_get_port_id (gm_port_t * *p*)

gm_get_port_id() returns the id of the GM port **p**.

Return values:
port_id The id of the GM port.

Parameters:
p **(IN)** The GM port.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.34 gm_get_unique_board_id.c File Reference

```
#include "gm_internal.h"
```

Functions

- gm_status_t gm_get_unique_board_id (gm_port_t *port, char unique[6])

8.34.1 Detailed Description

This file contains the GM API function gm_get_unique_board_id().

8.34.2 Function Documentation

8.34.2.1 gm_status_t gm_get_unique_board_id (gm_port_t * *port*, char *unique*[6])

gm_get_unique_board_id() copies the 6-byte MAC address of the network interface card associated with **port** to the buffer at **unique**.

Return values:

GM_SUCCESS Operation completed successfully.

Parameters:

port **(IN)** The GM port associated with a specific network interface card.

unique **(OUT)** Buffer to which the 6-byte MAC address (GM_GET_UNIQUE_BOARD_ID) is copied.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.35 gm_getpid.c File Reference

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

Functions

- gm_pid_t gm_getpid (void)

8.35.1 Detailed Description

This file contains the GM API function gm_getpid().

8.35.2 Function Documentation

8.35.2.1 gm_pid_t gm_getpid (void)

gm_getpid() is a cover for the usual Unix getpid() functionality.

Return values:
gm_pid_t The process ID of the parent of the current process.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.36 gm_handle_sent_tokens.c File Reference

```
#include "gm_debug.h"
#include "gm_internal.h"
```

8.36.1 Detailed Description

This file contains the GM API function gm_handle_sent_tokens() and is deprecated and included only for backward compatibility. Use gm_unknown() instead.

8.37 gm_hash.c File Reference

```
#include "gm_call_trace.h"
#include "gm_compiler.h"
#include "gm_crc32.h"
#include "gm_debug.h"
#include "gm_internal.h"
```

Data Structures

- struct gm_hash_entry
- struct gm_hash_segment
- struct gm_hash

Typedefs

- typedef gm_hash_entry gm_hash_entry_t
- typedef gm_hash_segment gm_hash_segment_t
- typedef gm_hash gm_hash_t

Functions

- GM_ENTRY_POINT struct gm_hash * gm_create_hash (long(*gm_user_compare)(void *key1, void *key2), unsigned long(*gm_user_hash)(void *key), gm_size_t key_len, gm_size_t data_len, gm_size_t min_cnt, int flags)
- GM_ENTRY_POINT void gm_destroy_hash (struct gm_hash *hash)
- GM_ENTRY_POINT void * gm_hash_remove (gm_hash_t *hash, void *key)
- GM_ENTRY_POINT void * gm_hash_find (gm_hash_t *hash, void *key)
- GM_ENTRY_POINT void gm_hash_rekey (gm_hash_t *hash, void *old_key, void *new_key)
- GM_ENTRY_POINT gm_status_t gm_hash_insert (gm_hash_t *hash, void *key, void *data)
- GM_ENTRY_POINT long gm_hash_compare_strings (void *key1, void *key2)
- GM_ENTRY_POINT unsigned long gm_hash_hash_string (void *key)
- GM_ENTRY_POINT long gm_hash_compare_longs (void *key1, void *key2)
- GM_ENTRY_POINT unsigned long gm_hash_hash_long (void *key)
- GM_ENTRY_POINT long gm_hash_compare_ints (void *key1, void *key2)

- GM_ENTRY_POINT unsigned long gm_hash_hash_int (void *key)
- GM_ENTRY_POINT long gm_hash_compare_ptrs (void *key1, void *key2)
- GM_ENTRY_POINT unsigned long gm_hash_hash_ptr (void *key)

8.37.1 Detailed Description

This file contains the GM API functions gm_create_hash(), gm_destroy_hash(), gm_hash_remove(), gm_hash_find(), gm_hash_insert(), gm_hash_rekey(), gm_hash_compare_strings(), gm_hash_hash_string(), gm_hash_compare_longs(), gm_hash_hash_long(), gm_hash_compare_ints(), gm_hash_hash_int(), gm_hash_compare_ptr(), gm_hash_hash_ptr().

This module implements a generic hash table. It uses lookaside lists to ensure efficient memory allocation in the kernel.

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.

8.37.2 Typedef Documentation

8.37.2.1 typedef struct gm_hash_entry gm_hash_entry_t

A hash table entry.

8.37.2.2 typedef struct gm_hash_segment gm_hash_segment_t

Structure representing a segment of allocated hash table bins. To double the size of the hash table, we allocate a new segment with just enough bins to double the number of bins in the hash table an prepend it to the list of hash segments. This way, we don't have to double-buffer the hash table while growing it, and we can grow the table closer to the limits of available memory. While we sometimes have to walk the O(log(N))-segment list to find a bin, the average lookup only looks at O(2) segments, so operations are still constant-average-time as expected for hash tables.

8.37.2.3 typedef struct gm_hash gm_hash_t

The state of a GM hash table, referenced by the client only using opaque pointers.

8.37.3 Function Documentation

8.37.3.1 GM_ENTRY_POINT struct gm_hash* gm_create_hash (long(* *gm_user_compare*)(void *key1, void *key2), unsigned long(* *gm_user_hash*)(void *key), gm_size_t *key_len*, gm_size_t *data_len*, gm_size_t *min_cnt*, int *flags*)

gm_create_hash() returns a newly-created 'gm_hash' structure or '0' if the hash table could not be created.

Return values:

gm_hash Handle to the hash table.

Parameters:

gm_user_compare **(IN)** 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_user_hash **(IN)** 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 **(IN)** 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 **(IN)** 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 **(IN)** Specifies the number of entries for which storage should be preallocated.

flags **(IN)** Should be '0' because no flags are currently defined.

See also:

gm_destroy_hash

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.37.3.2 GM_ENTRY_POINT void gm_destroy_hash (struct gm_hash * *hash*)

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

Parameters:

hash **(IN)** Handle to the hash table.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.37.3.3 GM_ENTRY_POINT void* gm_hash_remove (gm_hash_t * *hash*, void * *key*)

gm_hash_remove() 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.

Parameters:
hash **(IN)** Pointer to the hash table.

key **(IN)** The key for the hash table.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.37.3.4 GM_ENTRY_POINT void* gm_hash_find (gm_hash_t * *hash*, void * *key*)

gm_hash_find() 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.

Parameters:
hash **(IN)** Pointer to the hash table.

key **(IN)** The key for the hash table.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.37.3.5 GM_ENTRY_POINT void gm_hash_rekey (gm_hash_t * *hash*, void * *old_key*, void * *new_key*)

gm_hash_rekey() 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.

Parameters:

hash **(IN)** Pointer to the hash table.

old_key **(IN)** The previous key for the hash table.

new_key **(IN)** The new key for the hash table.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.37.3.6 GM_ENTRY_POINT gm_status_t gm_hash_insert (gm_hash_t * *hash*, void * *key*, void * *data*)

gm_hash_insert() 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.

Return values:

GM_SUCCESS Operation completed successfully.

Parameters:

hash **(IN)** Pointer to the hash table.

key **(IN)** The key for the hash table.

data **(IN)** Data to be inserted.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.37.3.7 GM_ENTRY_POINT long gm_hash_compare_strings (void * *key1*, void * *key2*)

gm_hash_compare_strings() is the function used to compare two strings.

Return values:
long

Parameters:
key1 **(IN)** The key for the first string.

key2 **(IN)** The key for the second string.

See also:
gm_hash_compare_ints gm_hash_compare_longs gm_hash_compare_ptrs

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.37.3.8 GM_ENTRY_POINT unsigned long gm_hash_hash_string (void * *key*)

gm_hash_hash_string() is the function used to hash keys.

Return values:
long

Parameters:
key **(IN)** The key for the string in the hash table.

See also:
gm_hash_compare_string gm_hash_hash_int gm_hash_hash_long gm_hash_hash_ptr

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.37.3.9 GM_ENTRY_POINT long gm_hash_compare_longs (void * *key1*, void * *key2*)

gm_hash_compare_longs() is the function used to compare two longs.

Return values:
long

Parameters:
key1 **(IN)**
key2 **(IN)**

See also:
gm_hash_compare_ints gm_hash_compare_strings gm_hash_compare_ptrs

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.37.3.10 GM_ENTRY_POINT unsigned long gm_hash_hash_long (void * *key*)

gm_hash_hash_long() is the function used to hash keys.

Return values:
long

Parameters:
key **(IN)**

See also:
gm_hash_compare_long gm_hash_hash_int gm_hash_hash_string gm_hash_hash_ptr

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.37.3.11 GM_ENTRY_POINT long gm_hash_compare_ints (void * *key1*, void * *key2*)

gm_hash_compare_ints() is the function used to compare two ints.

Return values:
long

Parameters:
key1 **(IN)** The key for the first int.

key2 **(IN)** The key for the second int.

See also:
gm_hash_compare_longs gm_hash_compare_strings gm_hash_compare_ptrs

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.37.3.12 GM_ENTRY_POINT unsigned long gm_hash_hash_int (void * *key*)

gm_hash_hash_int() is the function used to hash keys.

Return values:
long

Parameters:
key **(IN)**

See also:
gm_hash_compare_int gm_hash_hash_ptr gm_hash_hash_long gm_hash_hash_string

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.37.3.13 GM_ENTRY_POINT long gm_hash_compare_ptrs (void * *key1*, void * *key2*)

gm_hash_compare_ptrs() is the function used to compare two ptrs.

Return values:
long

Parameters:
key1 **(IN)** The key for the first ptr.

key2 **(IN)** The key for the second ptr.

See also:
gm_hash_compare_longs gm_hash_compare_strings gm_hash_compare_ints

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.37.3.14 GM_ENTRY_POINT unsigned long gm_hash_hash_ptr (void * *key*)

gm_hash_hash_ptr() is the function used to hash keys.

Return values:
long

Parameters:
key **(IN)**

See also:
gm_hash_compare_ptr gm_hash_hash_int gm_hash_hash_long gm_hash_hash_string

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.38 gm_hex_dump.c File Reference

```
#include "gm_config.h"
#include "gm_compiler.h"
#include "gm.h"
#include "gm_debug.h"
#include "gm_internal_funcs.h"
```

Functions

- GM_ENTRY_POINT void gm_hex_dump (const void *_ptr, gm_size_t len)

8.38.1 Detailed Description

This file contains the GM API function gm_hex_dump().

8.38.2 Function Documentation

8.38.2.1 GM_ENTRY_POINT void gm_hex_dump (const void * *_ptr*, gm_size_t *len*)

gm_hex_dump() prints the hex equivalent of data at **_ptr**.

Parameters:

_ptr **(IN)** Address of the data.

len **(IN)** The length (in bytes) of the data.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.39 gm_host_name_to_node_id.c File Reference

```
#include "gm_call_trace.h"
#include "gm_debug.h"
#include "gm_debug_yp.h"
#include "gm_internal.h"
#include "gm_lanai_command.h"
```

Functions

- GM_ENTRY_POINT gm_status_t gm_host_name_to_node_id_ex (gm_port_t *port, unsigned int timeout_usecs, const char *host_name, unsigned int *node_id)
- GM_ENTRY_POINT unsigned int gm_host_name_to_node_id (gm_port_t *port, char *host_name)

8.39.1 Detailed Description

The file containing the GM API function gm_host_name_to_node_id().

8.39.2 Function Documentation

8.39.2.1 GM_ENTRY_POINT gm_status_t gm_host_name_to_node_id_ex (gm_port_t * *port*, unsigned int *timeout_usecs*, const char * *host_name*, unsigned int * *node_id*)

gm_host_name_to_node_id_ex() translates a GM host name to a node ID.

Return values:

GM_SUCCESS Operation completed successfully.

Parameters:

port **(IN)** The GM port associated with the specific GM ID.

timeout_usecs **(IN)** The maximum length of time (in microseconds) to query, or 0 to use the default (long) timeout.

host_name **(IN)**

node_id **(OUT)** Where to store the node ID.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.39.2.2 GM_ENTRY_POINT unsigned int gm_host_name_to_node_id (gm_port_t * *port*, char * *host_name*)

This function is deprecated. Use gm_host_name_to_node_id_ex() instead.

gm_host_name_to_node_id() returns the GM ID associated **host_name** or GM_NO_SUCH_NODE_ID in case of an error.

Return values:
GM_SUCCESS Operation completed successfully.

GM_NO_SUCH_NODE_ID Error.

Parameters:
port **(IN)** The GM port associated with the specific GM ID.

host_name **(OUT)** Where to store the host name. At most GM_MAX_HOST_NAME_LEN+1 bytes will be written.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.40 gm_init.c File Reference

```
#include "gm_call_trace.h"
#include "gm_crc32.h"
#include "gm_debug.h"
#include "gm_enable_security.h"
#include "gm_internal.h"
```

Functions

- gm_status_t gm_init ()
- void gm_finalize ()

8.40.1 Detailed Description

This file contains the GM API functions gm_init() and gm_finalize().

8.40.2 Function Documentation

8.40.2.1 gm_status_t gm_init (void)

gm_init() initializes GM. It increments the GM initialization counter and initializes 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().

Return values:

GM_SUCCESS Operation completed successfully.

GM_FAILURE Error occurred.

See also:

gm_finalize gm_open gm_close gm_exit gm_abort

Author:

Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.40.2.2 void gm_finalize (void)

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

See also:
gm_init gm_open gm_close gm_abort gm_exit

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.41 gm_isprint.c File Reference

```
#include "gm_internal.h"
```

Functions

- int gm_isprint (int c)

8.41.1 Detailed Description

The file containing the GM API function gm_isprint().

8.41.2 Function Documentation

8.41.2.1 int gm_isprint (int *c*)

gm_isprint() is just like ANSI isprint(), only it works in the kernel and MCP.

Return values:

int The return value is nonzero if the character is a printable character including a space, and a zero value if not.

Parameters:

c **(OUT)** Printable character.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.42 gm_log2.c File Reference

```
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT unsigned long gm_log2_roundup (unsigned long n)

Variables

- GM_ENTRY_POINT const unsigned char gm_log2_roundup_table [257]

8.42.1 Detailed Description

This file contains the GM API functions related to base 2 logarithmic computations.

gm_log2_roundup_table, gm_log2_roundup().

8.42.2 Function Documentation

8.42.2.1 GM_ENTRY_POINT unsigned long gm_log2_roundup (unsigned long *n*)

gm_log2_roundup() returns the logarithm, base 2, of **n**, rounding up to the next integer.

Return values:
LOG

Parameters:
n **(IN)** The integer for which the logarithm will be computed.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.42.3 Variable Documentation

8.42.3.1 GM_ENTRY_POINT const unsigned char gm_log2_roundup_table[257]

Initial value:

```
 {
  0,
  0, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4,
  5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
  6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
  6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
  7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
  7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
  7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
  7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
  8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
  8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
  8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
  8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
  8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
  8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
  8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
  8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
}
```

Log 2 roundup table. (Details).

8.43 gm_lookaside.c File Reference

```
#include "gm.h"
#include "gm_cache_line.h"
#include "gm_call_trace.h"
#include "gm_debug.h"
#include "gm_internal.h"
#include "gm_malloc_debug.h"
#include "gm_struct_lock.h"
```

Data Structures

- struct gm_lookaside_segment
- struct gm_lookaside
- struct gm_lookaside_segment_list

Functions

- GM_ENTRY_POINT void * gm_lookaside_alloc (struct gm_lookaside *l)
- GM_ENTRY_POINT void * gm_lookaside_zalloc (struct gm_lookaside *l)
- GM_ENTRY_POINT void gm_lookaside_free (void *ptr)
- GM_ENTRY_POINT struct gm_lookaside * gm_create_lookaside (gm_size_t entry_len, gm_size_t min_entry_cnt)
- GM_ENTRY_POINT void gm_destroy_lookaside (struct gm_lookaside *l)

8.43.1 Detailed Description

This file contains the GM API functions gm_create_lookaside(), gm_destroy_lookaside(), gm_lookaside_alloc(), gm_lookaside_zalloc(), and gm_lookaside_free().

This file implements a lookaside list. It is mainly intended to allow efficient memory allocation of small structures in kernels where the minimum memory allocation is a page, but is not kernel-specific.

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_alloc() returns cache-line-aligned buffers, in an attempt to minimize the cost of accessing data stored in the buffers.

8.43.2 Function Documentation

8.43.2.1 GM_ENTRY_POINT void* gm_lookaside_alloc (struct gm_lookaside * *l*)

gm_lookaside_alloc() allocates an entry from the lookaside table, with debugging. It returns a buffer of size ENTRY_LEN specified when the entry list L was created, or '0' if the buffer could not be allocated.

Return values:
ptr Buffer of size ENTRY_LEN.

Parameters:
l **(IN)** Handle to the gm_lookaside list.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.43.2.2 GM_ENTRY_POINT void* gm_lookaside_zalloc (struct gm_lookaside * *l*)

gm_lookaside_zalloc() allocates and clear an entry from the lookaside table.

Return values:
ptr

Parameters:
l **(IN)** Handle to the gm_lookaside list.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.43.2.3 GM_ENTRY_POINT void gm_lookaside_free (void * *ptr*)

gm_lookaside_free() schedules an allocated entry to be freed, and actually performs any scheduled free. It 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.

Parameters:
ptr **(IN)** Pointer to the entry to be freed.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.43.2.4 GM_ENTRY_POINT struct gm_lookaside* gm_create_lookaside (gm_size_t *entry_len*, gm_size_t *min_entry_cnt*)

gm_create_lookaside() returns a newly created lookaside list to be used to allocate blocks of ENTRY_LEN bytes. MIN_ENTRY_CNT entries are preallocated.

Return values:
gm_lookaside Handle to the lookaside table.

Parameters:
entry_len **(IN)**

min_entry_cnt **(IN)**

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.43.2.5 GM_ENTRY_POINT void gm_destroy_lookaside (struct gm_lookaside * *l*)

gm_destroy_lookaside() frees a lookaside list and all associated resources, including any buffers currently allocated from the lookaside list.

Parameters:

l **(IN)** Handle to the lookaside table.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.44 gm_malloc.c File Reference

```
#include <stdlib.h>
#include "gm_call_trace.h"
#include "gm_compiler.h"
#include "gm_internal.h"
#include "gm_malloc_debug.h"
#include "gm_debug_malloc.h"
```

Functions

- GM_ENTRY_POINT void * gm_malloc (gm_size_t len)

8.44.1 Detailed Description

This file contains the GM API function gm_malloc().

8.44.2 Function Documentation

8.44.2.1 GM_ENTRY_POINT void* gm_malloc (gm_size_t *len*)

gm_malloc() returns a pointer to the specified amount of memory. In the kernel, the memory will be nonpageable.

Return values:
ptr Pointer to the specified amount of memory.

Parameters:
len **(IN)** The length in bytes to be malloc'ed.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.45 gm_mark.c File Reference

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

Data Structures

- union gm_mark_reference
- struct gm_mark_set

Typedefs

- typedef gm_mark_reference gm_mark_reference_t

Functions

- GM_ENTRY_POINT gm_status_t gm_mark (struct gm_mark_set *set, gm_mark_t *m)
- GM_ENTRY_POINT int gm_mark_is_valid (struct gm_mark_set *set, gm_mark_t *m)
- GM_ENTRY_POINT void gm_unmark (struct gm_mark_set *set, gm_mark_t *m)
- GM_ENTRY_POINT gm_status_t gm_create_mark_set (struct gm_mark_set **msp, unsigned long cnt)
- GM_ENTRY_POINT void gm_destroy_mark_set (struct gm_mark_set *set)
- GM_ENTRY_POINT void gm_unmark_all (struct gm_mark_set *set)

8.45.1 Detailed Description

This file implements a constant-time system for marking memory locations and verifying that they have not been corrupted. All operations have average execution time of O(1). If enough marks were specified in the call to gm_create_mark_set(), then all operations have worst-case performance of O(1); otherwise, the worst-case performance is O(N) where N is the number of marks in the mark set.

We do not simply use a magic number as a mark, or a mark set identifier as a mark, since this technique can lead to false positives where unitialized marks appear valid. With this system, any localized corruption of a mark or the mark set database will result in the mark being determined to be invalid. False positives in this system require that both the mark and mark set database be corrupted in a consistent way, which is basically impossible.

This implementation was inspired by a challenge from John Regehr, who was forwarding a challenge from a UVA CS professor, paraphrased as follows: Can you implement the functionality of an array where initialization, insertion, removal, and test for presense of an array element each has execution time bounded by a constant, assuming you are initially provided with a pointer to a sufficient amount of uninitialized memory. You are only allowed O(N) storage, where N is the max number of entries." The anwer is "Yes" and a lot of the same tricks are used here.

The GM "mark" API is introduced in 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.

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.

8.45.2 Typedef Documentation

8.45.2.1 typedef union gm_mark_reference gm_mark_reference_t

a reference to a mark, or a free entry. The reference is used to validate the mark. That mark is considered valid if and only if (set->reference[mark->tag] == mark).

8.45.3 Function Documentation

8.45.3.1 GM_ENTRY_POINT gm_status_t gm_mark (struct gm_mark_set * *set*, gm_mark_t * *m*)

gm_mark() adds '*MARK' to SET. Requires O(constant) time if the mark set has pre-allocated resources for the mark. Otherwise, requires O(constant) average time.

Return values:
GM_SUCCESS Operation completed successfully.

Parameters:
set **(IN)** Handle to the GM mark set.

m

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.45.3.2 GM_ENTRY_POINT int gm_mark_is_valid (struct gm_mark_set * *set*, gm_mark_t * *m*)

gm_mark_is_valid() returns nonzero value if '*MARK' is in SET. Requires O(constant) time.

Return values:
int

Parameters:
set **(IN)** Handle to the GM mark set.

m

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.45.3.3 GM_ENTRY_POINT void gm_unmark (struct gm_mark_set * *set*, gm_mark_t * *m*)

gm_unmark() removes '*MARK' from SET. Requires O(constant) time.

Parameters:

set **(IN)** Handle to the GM mark set.

m

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.45.3.4 GM_ENTRY_POINT gm_status_t gm_create_mark_set (struct gm_mark_set ** *msp*, unsigned long *cnt*)

gm_create_mark_set() returns a pointer to a new mark set at SET with enough preallocated resources to support INIT_COUNT. Returns GM_SUCCESS on success. Requires time comparable to malloc().

Return values:

GM_SUCCESS Operation completed successfully.

Parameters:

msp **(IN)** Handle to the GM mark set.

cnt

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.45.3.5 GM_ENTRY_POINT void gm_destroy_mark_set (struct gm_mark_set * *set*)

gm_destroy_mark_set() frees all resources associated with mark set '*SET'. Requires time comparable to free().

Parameters:
set **(IN)** Handle to the GM mark set.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.45.3.6 GM_ENTRY_POINT void gm_unmark_all (struct gm_mark_set * *set*)

gm_unmark_all() unmarks all marks for the mark set, freeing all but the initial number of preallocated mark references.

Removes all marks from SET. Requires O(constant) time.

Parameters:
set **(IN)** Handle to the GM mark set.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.46 gm_max_length_for_size.c File Reference

```
#include "gm_internal.h"
#include "gm_types.h"
```

Functions

- GM_ENTRY_POINT gm_size_t gm_max_length_for_size (unsigned int size)

8.46.1 Detailed Description

This file contains the GM API function gm_max_length_for_size().

8.46.2 Function Documentation

8.46.2.1 GM_ENTRY_POINT gm_size_t gm_max_length_for_size (unsigned int *size*)

gm_max_length_for_size() returns the maximum length of a message that will fit in a GM buffer of size **size**.

Return values:
gm_size_t

Parameters:
size **(IN)** The size of the GM buffer.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.47 gm_max_node_id.c File Reference

```
#include "gm_call_trace.h"
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT gm_status_t gm_max_node_id (gm_port_t *port, unsigned int *n)

8.47.1 Detailed Description

This file contains the GM API function gm_max_node_id().

8.47.2 Function Documentation

8.47.2.1 GM_ENTRY_POINT gm_status_t gm_max_node_id (gm_port_t * *port*, unsigned int * *n*)

gm_max_node_id() stores the maximum GM node ID supported by the network interface card corresponding to **port** at the address of **n**.

Return values:

GM_SUCCESS Operation completed successfully.

Parameters:

port **(IN)**

n **(OUT)** Buffer to store the maximum GM node ID.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.48 gm_max_node_id_in_use.c File Reference

```
#include "gm_call_trace.h"
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT gm_status_t gm_max_node_id_in_use (gm_port_t *port, unsigned int *n)

8.48.1 Detailed Description

This file contains the GM API function gm_max_node_id_in_use().

This function exists to allow gm_board_info to have a guess for an upper limit to the actual number of nodes in the route table Without it, gm_board_info calls thousands of gm_get_route() calls. When debugging, this is really a mess.

8.48.2 Function Documentation

8.48.2.1 GM_ENTRY_POINT gm_status_t gm_max_node_id_in_use (gm_port_t * *port*, unsigned int * *n*)

gm_max_node_id_in_use() returns the maximum GM node ID that is in use by the network attached to the port.

Return values:

GM_SUCCESS Operation completed successfully.

GM_UNATTACHED

Parameters:

port **(IN)** The GM port.

n **(OUT)** Buffer containing the maximum GM node ID.

Author:

Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.49 gm_memcmp.c File Reference

```
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT int gm_memcmp (const void *_a, const void *_b, gm_size_t len)

8.49.1 Detailed Description

This file contains the GM API function gm_memcmp().

8.49.2 Function Documentation

8.49.2.1 GM_ENTRY_POINT int gm_memcmp (const void * *_a*, const void * *_b*, gm_size_t *len*)

gm_memcmp() emulates the ANSI memcmp() function.

Return values:

int Returns an integer less than, equal to, or greater than zero if the first len bytes of a is found, respectively, to be less than, to match, or be greater than the first len bytes of b.

Parameters:

a (IN) The first memory area for comparison.

b (IN) The second memory area for comparison.

len (IN) The number of bytes to compare.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.50 gm_memorize_message.c File Reference

```
#include "gm_debug.h"
#include "gm_internal.h"
```

Functions

- void * gm_memorize_message (void *message, void *buffer, unsigned len)

8.50.1 Detailed Description

This file contains the GM API function gm_memorize_message().

8.50.2 Function Documentation

8.50.2.1 void* gm_memorize_message (void * *message*, void * *buffer*, unsigned *len*)

gm_memorize_message() is a wrapper around function memcpy(). gm_memorize_message() copies a message into a buffer if needed. 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 optionally optimizes the handling of FAST receive messages as described in ”See Chapter 9 [Receiving Messages].

Return values:

something

Parameters:

message **(IN)** Address of the message to be copied.

buffer **(OUT)** Address where the message is to be copied.

len **(IN)** The length in bytes of the message to be copied.

Author:

Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.51 gm_memset.c File Reference

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

Functions

- GM_ENTRY_POINT void * gm_memset (void *s, int c, gm_size_t n)

8.51.1 Detailed Description

8.51.2 Function Documentation

8.51.2.1 GM_ENTRY_POINT void* gm_memset (void * *s*, int *c*, gm_size_t *n*)

gm_memset() reimplements the UNIX function memset().

Return values:

void Returns a pointer to the memory area s.

Parameters:

s (IN) The memory area.

c (IN) The constant byte size.

n (IN) The number of bytes.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.52 gm_min_message_size.c File Reference

```
#include "gm_compiler.h"
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT unsigned int gm_min_message_size (gm_port_t *p)

8.52.1 Detailed Description

This file contains the GM API function gm_min_message_size().

8.52.2 Function Documentation

8.52.2.1 GM_ENTRY_POINT unsigned int gm_min_message_size (gm_port_t * *p*)

gm_min_message_size() returns the minimum supported message size. This value is accessed through the function API to avoid hard-coding it in user applications, allowing dynamic library upgrades.

Return values:
GM_MIN_MESSAGE_SIZE

Parameters:
p

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.53 gm_min_size_for_length.c File Reference

```
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT unsigned int gm_min_size_for_length (gm_size_t length)

8.53.1 Detailed Description

This file contains the GM API function gm_min_size_for_length().

8.53.2 Function Documentation

8.53.2.1 GM_ENTRY_POINT unsigned int gm_min_size_for_length (gm_size_t *length*)

gm_min_size_for_length() returns the minimum GM message buffer size required to store a message of length **length**.

Return values:
gm_log2_roundup

Parameters:
length **(IN)** The length of the message.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.54 gm_mtu.c File Reference

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

Functions

- GM_ENTRY_POINT unsigned int gm_mtu (struct gm_port *port)

8.54.1 Detailed Description

This file contains the GM API function gm_mtu().

8.54.2 Function Documentation

8.54.2.1 GM_ENTRY_POINT unsigned int gm_mtu (struct gm_port * *port*)

gm_mtu() returns the value of GM_MTU.

Return values:
GM_MTU

Parameters:
port **(IN)** The handle to the GM port.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.55 gm_mutex.c File Reference

```
#include "gm_call_trace.h"
#include "gm_compiler.h"
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT struct gm_mutex * gm_create_mutex ()
- GM_ENTRY_POINT void gm_destroy_mutex (struct gm_mutex *mu)
- GM_ENTRY_POINT void gm_mutex_enter (struct gm_mutex *mu)
- GM_ENTRY_POINT void gm_mutex_exit (struct gm_mutex *mu)

8.55.1 Detailed Description

This file contains the GM API functions gm_create_mutex(), gm_destroy_mutex(), gm_mutex_enter(), gm_mutex_exit().

8.55.2 Function Documentation

8.55.2.1 GM_ENTRY_POINT struct gm_mutex* gm_create_mutex (void)

gm_create_mutex() creates a GM mutex.

Return values:
gm_mutex **(OUT)** Handle to the GM mutex.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.55.2.2 GM_ENTRY_POINT void gm_destroy_mutex (struct gm_mutex * *mu*)

gm_destroy_mutex() destroys a GM mutex.

Parameters:
mu **(IN)** The handle to the GM mutex.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.55.2.3 GM_ENTRY_POINT void gm_mutex_enter (struct gm_mutex * *mu*)

gm_mutex_enter()

Parameters:
mu **(IN)** The handle to the GM mutex.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.55.2.4 GM_ENTRY_POINT void gm_mutex_exit (struct gm_mutex * *mu*)

gm_mutex_exit()

Parameters:
mu **(IN)** The handle to the GM mutex.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.56 gm_next_event_peek.c File Reference

```
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT int gm_next_event_peek (gm_port_t *p, gm_u16_t *sender)

8.56.1 Detailed Description

This file contains the GM API function gm_next_event_peek().

8.56.2 Function Documentation

8.56.2.1 GM_ENTRY_POINT int gm_next_event_peek (gm_port_t * *p*, gm_u16_t * *sender*)

gm_next_event_peek() 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.

Return values:

gm_recv_event_t Receive event type.

Parameters:

p **(IN)** The GM port for which the communication is received.

sender

See also:

gm_receive_pending gm_receive

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.57 gm_node_id_to_host_name.c File Reference

```
#include "gm_call_trace.h"
#include "gm_debug.h"
#include "gm_debug_yp.h"
#include "gm_internal.h"
#include "gm_lanai_command.h"
```

Functions

- GM_ENTRY_POINT char * gm_node_id_to_host_name (gm_port_t *port, unsigned int node_id)
- GM_ENTRY_POINT gm_status_t gm_node_id_to_host_name_ex (gm_port_t *port, unsigned int timeout_usecs, unsigned int node_id, char(*name)[GM_MAX_HOST_NAME_LEN+1])

8.57.1 Detailed Description

This file contains the GM API functions gm_node_id_to_host_name() and gm_node_id_to_host_name_ex().

8.57.2 Function Documentation

8.57.2.1 GM_ENTRY_POINT char* gm_node_id_to_host_name (gm_port_t * *port*, unsigned int *node_id*)

This function is deprecated.

gm_node_id_to_host_name() returns a pointer to the host name of the host containing the network interface card with GM node id node_id. The name referenced by the returned pointer is only valid until the next GM API call.

Return values:

host_name The host name of the host containing the GM node id node_id.

Parameters:

port **(IN)** The GM port associated with the network interface card.

node_id **(IN)** The GM node id associated with the network interface card.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.57.2.2 GM_ENTRY_POINT gm_status_t gm_node_id_to_host_name_ex (gm_port_t * *port*, unsigned int *timeout_usecs*, unsigned int *node_id*, char * *name*[GM_MAX_HOST_NAME_LEN+1])

Store at *name the host name of the host containing the network interface card with GM node id node_id. The buffer pointed to by name must have at least GM_MAX_HOST_NAME_LEN+1 bytes of storage.

Return values:

host_name The host name of the host containing the GM node id node_id.

Parameters:

timeout_usecs **(IN)** The maximum length of time (in microseconds) to query, or 0 to use the default (long) timeout.

port **(IN)** The GM port associated with the network interface card.

node_id **(IN)** The GM node id associated with the network interface card.

name **(OUT)** Where to store the pointer to the host name.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.58 gm_node_id_to_unique_id.c File Reference

```
#include "gm_call_trace.h"
#include "gm_internal.h"
#include "gm_global_id.h"
```

Functions

- GM_ENTRY_POINT gm_status_t gm_node_id_to_unique_id (gm_port_t *port, unsigned int node_id, char *uid)

8.58.1 Detailed Description

This file contains the GM API function gm_global_id_to_unique_id().

8.58.2 Function Documentation

8.58.2.1 GM_ENTRY_POINT gm_status_t gm_node_id_to_unique_id (gm_port_t * *port*, unsigned int *node_id*, char * *uid*)

gm_global_id_to_unique_id() stores the MAC address for the interface with GM ID **n** at **unique**.

Return values:

GM_SUCCESS Operation completed successfully.

GM_INVALID_PARAMETER

Parameters:

port **(IN)** A GM port.

node_id **(IN)** The node ID to convert.

uid **(OUT)** Where to store the unique ID corresponding to node_id.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.59 gm_num_ports.c File Reference

```
#include "gm_compiler.h"
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT unsigned int gm_num_ports (gm_port_t *p)

8.59.1 Detailed Description

This file contains the GM API function gm_num_ports().

8.59.2 Function Documentation

8.59.2.1 GM_ENTRY_POINT unsigned int gm_num_ports (gm_port_t * *p*)

gm_num_ports() returns the number of ports supported by this build. Note that this is a build-time value, and the 'p' parameter is actually meaningless at the present time.

This value is accessed through the function API to avoid hard-coding it in user applications, thus allowing dynamic library upgrades.

Return values:
GM_NUM_PORTS

Parameters:
p **(IN)** The GM port.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.60 gm_num_receive_tokens.c File Reference

```
#include "gm_compiler.h"
#include "gm_debug.h"
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT unsigned int gm_num_receive_tokens (gm_port_t *p)

8.60.1 Detailed Description

This file contains the GM API function gm_num_receive_tokens().

8.60.2 Function Documentation

8.60.2.1 GM_ENTRY_POINT unsigned int gm_num_receive_tokens (gm_port_t * p)

gm_num_receive_tokens() returns the number of receive tokens for this port. This value is accessed through the function API to avoid hard-coding it in user applications, allowing dynamic library upgrades.

Return values:
GM_NUM_RECV_TOKENS

Parameters:
p **(IN)** The GM port.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.61 gm_num_send_tokens.c File Reference

```
#include "gm_compiler.h"
#include "gm_debug.h"
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT unsigned int gm_num_send_tokens (gm_port_t *p)

8.61.1 Detailed Description

This file contains the GM API function gm_num_send_tokens().

8.61.2 Function Documentation

8.61.2.1 GM_ENTRY_POINT unsigned int gm_num_send_tokens (gm_port_t * *p*)

gm_num_send_tokens() returns the number of send tokens for this port. This value is accessed through the function API to avoid hard-coding it in user applications, allowing dynamic library upgrades.

Return values:
GM_NUM_SEND_TOKENS

Parameters:
p **(IN)** The GM port.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.62 gm_on_exit.c File Reference

```
#include "gm.h"
#include "gm_call_trace.h"
#include "gm_config.h"
#include "gm_internal_funcs.h"
```

Data Structures

- struct gm_on_exit_record

Typedefs

- typedef gm_on_exit_record gm_on_exit_record_t

Functions

- GM_ENTRY_POINT gm_status_t gm_on_exit (gm_on_exit_callback_t callback, void *arg)

8.62.1 Detailed Description

This file contains the GM API function gm_on_exit().

8.62.2 Typedef Documentation

8.62.2.1 typedef struct gm_on_exit_record gm_on_exit_record_t

List element storing the details of a callback that should be called upon exit.

8.62.3 Function Documentation

8.62.3.1 GM_ENTRY_POINT gm_status_t gm_on_exit (gm_on_exit_callback_t *callback*, void * *arg*)

gm_on_exit() is like Linux on_exit(). This function 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()'.

Call the callbacks in the reverse order registered inside gm_exit(), passing GM exit status and registered argument to the callback.

Return values:

GM_SUCCESS Operation completed successfully.

GM_OUT_OF_MEMORY

Parameters:

callback

arg

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.63 gm_open.c File Reference

```
#include "gm_call_trace.h"
#include "gm_compiler.h"
#include "gm_debug.h"
#include "gm_debug_open.h"
#include "gm_internal.h"
#include "gm_internal_funcs.h"
#include "gm_enable_ethernet.h"
#include "gm_enable_security.h"
#include "gm_enable_trace.h"
#include "gm_ptr_hash.h"
```

Functions

- GM_ENTRY_POINT gm_status_t gm_open (gm_port_t **port_p, unsigned unit, unsigned port_id, const char *client_type, enum gm_api_version api_version)

8.63.1 Detailed Description

The file containing the GM API function gm_open().

8.63.2 Function Documentation

8.63.2.1 GM_ENTRY_POINT gm_status_t gm_open (gm_port_t ** *port_p*, unsigned *unit*, unsigned *port_id*, const char * *client_type*, enum gm_api_version *api_version*)

gm_open() opens a GM port **port_p** for LANai interface **unit**, a pointer to the port's state at ***port_p**. The pointer must be passed to all subsequent functions that operate on the opened port. **port_id** 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 (internal use) and 1 (mapper) and 3 (ethernet emulation) are reserved, so clients should use port 2 and ports 4-7.

gm_open() is to be called by clients other than the daemon and mapper.

Return values:

GM_SUCCESS Operation completed successfully.

GM_INVALID_PARAMETER Invalid parameter passed.

GM_BUSY Could not open device.

GM_NO_SUCH_DEVICE port_id > GM_NUM_PORTS - 1.

GM_INCOMPATIBLE_LIB_AND_DRIVER The GM user library linked with this program may not be compatible with the installed driver.

GM_OUT_OF_MEMORY Could not allocate storage for user port.

Parameters:

port_p **(IN)** Pointer to the handle of the GM port.

unit **(IN)** The device for the Myrinet interface. = 0 if device is myri0.

port_id **(OUT)** The id of the GM port that is opened. = 2, 4, 5, 6, or 7 (Ports 0, 1, and 3 are for privileged use only.)

client_type **(IN)** Unused.

api_version **(IN)** GM_API_VERSION = GM_API_VERSION as defined in gm.h

See also:

gm_close gm_init gm_abort gm_finalize gm_exit

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.64 gm_page_alloc.c File Reference

```
#include "gm_call_trace.h"
#include "gm_debug.h"
#include "gm_internal.h"
#include "gm_malloc_debug.h"
```

Data Structures

- struct gm_free_page
- struct gm_page_allocation_record

Functions

- GM_ENTRY_POINT void * gm_page_alloc ()
- GM_ENTRY_POINT void gm_page_free (void *ptr)

8.64.1 Detailed Description

This file contains gm_page_alloc() and gm_page_free().

These GM API functions allow pages to be allocated and freed.

These modules implement a platform-independent page allocation interface, with automatic initialization and finalization for use in the kernel.

In some cases, this file uses platform-specific features to implement page allocation efficiently, but by default this module allocates blocks of pages and keeps a page free list to recycle freed pages, but the module never frees the blocks of pages until all pages have been freed.

8.64.2 Function Documentation

8.64.2.1 GM_ENTRY_POINT void* gm_page_alloc (void)

gm_page_alloc() returns a ptr to a newly allocated page-aligned buffer of length GM_PAGE_LEN.

Return values:
ptr Page-aligned buffer of length GM_PAGE_LEN.

See also:
gm_page_free

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.64.2.2 GM_ENTRY_POINT void gm_page_free (void * *ptr*)

gm_page_free() frees the page of memory at **ptr** previously allocated by gm_page_alloc(). If all pages have been freed, free all of the memory allocated for pages.

Parameters:
ptr **(IN)** Address of the memory page to be freed.

See also:
gm_page_alloc

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.65 gm_perror.c File Reference

```
#include "gm_debug.h"
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT void gm_perror (const char *message, gm_status_t error)

8.65.1 Detailed Description

This file contains the GM API function gm_perror().

8.65.2 Function Documentation

8.65.2.1 GM_ENTRY_POINT void gm_perror (const char * *message*, gm_status_t *error*)

gm_perror() is similar to ANSI 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**.

Parameters:

message **(OUT)** Textual description of the GM error.

error **(IN)** GM Error code.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.66 gm_printf.c File Reference

```
#include "gm_config.h"
#include "gm.h"
#include <stdio.h>
```

Functions

- GM_ENTRY_POINT int gm_printf (const char *format,...)

8.66.1 Detailed Description

8.66.2 Function Documentation

8.66.2.1 GM_ENTRY_POINT int gm_printf (const char * *format*, ...)

gm_printf() emulates or invokes the ANSI standard printf() function.

Return values:
0 Operation completed successfully.

Parameters:
format Specifies how the arguments are converted for output.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.67 gm_provide_receive_buffer.c File Reference

```
#include "gm_debug.h"
#include "gm_debug_recv_tokens.h"
#include "gm_internal.h"
#include "gm_send_queue.h"
```

Functions

- GM_ENTRY_POINT void gm_provide_receive_buffer_with_tag (gm_port_t *p, void *ptr, unsigned size, unsigned priority, unsigned int tag)

8.67.1 Detailed Description

This file contains the GM API function gm_provide_receive_buffer_with_tag().

8.67.2 Function Documentation

8.67.2.1 GM_ENTRY_POINT void gm_provide_receive_buffer_with_tag (gm_port_t * *p*, void * *ptr*, unsigned *size*, unsigned *priority*, unsigned int *tag*)

gm_provide_receive_buffer_with_tag() 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 a port being closed after a timeout. This timeout is a function of the number of packets sent.

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_with_tag(port,buffer,size,priority,tag) 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_with_tag() 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 a buffer. It may be used in any way the client desires, and need not be unique.

Parameters:

p **(IN)** The GM port.

ptr **(IN)** The address of the message communicated.

size **(IN)** The size of the message.

priority **(IN)** The priority of the message.

tag **(OUT)** The tag for a receive event queue.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.68 gm_put.c File Reference

```
#include "gm_call_trace.h"
#include "gm_debug.h"
#include "gm_enable_put.h"
#include "gm_internal.h"
#include "gm_send_queue.h"
```

Functions

- GM_ENTRY_POINT void gm_put (gm_port_t *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)

8.68.1 Detailed Description

This file contains the GM API function gm_put().

8.68.2 Function Documentation

8.68.2.1 GM_ENTRY_POINT void gm_put (gm_port_t * *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*)

gm_put() transfers the **len** bytes at **source_buffer** to **target_port_id** on **target_node_id** with priority **priority** and stores 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().

Parameters:

p **(IN)** The GM port on the source/sender GM node from which the communication is being sent.

source_buffer **(IN)** Address of the send buffer.

target_buffer **(OUT)** Address of the receive buffer.

len **(IN)** The length in bytes of the buffer to be sent.

priority **(IN)** The priority of the data being sent.

target_node_id **(IN)** The GM node to which the data is being sent.

target_port_id **(IN)** The GM port on the destination GM node to which the message is being sent.

callback **(IN)** The function called when the send is complete.

context **(IN)** Pointer to an integer or to a structure that is passed to the callback function.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.69 gm_rand.c File Reference

```
#include "gm_internal.h"
#include "gm_crc32.h"
```

Functions

- GM_ENTRY_POINT int gm_rand ()
- GM_ENTRY_POINT void gm_srand (int seed)

8.69.1 Detailed Description

This file contains the GM API functions gm_rand() and gm_srand().

8.69.2 Function Documentation

8.69.2.1 GM_ENTRY_POINT int gm_rand (void)

gm_rand() returns a pseudo-random integer, using a poor but fast random number generator.

Return values:
RANDOM_NUMBER The random number that was generated.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.69.2.2 GM_ENTRY_POINT void gm_srand (int *seed*)

gm_srand() returns a pseudo-random integer, and requires a seed for the random number generator.

Parameters:
seed **(IN)** Seed for the random number generator.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.70 gm_rand_mod.c File Reference

```
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT unsigned int gm_rand_mod (unsigned int a)

8.70.1 Detailed Description

8.70.2 Function Documentation

8.70.2.1 GM_ENTRY_POINT unsigned int gm_rand_mod (unsigned int *a*)

gm_rand_mod() returns a pseudo-random number modulo **modulus**, using

a poor but fast random number generator.

Return values:
RANDOM_NUMBER

Parameters:
a **(IN)** The modulus bound.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.71 gm_receive.c File Reference

```
#include "gm_call_trace.h"
#include "gm_debug.h"
#include "gm_debug_recv_tokens.h"
#include "gm_enable_trace.h"
#include "gm_internal.h"
#include "gm_tick.h"
```

Functions

- GM_ENTRY_POINT gm_recv_event_t * gm_receive (gm_port_t *p)
- GM_ENTRY_POINT gm_recv_event_t * gm_receive_debug_buffers (gm_port_t *port)

8.71.1 Detailed Description

This file contains the GM API functions gm_receive() and gm_receive_debug_buffers().

8.71.2 Function Documentation

8.71.2.1 GM_ENTRY_POINT gm_recv_event_t* gm_receive (gm_port_t * *p*)

gm_receive() returns a receive event. If no significant receive event is pending, then an event of type GM_NO_RECV_EVENT is immediately returned.

Return values:
gm_recv_event_t

Parameters:
p **(IN)** The GM port for which the communication is received.

See also:
gm_receive_pending

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.71.2.2 GM_ENTRY_POINT gm_recv_event_t* gm_receive_debug_buffers (gm_port_t * *port*)

gm_receive_debug_buffers()

Return values:
gm_recv_event_t

Parameters:
port **(IN)** The GM port for which the communication is received.

See also:
gm_receive_pending gm_receive

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.72 gm_receive_pending.c File Reference

```
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT int gm_receive_pending (gm_port_t *p)

8.72.1 Detailed Description

This file contains the GM API function gm_receive_pending().

8.72.2 Function Documentation

8.72.2.1 GM_ENTRY_POINT int gm_receive_pending (gm_port_t * *p*)

gm_receive_pending() 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.

Return values:
event.recv.type

Parameters:
p **(IN)** The GM port for which the received communication is pending.

See also:
gm_receive

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.73 gm_register.c File Reference

```
#include "gm_call_trace.h"
#include "gm_compiler.h"
#include "gm_debug.h"
#include "gm_internal.h"
#include "gm_io.h"
#include "gm_debug_mem_register.h"
```

Functions

- GM_ENTRY_POINT gm_status_t gm_register_memory_ex (gm_port_t *p, void *_ptr, gm_size_t length, void *_pvma)
- GM_ENTRY_POINT gm_status_t gm_register_memory (gm_port_t *p, void *ptr, gm_size_t length)

8.73.1 Detailed Description

This file contains the GM API function gm_register_memory().

8.73.2 Function Documentation

8.73.2.1 GM_ENTRY_POINT gm_status_t gm_register_memory_ex (gm_port_t * *p*, void * *_ptr*, gm_size_t *length*, void * *_pvma*)

gm_register_memory_ex() registers **length** bytes of user virtual memory starting at **ptr** for DMA transfers, associating the memory with port virtual address **pvma**. 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 2.7 and earlier due to operating system limitations.

Note that **pvma** must be used in all subsequent GM API calls to refer to the registered memory region.

gm_register_memory(p,ptr,len) is equivalent to gm_register_memory_ex(p,ptr,len,ptr).

Return values:

GM_SUCCESS Operation completed successfully.

GM_FAILURE

GM_PERMISSION_DENIED

GM_INVALID_PARAMETER

GM_OUT_OF_MEMORY

Parameters:

p **(IN)** Handle to the GM port.

ptr **(IN)** The address of the memory region to be registered.

length **(IN)** The length in bytes of the memory region to be registered.

pvma **(IN)** The port virtual memory address with which to associate this region.

See also:

gm_register_memory , gm_deregister_memory

Author:

Glenn Brown

Version:

GM_API_VERSION_2_0_6

8.73.2.2 GM_ENTRY_POINT gm_status_t gm_register_memory (gm_port_t * *p*, void * *ptr*, gm_size_t *length*)

gm_register_memory() registers **length** 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 2.7 and earlier due to operating system limitations.

Return values:

GM_SUCCESS Operation completed successfully.

GM_FAILURE

GM_PERMISSION_DENIED

GM_INVALID_PARAMETER

GM_OUT_OF_MEMORY

Parameters:

p **(IN)** Handle to the GM port.

ptr **(OUT)** The address of the memory location to be registered.

length **(IN)** The length in bytes of the memory location to be registered.

See also:

gm_deregister_memory , gm_register_memory_ex

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.74 gm_resume_sending.c File Reference

```
#include "gm_internal.h"
#include "_gm_modsend.h"
```

Functions

- GM_ENTRY_POINT void gm_resume_sending (struct gm_port *p, unsigned int priority, unsigned int target_node_id, unsigned int target_port_id, gm_send_completion_callback_t callback, void *context)

8.74.1 Detailed Description

The file contains the GM API function gm_resume_sending().

8.74.2 Function Documentation

8.74.2.1 GM_ENTRY_POINT void gm_resume_sending (struct gm_port * *p*, unsigned int *priority*, unsigned int *target_node_id*, unsigned int *target_port_id*, gm_send_completion_callback_t *callback*, void * *context*)

gm_resume_sending(0 reenables 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. The message that generated the error is not resent. The first four parameters must match those of the failed send. It should be called only once per reported error. This function requires a send token, which will be returned to the client in the callback function.

gm_resume_sending() and gm_drop_sends(), as most gm requests, require a send token, and the callback you give to them is just meant to return this token. These gm_requests always succeed (if called in a valid manner), so the callback will **always** be called with **GM_SUCCESS** (which here does not mean at all that something was sent successfully, just that the request has been taken into account, and the token used for that request was recycled).

Parameters:

p **(IN)** The handle to the GM port.

priority **(IN)** The priority of the message being sent.

target_node_id **(IN)** The GM node to which the message is being sent.

target_port_id **(IN)** The GM port on the destination GM node to which the message is being sent.

callback **(IN)** The function called when the send is complete.

context **(IN)** Pointer to an integer or to a structure that is passed to the callback function.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.75 gm_send.c File Reference

```
#include "gm_call_trace.h"
#include "gm_debug.h"
#include "gm_enable_fast_small_send.h"
#include "gm_enable_trace.h"
#include "gm_internal.h"
#include "gm_send_queue.h"
```

Functions

- GM_ENTRY_POINT void gm_send_with_callback (gm_port_t *p, void *message, unsigned int size, gm_size_t len, unsigned int priority, unsigned int target_node_id, unsigned int target_port_id, gm_send_completion_callback_t callback, void *context)

8.75.1 Detailed Description

This file contains the GM API functions gm_send_with_callback().

8.75.2 Function Documentation

8.75.2.1 GM_ENTRY_POINT void gm_send_with_callback (gm_port_t * *p*, void * *message*, unsigned int *size*, gm_size_t *len*, unsigned int *priority*, unsigned int *target_node_id*, unsigned int *target_port_id*, gm_send_completion_callback_t *callback*, void * *context*)

gm_send_with_callback() is a fully asynchronous send. It queues the **message** of length **len** to be sent with priority **priority** to node **target_node_id**. As GM is event-based, the effective completion of the send is notified to the client software by the execution of the callback function specified by **callback**. Before calling gm_send_with_callback(), the client software must first possess a send token of the same priority, and by calling gm_send_with_callback() the client implicitly relinquishes this send token. After a call to gm_send_with_callback(..., 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.

Note: 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.

Parameters:

p **(IN)** A pointer to the GM port on the source/sender GM node over which the message is to be sent.

message **(IN)** A pointer to the data to be sent.

size **(IN)** The size receive buffer in which to store the message on the remote node.

len **(IN)** The length (in bytes) of the message to be sent.

priority **(IN)** The priority with which to send the message ('GM_HIGH_PRIORITY' or 'GM_LOW_PRIORITY').

target_node_id **(IN)** The ID of the GM node to which the message should be sent.

target_port_id **(IN)** The ID of the GM port to which the message should be sent.

callback **(IN)** The client function to call when the send completes.

context **(IN)** A pointer to pass to the CALLBACK function when it is called.

See also:

gm_send_to_peer_with_callback

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.76 gm_send_to_peer.c File Reference

```
#include "gm_internal.h"
#include "gm_enable_fast_small_send.h"
#include "gm_send_queue.h"
```

Functions

- GM_ENTRY_POINT void gm_send_to_peer_with_callback (gm_port_t *p, void *message, unsigned int size, gm_size_t len, unsigned int priority, unsigned int target_node_id, gm_send_completion_callback_t callback, void *context)

8.76.1 Detailed Description

This file contains the GM API function gm_send_to_peer_with_callback().

8.76.2 Function Documentation

8.76.2.1 GM_ENTRY_POINT void gm_send_to_peer_with_callback (gm_port_t * *p*, void * *message*, unsigned int *size*, gm_size_t *len*, unsigned int *priority*, unsigned int *target_node_id*, gm_send_completion_callback_t *callback*, void * *context*)

gm_send_to_peer_with_callback() is an asynchronous send like gm_send_with_callback(), only with the **target_node_id** implicitly set to the same ID as **port** (sending to the same port). This function is marginally faster than gm_send_with_callback().

Parameters:

p **(IN)** A pointer to the GM port over which the message is to be sent.

message **(IN)** A pointer to the data to be sent.

size **(IN)** The size receive buffer in which to store the message on the remote node.

len **(IN)** The length (in bytes) of the message to be sent.

priority **(IN)** The priority with which to send the message ('GM_HIGH_PRIORITY' or 'GM_LOW_PRIORITY').

target_node_id **(IN)** The ID of the GM node to which the message should be sent.

callback **(IN)** The client function to call when the send completes.

context **(IN)** A pointer to pass to the CALLBACK function when it is called.

See also:
gm_send_with_callback

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.77 gm_send_token_available.c File Reference

```
#include "gm_internal.h"
```

Functions

- int gm_send_token_available (gm_port_t *p, unsigned priority)

8.77.1 Detailed Description

This file contains the GM API function gm_send_token_available().

8.77.2 Function Documentation

8.77.2.1 int gm_send_token_available (gm_port_t * *p*, unsigned *priority*)

gm_send_token_available() tests for the availability of a send token without allocating the send token. It is similar to the function gm_alloc_send_token(port,priority) without the allocation.

Return values:
int

Parameters:
p **(IN)** The GM port on the source/sender GM node from which the communication would be sent.

priority **(IN)** The priority of the message to be sent.

See also:
gm_alloc_send_token

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.78 gm_set_acceptable_sizes.c File Reference

```
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT gm_status_t gm_set_acceptable_sizes (struct gm_port *p, enum gm_priority priority, gm_size_t mask)

8.78.1 Detailed Description

This file contains the GM API function gm_set_acceptable_sizes().

8.78.2 Function Documentation

8.78.2.1 GM_ENTRY_POINT gm_status_t gm_set_acceptable_sizes (struct gm_port * *p*, enum gm_priority *priority*, gm_size_t *mask*)

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

Note: the MASK is a long to support larger than 2GByte packets (those with size larger than 31).

Return values:

GM_SUCCESS Operation completed successfully.

GM_PERMISSION_DENIED Port number hasn't been set.

GM_INTERNAL_ERROR LANai is not running.

GM_INVALID_PARAMETER The priority has an invalid value.

Parameters:

p **(IN)** The GM port in use.

priority **(IN)** The priority of the message.

mask

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.79 gm_set_alarm.c File Reference

```
#include "gm_call_trace.h"
#include "gm_debug.h"
#include "gm_cmp64.h"
#include "gm_internal.h"
#include "gm_send_queue.h"
#include "gm_set_alarm.h"
```

Functions

- GM_ENTRY_POINT void gm_cancel_alarm (gm_alarm_t *my_alarm)
- GM_ENTRY_POINT void gm_initialize_alarm (gm_alarm_t *my_alarm)
- GM_ENTRY_POINT void gm_set_alarm (gm_port_t *port, gm_alarm_t *my_alarm, gm_u64_t usecs, void(*callback)(void *), void *context)

8.79.1 Detailed Description

This file includes source for the user-level alarm API calls gm_initialize_alarm(), gm_set_alarm(), and gm_cancel_alarm().

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.

These source for all the functions is included here, because if the user needs one, the user needs them all.

This code is a bit tricky, since the LANai provides only one alarm per port. This alarm is used to time the first pending alarm, but if any new alarm is set, then the LANai alarm must be flushed to find out what time it is and potentially to allow the alarm to be rescheduled for an earlier time. Therefore, unless no other alarm is set, alarms are set by adding them to an ”unset alarm” queue, flushing any pending alarm, and then adding the alarms to the sorted alarm queue only once the pending LANai alarm has been flushed or triggered.

8.79.2 Function Documentation

8.79.2.1 GM_ENTRY_POINT void gm_cancel_alarm (gm_alarm_t * *my_alarm*)

gm_cancel_alarm() cancels a scheduled alarm, or does nothing if an alarm is not scheduled. Alarms are cancelled by simply removing them from the alarm list so that when the next GM_ALARM_EVENT arrives from the LANai _gm_handle_alarm() will not find the alarm.

Parameters:
my_alarm **(IN)** Alarm to be cancelled.

See also:
gm_initialize_alarm gm_set_alarm gm_flush_alarm

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.79.2.2 GM_ENTRY_POINT void gm_initialize_alarm (gm_alarm_t * *my_alarm*)

gm_initialize_alarm() initializes 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().

Parameters:
my_alarm **(IN)** Alarm to be initialized.

See also:
gm_cancel_alarm gm_set_alarm gm_flush_alarm

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.79.2.3 GM_ENTRY_POINT void gm_set_alarm (gm_port_t * *port*, gm_alarm_t * *my_alarm*, gm_u64_t *usecs*, void(* *callback*)(void *), void * *context*)

gm_set_alarm() sets an alarm, which may already be pending. If it is pending, it is rescheduled. The user supplies MY_ALARM, a pointer to storage for the alarm state, which has been initialized with gm_init_alarm_my_alarm(). When the alarm occurs, CALLBACK(CONTEXT) is called.

gm_set_alarm() schedules **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 the effect of alarm callbacks only here

  break;
```

Parameters:

port **(IN)** A pointer to the GM Port.

my_alarm **(IN)** Alarm to be initialized.

usecs

callback

context

See also:

gm_cancel_alarm gm_initialize_alarm gm_flush_alarm

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.80 gm_set_enable_nack_down.c File Reference

```
#include "gm_internal.h"
#include "gm_debug.h"
```

Functions

- gm_status_t gm_set_enable_nack_down (struct gm_port *port, int flag)

8.80.1 Detailed Description

This file contains the GM API function gm_set_enable_nack_down().

8.80.2 Function Documentation

8.80.2.1 gm_status_t gm_set_enable_nack_down (struct gm_port * *port*, int *flag*)

gm_set_enable_nack_down()

Return values:

GM_SUCCESS Operation completed successfully.

GM_PERMISSION_DENIED

Parameters:

port **(IN)** Handle to the GM port.

flag

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.81 gm_simple_example.h File Reference

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

8.81.1 Detailed Description

Include file for gm_simple_example_send.c and gm_simple_example_recv.c.

8.82 gm_sleep.c File Reference

```
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT int gm_sleep (unsigned int seconds)

8.82.1 Detailed Description

This file contains the GM API function gm_sleep().

8.82.2 Function Documentation

8.82.2.1 GM_ENTRY_POINT int gm_sleep (unsigned int *seconds*)

gm_sleep() emulates the ANSI standard sleep(), sleeping the entire process for **seconds** seconds.

Return values:
SLEEP

Parameters:
seconds **(IN)** The number of seconds for which the process should sleep.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.83 gm_strcmp.c File Reference

```
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT int gm_strcmp (const char *a, const char *b)

8.83.1 Detailed Description

This file contains the GM API function gm_strcmp().

8.83.2 Function Documentation

8.83.2.1 GM_ENTRY_POINT int gm_strcmp (const char * *a*, const char * *b*)

gm_strcmp() reimplements strcmp().

Return values:

int Returns an integer less than, equal to, or greater than zero if a is found, respectively, to be less than, to match, or be greater than b.

Parameters:

a The first string to be compared.

b The second string to be compared.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.84 gm_strdup.c File Reference

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

Functions

- GM_ENTRY_POINT char * gm_strdup (const char *in)

8.84.1 Detailed Description

8.84.2 Function Documentation

8.84.2.1 GM_ENTRY_POINT char* gm_strdup (const char * *in*)

gm_strdup() reimplements the UNIX function strdup().

Return values:

char * Returns a pointer to the duplicated string, or NULL if insufficient memory was available.

Parameters:

in (IN) The string to be duplicated.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.85 gm_strerror.c File Reference

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

Functions

- GM_ENTRY_POINT char * gm_strerror (gm_status_t error)

8.85.1 Detailed Description

This file contains the GM API function gm_strerror().

8.85.2 Function Documentation

8.85.2.1 GM_ENTRY_POINT char* gm_strerror (gm_status_t *error*)

gm_strerror() is an error function for GM. The error is only valid until next call to this function.

Return values:
char

Parameters:
error **(IN)** GM status code.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.86 gm_strlen.c File Reference

```
#include "gm_internal.h"
#include "gm_debug.h"
```

Functions

- GM_ENTRY_POINT gm_size_t gm_strlen (const char *cptr)

8.86.1 Detailed Description

This file contains the GM API function gm_strlen().

8.86.2 Function Documentation

8.86.2.1 GM_ENTRY_POINT gm_size_t gm_strlen (const char * *cptr*)

gm_strlen() calculates the length of a string.

Return values:
gm_size_t The length of the string.

Parameters:
cptr (IN) The string.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.87 gm_strncasecmp.c File Reference

```
#include "gm_internal.h"
#include "gm_debug.h"
```

Functions

- GM_ENTRY_POINT int gm_strncasecmp (const char *a, const char *b, int len)

8.87.1 Detailed Description

This file contains the GM API function gm_strncasecmp().

8.87.2 Function Documentation

8.87.2.1 GM_ENTRY_POINT int gm_strncasecmp (const char * *a*, const char * *b*, int *len*)

gm_strncasecmp() reimplements strncasecmp().

Return values:

int Returns an integer less than, equal to, or greater than zero if the first len bytes of a is found, respectively, to be less than, to match, or be greater than b.

Parameters:

a (IN) The first string to be compared.

b (IN) The second string to be compared.

len (IN) The number of bytes.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.88 gm_strncmp.c File Reference

```
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT int gm_strncmp (const char *a, const char *b, int len)

8.88.1 Detailed Description

This file contains the GM API function gm_strncmp().

8.88.2 Function Documentation

8.88.2.1 GM_ENTRY_POINT int gm_strncmp (const char * *a*, const char * *b*, int *len*)

gm_strncmp() reimplements strncmp().

Return values:

int Returns an integer less than, equal to, or greater than zero if the first len bytes of a is found, respectively, to be less than, to match, or be greater than b.

Parameters:

a (IN) The first string to be compared.

b (IN) The second string to be compared.

len (IN) The length in bytes.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.89 gm_strncpy.c File Reference

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

Functions

- GM_ENTRY_POINT char * gm_strncpy (char *to, const char *from, int len)

8.89.1 Detailed Description

This file contains the GM API function gm_strncpy().

8.89.2 Function Documentation

8.89.2.1 GM_ENTRY_POINT char* gm_strncpy (char * *to*, const char * *from*, int *len*)

gm_strncpy() copies exactly n bytes, truncating src or adding null characters to dst if necessary. The result will not be null-terminated if the length of src is n or more.

Return values:

char * Returns a pointer to a destination string.

Parameters:

to (IN) The destination string to be copied.

from (IN) The source string to be copied.

len (IN) The number of bytes to be copied.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.90 gm_ticks.c File Reference

```
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT gm_u64_t gm_ticks (struct gm_port *port)

8.90.1 Detailed Description

This file contains the GM API function gm_ticks().

8.90.2 Function Documentation

8.90.2.1 GM_ENTRY_POINT gm_u64_t gm_ticks (struct gm_port * *port*)

gm_ticks() returns a 64-bit extended version of the LANai real time clock (RTC). For implementation reasons, the granularity of gm_ticks() is 50 microseconds at the application level.

Return values:
gm_u64_t

Parameters:
port **(IN)** The handle to the GM port.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.91 gm_unique_id.c File Reference

```
#include "gm_internal.h"
```

Functions

- gm_status_t gm_unique_id (gm_port_t *port, char *id)

8.91.1 Detailed Description

This file contains the GM API function gm_unique_id().

8.91.2 Function Documentation

8.91.2.1 gm_status_t gm_unique_id (gm_port_t * *port*, char * *id*)

gm_unique_id() returns the board id number for an interface.

Return values:

GM_SUCCESS Operation completed successfully.

Parameters:

port **(IN)** The GM port.

id **(OUT)** Contains the LANai board id number.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.92 gm_unique_id_to_node_id.c File Reference

```
#include "gm_call_trace.h"
#include "gm_debug.h"
#include "gm_internal.h"
#include "gm_global_id.h"
```

Functions

- gm_status_t gm_unique_id_to_node_id (gm_port_t *port, char *unique, unsigned int *node_id)

8.92.1 Detailed Description

This file contains the GM API function gm_unique_id_to_node_id().

8.92.2 Function Documentation

8.92.2.1 gm_status_t gm_unique_id_to_node_id (gm_port_t * *port*, char * *unique*, unsigned int * *node_id*)

gm_unique_id_to_node_id() takes the MAC address and returns the GM node id for a specific port.

Return values:

GM_SUCCESS Operation completed successfully.

GM_INVALID_PARAMETER

Parameters:

port **(IN)** The GM port.

unique **(IN)** The unique ID to translate.

node_id (OUT) Where to store the node ID corresponding to unique.

Author:

Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.93 gm_unknown.c File Reference

```
#include "gm_call_trace.h"
#include "gm_debug.h"
#include "gm_debug_blocking.h"
#include "gm_enable_put.h"
#include "gm_internal.h"
```

Functions

- GM_ENTRY_POINT void gm_unknown (gm_port_t *p, gm_recv_event_t *e)

8.93.1 Detailed Description

This file contains the GM API function gm_unknown().

8.93.2 Function Documentation

8.93.2.1 GM_ENTRY_POINT void gm_unknown (gm_port_t * *p*, gm_recv_event_t * *e*)

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

Parameters:

p **(IN)** The GM port associated with a specific event.

e **(IN)** The GM receive event.

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

8.94 gm_zone.c File Reference

```
#include "gm_call_trace.h"
#include "gm_internal.h"
#include "gm_malloc_debug.h"
```

Data Structures

- struct gm_zone_area
- struct gm_zone

Typedefs

- typedef gm_zone_area gm_zone_area_t
- typedef gm_zone gm_zone_t

Functions

- GM_ENTRY_POINT struct gm_zone * gm_zone_create_zone (void *base, gm_size_t length)
- GM_ENTRY_POINT void gm_zone_destroy_zone (struct gm_zone *zone)
- GM_ENTRY_POINT void * gm_zone_free (struct gm_zone *zone, void *a)
- GM_ENTRY_POINT void * gm_zone_malloc (struct gm_zone *zone, gm_size_t length)
- GM_ENTRY_POINT void * gm_zone_calloc (struct gm_zone *zone, gm_size_t count, gm_size_t length)
- GM_ENTRY_POINT int gm_zone_addr_in_zone (struct gm_zone *zone, void *p)

8.94.1 Detailed Description

This file contains the GM API functions gm_zone_create_zone(), gm_zone_destroy_zone(), gm_zone_free(), gm_zone_malloc(), and gm_zone_calloc(), gm_zone_addr_in_zone().

This file provides alloc, calloc, and free routines to manage an externally specified ”zone” of memory.

This package use buddy-system memory allocation to allocate (2^n)-byte regions of memory, where ”n” is referred to as the ”size” of the allocated area of memory.

All allocated (or freed) areas are maximally aligned. A zone is a chunk of memory starting and ending on page boundaries. The size and state of each area are encoded in a pair of bit-arrays. One array has a bit set for each position corresponding to a buddy-boundary. The second array has a bit set for each position corresponding to an area that is not free.

In the code, variables named with ”size” are in logarithmic units and those named ”length” are in real units.

DEBUGGING

If debugging is turned on, we ”mark” all valid areas in the zone, using the gm_mark API, and we check each area passed to a function. This should catch any DMA overruns type corruption of the data structures stored in the managed memory.

8.94.2 Define Documentation

8.94.2.1 #define GM_AREA_FOR_PTR(ptr)

Value:

```
((gm_zone_area_t *)                     \
                              ((char *) (ptr) - GM_ZONE_AREA_PTR_OFFSET))
```

8.94.3 Typedef Documentation

8.94.3.1 typedef struct gm_zone_area gm_zone_area_t

Zones are divided into managed buffers called ”areas”, which may either represent free buffers or buffers holding user data.

8.94.3.2 typedef struct gm_zone gm_zone_t

State of a zone, which is a region of memory from which one can allocate memory using the zone allocation functions.

8.94.4 Function Documentation

8.94.4.1 GM_ENTRY_POINT struct gm_zone* gm_zone_create_zone (void * *base*, gm_size_t *length*)

gm_zone_create_zone()

Return values:
gm_zone Handle to the GM zone.

Parameters:
base

length

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.94.4.2 GM_ENTRY_POINT void gm_zone_destroy_zone (struct gm_zone * *zone*)

gm_zone_destroy_zone()

Parameters:
zone **(IN)** Pointer to the GM zone.

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.94.4.3 GM_ENTRY_POINT void* gm_zone_free (struct gm_zone * *zone*, void * *a*)

gm_zone_free()

Parameters:
zone **(IN)** Pointer to the GM zone.

a

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.94.4.4 GM_ENTRY_POINT void* gm_zone_malloc (struct gm_zone * *zone*, gm_size_t *length*)

gm_zone_malloc() mallocs a GM zone.

Parameters:
zone **(IN)** Pointer to the GM zone.

length

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.94.4.5 GM_ENTRY_POINT void* gm_zone_calloc (struct gm_zone * *zone*, gm_size_t *count*, gm_size_t *length*)

gm_zone_calloc() callocs a GM zone.

Parameters:
zone **(IN)** Pointer to the GM zone.

count

length

Author:
Glenn Brown

Version:
GM_API_VERSION (as defined in gm.h)

8.94.4.6 GM_ENTRY_POINT int gm_zone_addr_in_zone (struct gm_zone * *zone*, void * *p*)

gm_zone_addr_in_zone()

Parameters:

zone **(IN)** Pointer to the GM zone.

p

Author:

Glenn Brown

Version:

GM_API_VERSION (as defined in gm.h)

Chapter 9

GM Page Documentation

9.1 XI. 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.

- gm_initialize_alarm()
- gm_cancel_alarm()
- gm_set_alarm()
- gm_flush_alarm()

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;
```

9.2 VII. Page Allocation

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

- gm_page_alloc()
- gm_page_free()
- gm_alloc_pages()
- gm_free_pages()

9.3 XV. GM Constants, Macros, and Enumerated Types

A number of constants, macros, and enumerated types are defined in the gm.h include file.

- Enum: GM_HIGH_PRIORITY The priority of high priority messages (1).
- Enum: GM_LOW_PRIORITY The priority of low priority messages (0).

/∗ Maximum length of GM host name

9.4 I. Copyright Notice

Myricom GM Myrinet(R) software and documentation

Myricom, Inc. Email: <info@myri.com>
325 N. Santa Anita Ave. World Wide Web: ‘http://www.myri.com/’
Arcadia, CA 91024

9.5 II. About This Document

This document describes the GM message passing system and the GM-2.0 API. Several GM-1.0 API functions have been deprecated; however, the 1.0 API will continue to be supported by the GM libraries for the foreseeable future. New programs should use the GM API as described in this document.

9.6 X. 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 inconvenience 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 behavior in most cases.

In the absence 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 the line

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

in your source code. 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:

Network to host conversion routines.

gm_ntoh_u8() unsigned 8-bit

gm_ntoh_u16() unsigned 16-bit

gm_ntoh_u32() unsigned 32-bit

gm_ntoh_u64() unsigned 64-bit

gm_ntoh_s8() signed 8-bit

gm_ntoh_s16() signed 16-bit

gm_ntoh_s32() signed 32-bit

gm_ntoh_s64() signed 64-bit

Host to network conversion routines.

gm_hton_u8() unsigned 8-bit

gm_hton_u16() unsigned 16-bit

gm_hton_u32() unsigned 32-bit

gm_hton_u64() unsigned 64-bit

gm_hton_s8() signed 8-bit

gm_hton_s16() signed 16-bit

gm_hton_s32() signed 32-bit

gm_hton_s64() signed 64-bit

(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.

9.7 XIV. Example Programs

These GM example programs are intended primarily as illustrations and models of GM API usage. As such, they supplement the GM API documentation, and it is probably more useful to read their source than to execute them.

These programs illustrate the use of simple message sending (gm_send_to_peer_with_callback()) and receiving (gm_receive()), and also the use of gm_directed_send_with_callback().

- gm_simple_example.h
- gm_simple_example_send.c
- gm_simple_example_recv.c

9.8 XII. 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 re-enable 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`.

A complete list of send completion status codes can be found in gm.h and section VIII. Sending Messages.

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 re-enable 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 re-enable 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.

Here is an example program demonstrating the use of gm_drop_sends(). In this example, there are no messages queued after the message that has just been discarded, and gm_drop_sends() and gm_resume_sending() are equivalent. They just re-enable the target subport for further gm_send_with_callback() calls. If the send callback returns you an error, that means the corresponding message has been definitely discarded, both gm_resume_sending() and gm_drop_sends() only impact messages that have been queued **after** the message that has just been discarded.

```
#include <stdio.h>
#include <assert.h>

#include "gm.h"

unsigned int received,sent;
unsigned int my_gm_node_id;

static void test_send_callback (struct gm_port *port, void *context,
                                gm_status_t status);

static void
drop_send_callback (struct gm_port *port,
                    void *context,
                    gm_status_t status)
{
   fprintf(stderr, "Got gm_drop_send notification, start resending\n");
   gm_send_with_callback(port, context, 20, 1, GM_LOW_PRIORITY,
                            my_gm_node_id, 7, test_send_callback, context);
}


static void
test_send_callback (struct gm_port *port,
                    void *context,
                    gm_status_t status)
{
  switch (status)
    {
    case GM_SUCCESS:
      fprintf(stderr, "Send successfully delivered\n");
      sent += 1;
      break;

    case GM_SEND_TIMED_OUT:
      fprintf(stderr, "Send timeout, provide buffers and initiate resend...\n");
      gm_provide_receive_buffer(port, context, 20, GM_LOW_PRIORITY);
      gm_provide_receive_buffer(port, context, 20, GM_LOW_PRIORITY);
      gm_drop_sends (port, GM_LOW_PRIORITY, my_gm_node_id, 7,
                          drop_send_callback, context);
      break;
```

```
     case GM_SEND_DROPPED:
      fprintf(stderr, "Got DROPPED_SEND notification, resend\n");
      gm_send_with_callback(port, context, 20, 1, GM_LOW_PRIORITY,
                            my_gm_node_id, 7, test_send_callback, context);
       break;

    default:
      fprintf(stderr, "Something bad happen\n");
      assert (0);
    }
}


int main (void)
{
  struct gm_port *port;
  gm_recv_event_t *event;
  char *token;

  assert (gm_open (&port, 0, 7, "Test Resume", GM_API_VERSION) == GM_SUCCESS);
  token = gm_dma_malloc (port, sizeof (char));
  assert (token != NULL);

  received = 0;
  gm_get_node_id(port, &my_gm_node_id);
  gm_send_with_callback (port, token, 20, sizeof (char),
                         GM_LOW_PRIORITY, my_gm_node_id, 7,
                         test_send_callback, token);
  gm_send_with_callback (port, token, 20, sizeof (char),
                         GM_LOW_PRIORITY, my_gm_node_id, 7,
                         test_send_callback, token);

  while (received < 2 || sent < 2)
    {
      event = gm_receive (port);
      switch (gm_ntoh_u8 (event->recv.type))
        {
        case GM_NO_RECV_EVENT:
          break;
         case GM_RECV_EVENT:
          fprintf(stderr,"received message\n");
          received += 1;
          break;
        default:
          gm_unknown (port, event);
        }
    }

  gm_dma_free (port, token);
  gm_close (port);
}
```

9.9 V. 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_2_0' as defined in `gm.h`.

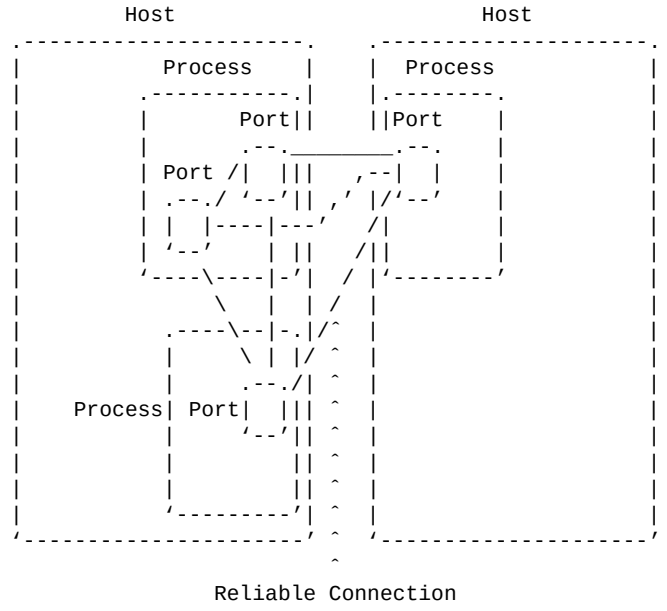
Note that while the GM API uses '`struct` gm_port *' pointers throughout, these pointers are opaque to the client. The client should not attempt to dereference these pointers.

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.

9.10 IV. 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.)



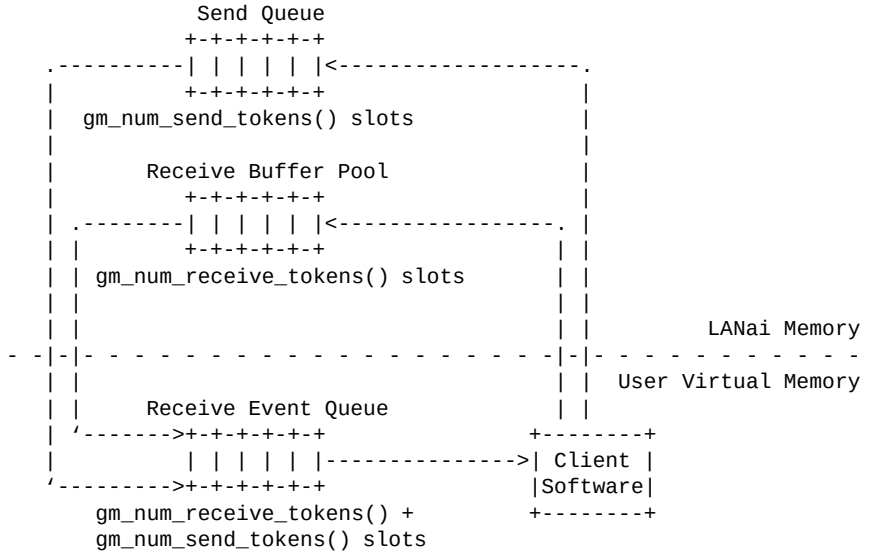
9.10.1 1. GM Endpoints (Ports)

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.

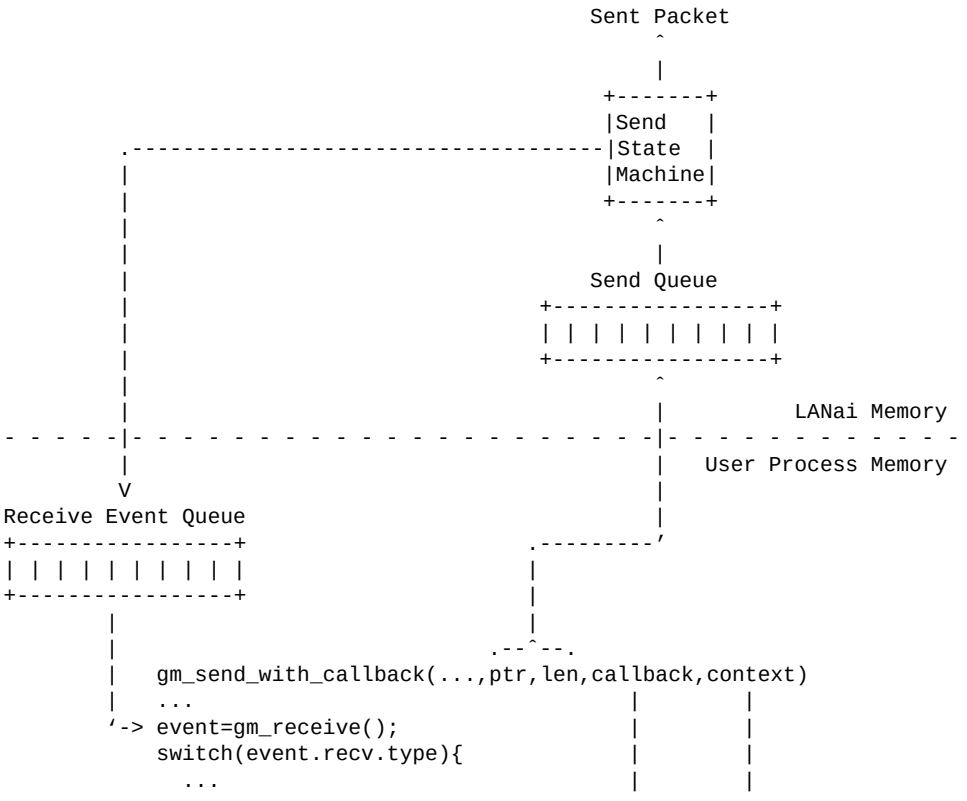
Message order is preserved only for messages of the same priority, from the same sending port, and directed to the same receiving port. Messages with differing priority never block each other. Consequently, low priority messages may pass high priority messages, unlike in some other communication systems. 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.

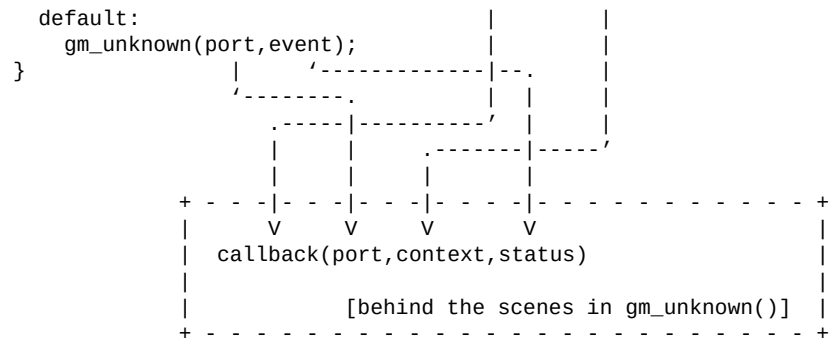
Both sends and receives in GM are regulated by implicit tokens, representing space allocated to the client in various internal GM queues, as depicted in the following figure. At initialization, the client implicitly possesses gm_num_send_tokens() send tokens, and gm_num_receive_tokens() receive tokens. 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.



9.10.2 2. User Token Flow (Sending)

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 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.





The following functions require send tokens:

- gm_datagram_send()
- gm_directed_send_with_callback()
- gm_drop_sends()
- gm_resume_sending()
- gm_send_to_peer_with_callback()
- gm_send_with_callback()

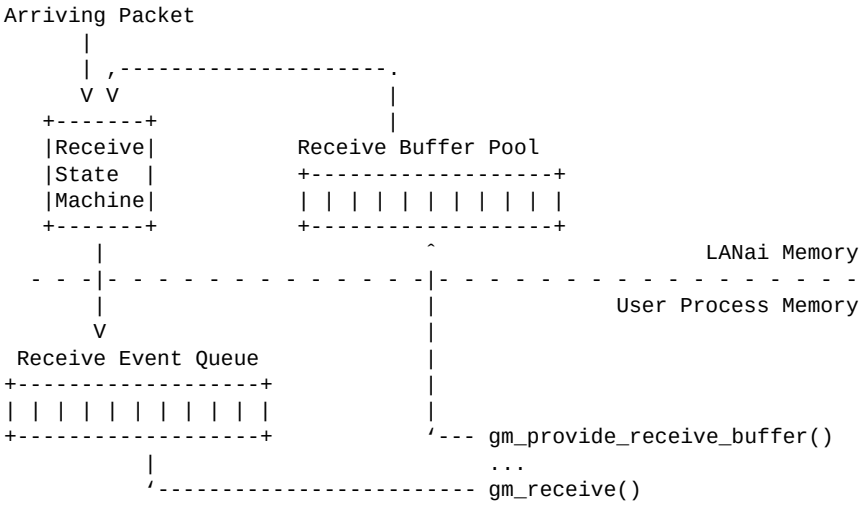
The send token is implicitly returned to the client when the function's callback is called or, for the GM-1.0 functions gm_send() and gm_send_to_peer(), a send token is implicitly passed to the client with each pointer returned in a `GM_SENT_EVENT`. (The legacy `GM_SENT_EVENTS` are generated if and only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

9.10.3 3. User Token Flow (Receiving)

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(), gm_blocking_receive(), or gm_blocking_receive_no_spin() function. The gm_receive(), gm_blocking_receive(), or gm_blocking_receive_no_spin() 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.



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.

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()

9.11 III. 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.

9.11.1 1. 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 MPI or VI.

- **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

$$\log_2 (\text{LENGTH} + 8)$$

  where LENGTH is the length of the message.

The **size** of a receive buffer is any positive integer **less** than or equal to

$$\log_2 (\text{LENGTH} + 8)$$

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.

9.11.2 2. Notation

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

9.12 IX. 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_calloc() or gm_dma_malloc() 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 one of these three functions, 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 X. Endian Conversion.

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` or `GM_HIGH_RECV_EVENT` 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_NO_RECV_EVENT** No event is in the event queue.
- **GM_ALARM_EVENT** GM_ALARM_EVENT 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` to gm_unknown(), as in the 'test/gm_allsize.c' example program. See the documentation for gm_set_alarm() for more information.
- **GM_SENT_EVENT** This type indicates that one or more sends completed. ∗Developers using the GM-1.5 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.

- **_GM_SLEEP_EVENT**
- **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_RAW_RECV_EVENT**

- **GM_BAD_SEND_DETECTED_EVENT**

- **GM_SEND_TOKEN_VIOLATION_EVENT**

- **GM_RECV_TOKEN_VIOLATION_EVENT**

- **GM_BAD_RECV_TOKEN_EVENT**

- **GM_ALARM_VIOLATION_EVENT**

- **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.

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.message),
                      gm_ntohp (e->recv.buffer),
                      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.

9.13 VIII. 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_with_callback() and the send completion, because doing so will cause undefined data to be delivered to the receiver.

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.

The send completion status codes (listed in gm.h) are as follows:

- **GM_SUCCESS** The send succeeded. This status code does not indicate an error.
- **GM_FAILURE** Operation failed.
- **GM_INPUT_BUFFER_TOO_SMALL** Input buffer is too small.
- **GM_OUTPUT_BUFFER_TOO_SMALL** Output buffer is too small.
- **GM_TRY_AGAIN** Try again.
- **GM_BUSY** GM Port is Busy.
- **GM_MEMORY_FAULT** Memory Fault.
- **GM_INTERRUPTED** Interrupted.
- **GM_INVALID_PARAMETER** Invalid input parameter.

- **GM_OUT_OF_MEMORY** Out of Memory.
- **GM_INVALID_COMMAND** Invalid Command.
- **GM_PERMISSION_DENIED** Permission Denied.
- **GM_INTERNAL_ERROR** Internal Error.
- **GM_UNATTACHED** Unattached.
- **GM_UNSUPPORTED_DEVICE** Unsupported Device.
- **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.
- **GM_NODE_ID_NOT_YET_SET** Node ID is not yet set.
- **GM_STILL_SHUTTING_DOWN** GM Port is still shutting down.
- **GM_CLONE_BUSY** GM Clone Busy.

- **GM_NO_SUCH_DEVICE** No such device.
- **GM_ABORTED** Aborted.
- **GM_INCOMPATIBLE_LIB_AND_DRIVER** Incompatible GM library and driver.
- **GM_UNTRANSLATED_SYSTEM_ERROR** Untranslated System Error.
- **GM_ACCESS_DENIED** Access Denied.
- **GM_NO_DRIVER_SUPPORT** No Driver Support.
- **GM_PTE_REF_CNT_OVERFLOW** PTE Ref Cnt Overflow.
- **GM_NOT_SUPPORTED_IN_KERNEL** Not supported in the kernel.
- **GM_NOT_SUPPORTED_ON_ARCH** Not supported for this architecture.
- **GM_NO_MATCH** No match.
- **GM_USER_ERROR** User error.
- **GM_TIMED_OUT** Timed out.
- **GM_DATA_CORRUPTED** Data has been corrupted.
- **GM_HARDWARE_FAULT** Hardware fault.
- **GM_SEND_ORPHANED** Send orphaned.
- **GM_MINOR_OVERFLOW** Minor overflow.
- **GM_PAGE_TABLE_FULL** Page Table is Full.
- **GM_UC_ERROR** UC Error.
- **GM_INVALID_PORT_NUMBER** Invalid Port Number.

9.14 VI. Memory Setup

GM will only send messages from memory allocated with a gm_dma_calloc() or gm_dma_malloc() 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, and a BOGUS send or receive will appear in the kernel log.

Note that some operating systems (e.g.: Solaris) do not support gm_register_memory() due to operating system limitations, so the gm_dma_calloc() or gm_dma_malloc() 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_with_callback() (gm_put()) 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.

9.15 XVI. Function Summary

We have subdivided this GM function summary into a listing of ”Basic GM API” functions and ”Advanced GM API” functions.

The following ”basic” functions should suffice for most GM API applications.

Basic GM API

Initialization

- gm_init()
- gm_finalize()
- gm_open()
- gm_close()
- gm_exit()
- gm_abort()

Currently, gm_open() implicitly calls gm_init() for the caller and gm_close() implicitly calls gm_finalize(), but developers should not rely on this.

Memory Setup

- gm_dma_calloc()
- gm_dma_malloc()
- gm_dma_free()

Sending Messages

- gm_send_to_peer_with_callback()
- gm_send_with_callback()

Receiving Messages

- gm_provide_receive_buffer_with_tag()
- gm_receive()
- gm_unknown()

Endian Conversion

- gm_ntoh_u8()
- gm_ntoh_u16()
- gm_ntoh_u32()
- gm_ntoh_u64()
- gm_ntoh_s8()
- gm_ntoh_s16()
- gm_ntoh_s32()
- gm_ntoh_s64()

Error Handling

- gm_perror()
- gm_eprintf()
- gm_printf()

Utility Functions

- gm_max_length_for_size()
- gm_min_message_size()
- gm_min_size_for_length()
- gm_set_acceptable_sizes()

Advanced GM API

The following ”advanced” functions supplement the Basic GM API functions; they may be useful for more complex applications.

Memory Setup

- gm_allow_remote_memory_access()
- gm_register_memory()
- gm_deregister_memory()
- gm_alloc_pages()
- gm_free_pages()
- gm_page_alloc()
- gm_page_free()

Sending Messages

- gm_alloc_send_token()
- gm_datagram_send()
- gm_datagram_send_4()
- gm_directcopy_get()
- gm_directed_send_with_callback()
- gm_put()
- gm_free_send_token()
- gm_free_send_tokens()
- gm_num_send_tokens()
- gm_send_token_available()

Receiving Messages

- gm_blocking_receive()
- gm_blocking_receive_no_spin()
- gm_get()
- gm_memorize_message()
- gm_next_event_peek()
- gm_num_receive_tokens()
- gm_receive_pending()

Endian Conversion

- gm_hton_u8()
- gm_hton_u16()
- gm_hton_u32()
- gm_hton_u64()
- gm_hton_s8()
- gm_hton_s16()
- gm_hton_s32()
- gm_hton_s64()
- gm_htopci_u32()

Alarms

- gm_cancel_alarm()
- gm_flush_alarm()

- gm_initialize_alarm()
- gm_set_alarm()

High Availability Extensions

- gm_drop_sends()
- gm_resume_sending()

Utility Functions

- gm_get_host_name()
- gm_get_node_id()
- gm_get_node_type()
- gm_get_unique_board_id()
- gm_get_mapper_unique_id()
- gm_getpid()
- _gm_get_route()
- gm_get_port_id()
- gm_host_name_to_node_id_ex()
- gm_max_node_id()
- gm_max_node_id_in_use()
- gm_mtu()
- gm_node_id_to_host_name_ex()
- gm_node_id_to_unique_id()
- gm_num_ports()

- gm_set_enable_nack_down()
- gm_unique_id()
- gm_unique_id_to_node_id()

Miscellaneous Routines

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.

- gm_bcopy()
- gm_bzero()
- gm_calloc()
- gm_free()
- gm_hex_dump()
- gm_isprint()
- gm_log2_roundup()
- gm_malloc()
- gm_memcmp()
- gm_memset()
- gm_on_exit()
- gm_rand()
- gm_rand_mod()
- gm_srand()
- gm_sleep()

- gm_strdup()
- gm_strerror()
- gm_strlen()
- gm_strncpy()
- gm_strcmp()
- gm_strncmp()
- gm_strncasecmp()
- gm_ticks()

Utility Modules

These GM internal modules may be useful to GM developers.

- CRC Functions
  - gm_crc()
  - gm_crc_str()
- Hash Table Functions
  - gm_create_hash()
  - gm_destroy_hash()
  - gm_hash_rekey()
  - gm_hash_remove()
  - gm_hash_find()
  - gm_hash_insert()
  - gm_hash_compare_strings()
  - gm_hash_hash_string()
  - gm_hash_compare_longs()
  - gm_hash_hash_long()
  - gm_hash_compare_ints()
  - gm_hash_hash_int()
  - gm_hash_compare_ptrs()
  - gm_hash_hash_ptr()
- Lookaside Functions
  - gm_create_lookaside()
  - gm_destroy_lookaside()

  - gm_lookaside_alloc()
  - gm_lookaside_zalloc()
  - gm_lookaside_free()
  - gm_create_mark_set()
  - gm_destroy_mark_set()

- Mark Functions
  - gm_mark()
  - gm_mark_is_valid()
  - gm_unmark()
  - gm_mark_set_unmark_all()

- Zone Functions
  - gm_zone_create_zone()
  - gm_zone_destroy_zone()
  - gm_zone_free()
  - gm_zone_malloc()
  - gm_zone_calloc()
  - gm_zone_addr_in_zone()

- Mutex Functions
  - gm_create_mutex()
  - gm_destroy_mutex()
  - gm_mutex_enter()
  - gm_mutex_exit()

- Buffer Debugging
  - gm_dump_buffers()
  - gm_register_buffer()
  - gm_unregister_buffer()

9.16 XIII. Utility Modules

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

9.16.1 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.

- gm_crc()
- gm_crc_str()

9.16.2 2. Hash Table

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 `int`s 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.

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

- gm_create_hash()
- gm_destroy_hash()
- gm_hash_rekey()
- gm_hash_remove()
- gm_hash_find()
- gm_hash_insert()

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.

9.16.3 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:

- gm_create_lookaside()
- gm_destroy_lookaside()
- gm_lookaside_alloc()
- gm_lookaside_zalloc()
- gm_lookaside_free()

9.16.4 4. Marks

The GM "mark" API was introduced in 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.

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.

The following APIs are provided:

- gm_create_mark_set()
- gm_destroy_mark_set()
- gm_mark()
- gm_mark_is_valid()
- gm_unmark()
- gm_mark_set_unmark_all()

9.16.5 5. Zones

These GM API routines manage an externally specified *zone* of memory. A zone is a chunk of memory starting and ending on page boundaries. The size and state of each area are encoded in a pair of bit-arrays. All allocated (or freed) areas are maximally aligned.

The following APIs are provided:

- gm_zone_create_zone()
- gm_zone_destroy_zone()
- gm_zone_free()
- gm_zone_malloc()
- gm_zone_calloc()
- gm_zone_addr_in_zone()

9.16.6 6. Mutexes

GM mutex routines have the following API:

- gm_create_mutex()
- gm_destroy_mutex()
- gm_mutex_enter()
- gm_mutex_exit()

9.16.7 7. Buffer Debugging

There are three API functions provided for buffer debugging:

- gm_dump_buffers()

- gm_register_buffer()
- gm_unregister_buffer()

Index