Event Manager User Guide

007-4661-001

CONTRIBUTORS
Written by Darrin Goss
Edited by Cindi Leiser
Production by Glen Traefald
Engineering contributions by Andrei Vilkotski, Elena Gorvitovskaia, and Jonathan Lim.

Cover design by Sarah Bolles, Sarah Bolles Design, and Dany Galgani, SGI Technical Publications.

Record of Revision

Version	Description
001	August 2003 Original publication

Contents

Figures

Tables

Examples

About This Document

This document describes the Event Manager application. It includes the following topics:

- An overview of the Event Manager application
- A description of the Event Manager application programming interface (API)
- Information about creating producer, subscriber, and consumer applications
- Information about the `eventmond` command-line options

Obtaining Publications

You can obtain SGI documentation in the following ways:

- Visit the online SGI Technical Publications Library at http://docs.sgi.com. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.
- If it is installed on your SGI system, you can use InfoSearch, an online tool that provides a more limited set of online books, release notes, and man pages. With an IRIX system, select **Help** from the Toolchest, and then select **InfoSearch**. Or you can type `infosearch` on a command line.
- You can also view release notes by typing either `grelnotes` or `relnotes` on a command line.
- You can also view man pages by typing `man` <*title*> on a command line.

Conventions

The following conventions are used throughout this publication:

Convention	**Meaning**
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
[]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, contact SGI. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library webpage:

  http://docs.sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Pkwy, M/S 535
  Mountain View, California 94043-1351

- Send a fax to the attention of “Technical Publications” at +1 650 932 0801.

SGI values your comments and will respond to them promptly.

Overview

The Event Manager collects event information from other applications. It runs independently of all other applications and enables local or remote applications to receive event data from it on a subscription basis.

The Event Manager uses a producer/consumer architecture (refer to Figure 1-1).

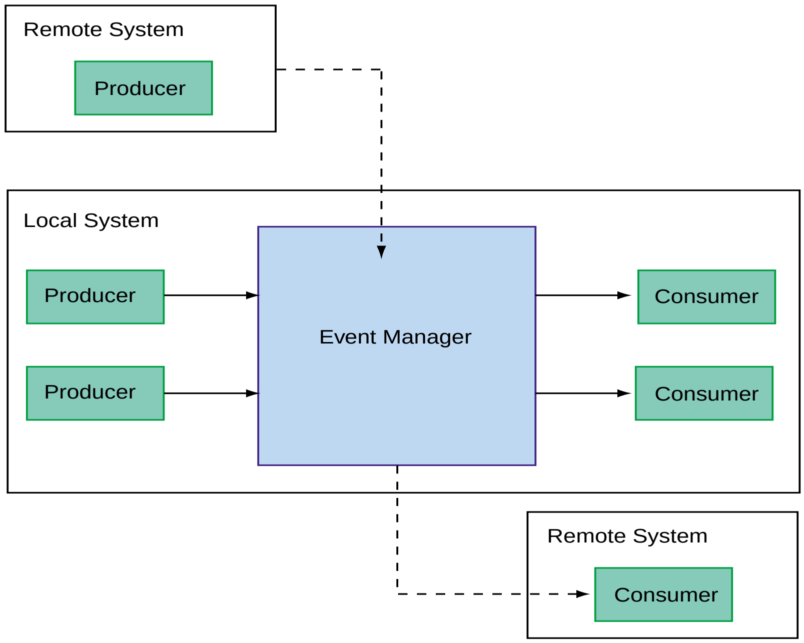


Figure 1-1 Event Manager Architecture

Applications that create events and send them to the Event Manager are called producer applications (or *producers*). Applications that subscribe to receive event information from the Event Manager are called consumer applications (or *consumers*). Producers and consumers can reside on the same system as the Event Manager or on remote systems.

When the Event Manager registers an event from a producer application, it forwards the event information to all consumer applications that are subscribed to receive information about the event. Those applications must include functionality to process the event because the Event Manager simply forwards the event information that it receives; the Event Manager does not process the event information.

The Event Manager runs as a daemon (`eventmond`) that starts at system startup and waits to receive events from producer applications and send event information to consumer applications. You can also manually run `eventmond` to send commands to the daemon through secured UNIX domain sockets.

Event management uses the following components (refer to Figure 1-2):

- Event manager
- Event producer
- Event subscriber
- Event consumer
- Event manager application programming interface (API)

Note: A single application can combine any or all of the event subscriber, producer, and consumer functions.

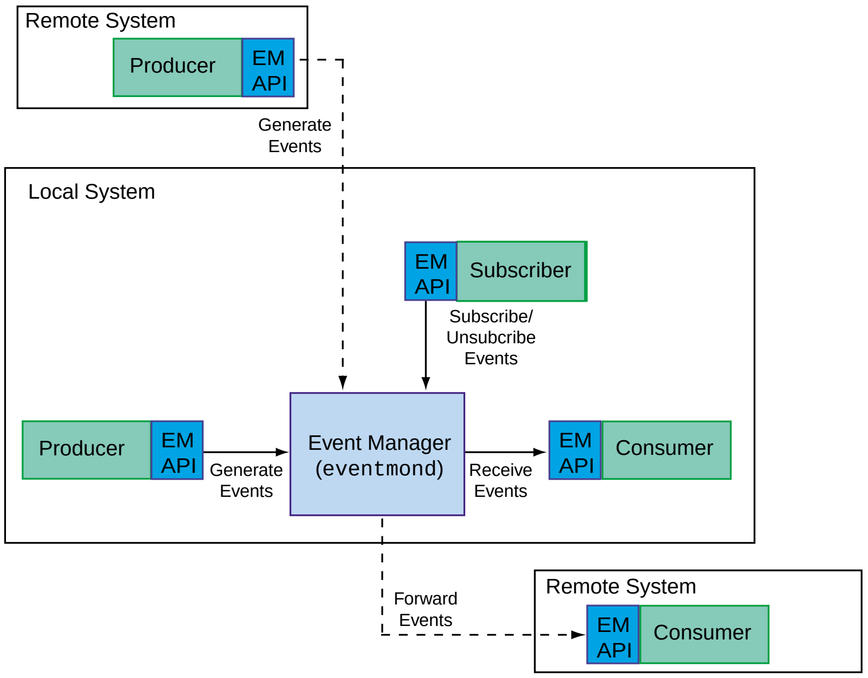


Figure 1-2 Event Manager Components

Event Manager

The Event Manager is a multi-threaded UNIX process that normally runs at system startup as a daemon (`eventmond`) and monitors TCP/IP port 5553 for events from producers and subscription/unsubscription requests from subscribers. When the Event Manager detects an event, it forwards the event information to all local and remote consumers that are subscribed to the event. The Event Manager discards any events that are not subscribed to a consumer.

You can use the `eventmond` command to configure the daemon and control tasks that it runs. (Refer to Chapter 4, “eventmond Command-line Options,” for more information)

Note: This new version of `eventmond` replaces the version of `eventmond` that shipped with earlier versions of IRIX. It is named `eventmond` to provide compatibility with older versions of IRIX.

Event Producer

An event producer is an application that creates events and sends them to the Event Manager via the Event Manager API. There are two types of event producer applications:

- A separate process that runs on the local system or on a remote system.
- A shared library (also called a dynamic shared object [DSO]) that the Event Manager loads and executes.

Example event producers include the `syslog` DSO and the `availmon` and `configmon` standalone applications.

Event Subscriber

An event subscriber is a special subscription management application. It subscribes a consumer to an event so the consumer can receive event information from the Event Manager. It unsubscribes the consumer from the event when the consumer no longer requires information about the event.

An example event subscriber is `espconfig`, which handles event subscription for Embedded Support Partner (ESP) version 3.0.

Note: Event subscriber functions can also be contained in producer or consumer applications.

Event Consumer

An event consumer is an application that subscribes to receive events and then processes the event data that it receives. There are four types of event consumer applications:

- A *shared library consumer* is a compiled function that is linked in a shared library that the Event Manager can dynamically load into its address space and then execute. The Event Manager passes the event as a parameter to the function. You must specify the name of the shared library and the function that you want to call in the shared library when you subscribe the consumer to the event. If the shared library is not in a standard library directory (for example, `/usr/lib`), you must specify the full path to the shared library when you subscribe the consumer.
- An *executable consumer* can be any type of executable file that the Event Manager can execute with a `fork()/exec()` sequence. If the executable file is not accessible through the `PATH` environment variable, you must specify the full path to the file when you subscribe the consumer.
- A *shared memory executable consumer* is an application that can access the event information directly from memory that is shared with the Event Manager. The Event Manager sends the consumer a command-line parameter that specifies a key which indicates where the consumer can access the event information.
- A *forwarding consumer* is a feature of the Event Manager that simply forwards the event to another application. It does not process the event.

Consumers must subscribe to events with the Event Manager to receive information about the events that occur. Consumers should unsubscribe from events when they no longer need to receive the event information. Event subscription and unsubscription is normally performed by a subscriber application, but it can be performed by a producer, consumer, or subscriber application.

An example event consumer application is the Embedded Support Partner consumer DSO that processes all events for ESP by logging the events and performing actions assigned to them.

Event Manager API

The Event Manager application programming interface (API) contains a set of functions that enable other applications to communicate with the Event Manager. Event producers send information to the Event Manager via the Event Manager API. Event handlers (consumers) receive event information via the Event Manager API.

Subscriber applications use the Event Manager API to manage the event subscription/unsubscription process for consumers. The API also enables applications to access information within an event.

The API library is dynamically linked to the applications.

Refer to Chapter 2, "Event Manager API," for descriptions of the Event Manager API functions.

Event Manager API

The Event Manager application programming interface (API) provides functions that enable applications to communicate with the Event Manager daemon (`eventmond`). These functions enable the applications to:

- Subscribe events
- Unsubscribe events
- Log events
- Query events

The Event Manager API uses a TCP/IP socket to communicate with the Event Manager daemon. The `emgrapi.h` file contains the function declarations, and the `libemgrapi.so` file contains the actual functions.

This chapter describes the functions that the Event Manager API contains. Chapter 3, "Creating Producer, Subscriber, and Consumer Applications," describes how to use the functions to create producer, subscriber, and consumer applications.

API Data Structures

The Event Manager API functions use the following special data structures:

- Event structure
- GeneralBlock structure

Event Structure

The API functions use a structure called *event* (`EmgrEvent_t`) to pass event information. The event structure includes a fixed portion (the event header) and a variable portion (the event data). The event header contains information about the event (application that created it, host where it originated, time that it occurred, and so on), and the event data contains the actual event.

Note: An event structure contains all public data for the event. All private data that the API control layer requires is stored in a private event control structure that contains the event structure as its first element. Do not directly allocate memory for an event structure; use the `emgrAllocEvent()` API function to allocate event resources.

Figure 2-1 shows the layout of the event structure.

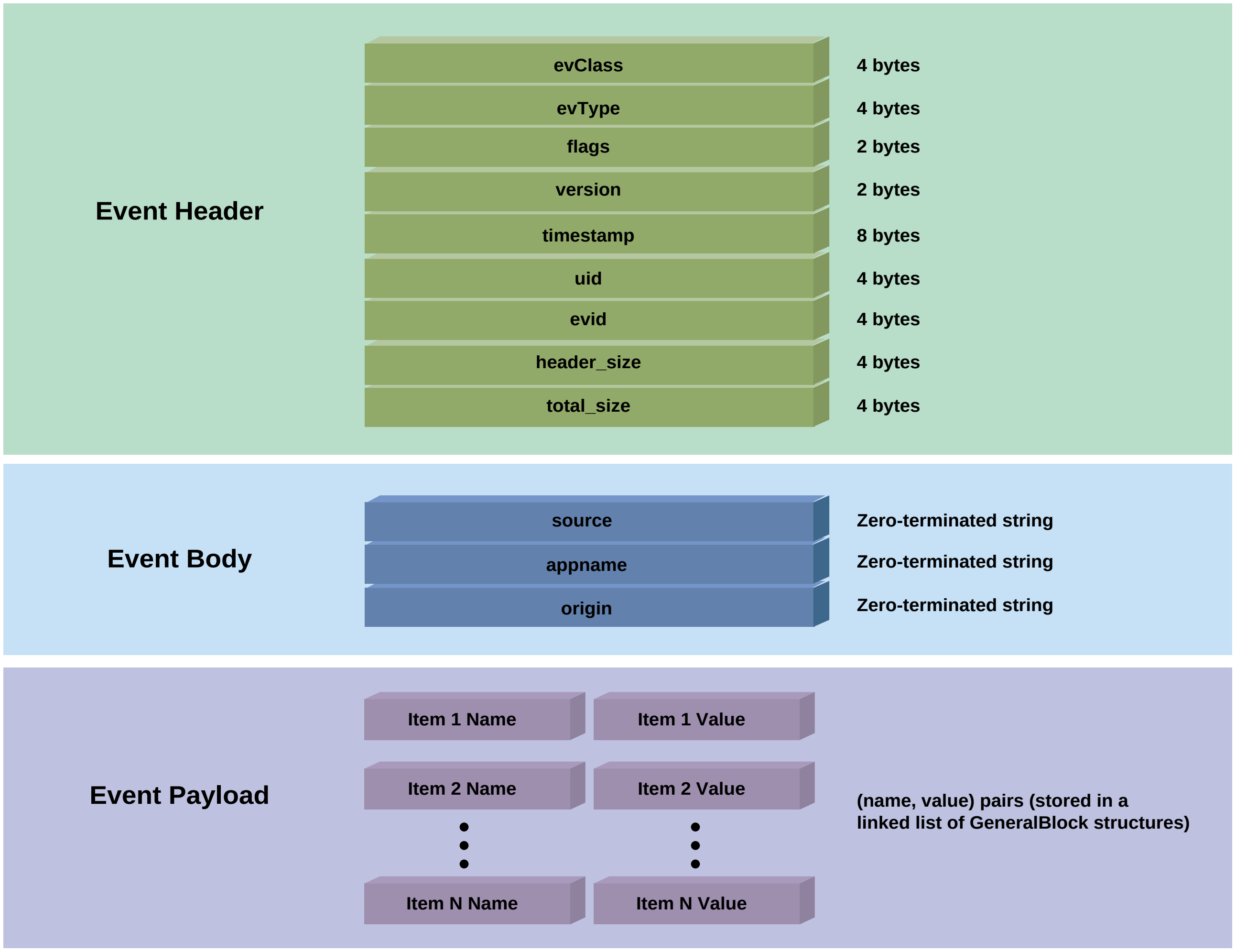


Figure 2-1 Event Structure Layout

Table 2-1 describes the fields that the event header contains.

Table 2-1 Event Header Fields

Field	Description	Size
`evClass`	Event class ID number	4 bytes
`evType`	Event type ID number that is unique to each application	4 bytes
`flags`	Internal flags that indicate how to handle the message	2 bytes
`version`	Event version number that is specific to each application	2 bytes
`timestamp`	Time that the event occurred	8 bytes
`uid`	User ID number of the process that generated the event	4 bytes
`evid`	Event ID number	4 bytes
`header_size`	Size of the event header plus event body	4 bytes
`total_size`	Size of the entire event	4 bytes

Table 2-2 describes the fields that the event body contains.

Table 2-2 Event Body Fields

Field	Description	Size
`source`	Hostname (including domain name) of the system that generated the event	Variable (included in a zero- terminated string)
`appname`	Application that owns the event (for example, Kernel or UNIX)	Variable (included in a zero terminated string)
`origin`	Application that generated the event (for example, SYSLOG)	Variable (included in a zero terminated string)

The event payload portion of the event structure contains a linked list of “items” that are formatted as (name, value) pairs. The *name* portion is a zero-terminated string that acts like an additional field in the event structure. The *value* portion is a typed value that the Event Manager can use to filter expressions or a consumer application can use as event data.

You must enter information in the following fields before you can send event information via the Event Manager API:

- The `source` field must be set to the hostname of the system where the event originates.
- The `appname` field must be set to the application that is sending the event (for example, ESP, CXFS, etc.).

The following definitions in the `emgrapi.h` file create the event structure:

- Definition of the event header:

```
typedef struct EmgrEventHeader {

    int32_t evClass;     /* Event Class number (application specific) */
    int32_t evType;     /* Event Type number (application specific) */
    int16_t flags;       /* Event flags */
    int16_t version;     /* Event Version (application specific) */
    int64_t timestamp;   /* Event Time (GMT) */
    int32_t uid;         /* User ID of a process which sends an event */
    uint32_t evid;        /* Event id  <16 bit sq number><16 bit random> */
    int32_t header_size; /* size of fixed and variable parts of header */
    int32_t   total_size;  /* size of entire event including header  */

} EmgrEventHeader_t;
```

- Definition of the event body:

```
typedef struct EmgrEvent {

/* Fixed portion of public event data
 */
EmgrEventHeader_t header;

/* Variable portion of public event data
 */
char *source;           /* fully qualified local hostname */
char *appname;          /* Name of application that owns this event */
char *origin;           /* Name of application that logged this event */

/* Private API data is appended here
 *
 * Note: Never directly allocate memory for an EmgrEvent struct.
 * Always use the emgrAllocEvent() function to allocate an event struct.
 */
} EmgrEvent_t;
```

GeneralBlock Structure

The GeneralBlock structure is an abstract structure that defines data within an event structure. The following definition in the `emgrGb.h` file creates the GeneralBlock structure:

```
typedef struct GeneralBlock {
      int32_t type;
      int32_t length;

      void *pValue;

      char    tag[1];
} GeneralBlock_t;
```

The GeneralBlock structure provides a way to represent (name, value) pairs for various types of data. The structure currently supports strings, binary data, and file data; however, it is an open structure that can support other types if needed. The event data is a linked list of GeneralBlock structures.

When a data item in an event structure is a file, there is a `GbFileValue` structure that defines it. The `GbFileValue` structure contains the size of the file data and related information, the modification time of the file, the path to the file, and the raw file data. The following definition in the `emgrGb.h` file creates the `GbFileValue` structure:

```
typedef struct GbFileValue {
      int32_t  size; /* size of the static attributes (size and
                        mod time) + the length of the path and file */
      char     modTime[24];
      char     *path;
      int8_t   *pContent;
} GbFileValue_t;
```

API Functions

The API functions have the following categories:

- Event manipulation functions:
  - Creation/definition functions
  - Access functions
- Subscription manipulation functions
- Transmission/execution functions
- Configuration functions

Table 2-3 lists the functions that belong to each category and the type of application (producer, subscriber, and/or consumer) that uses each function. Descriptions of the individual functions appear in alphabetical order after the table.

Table 2-3 API Function Categories

Category	**Function**	**Used By**[a]
Event manipulation (creation/definition)	`emgrAddGbToEvent()`	P
	`emgrAddIntItemToEvent()`	P/S
	`emgrAddItemToEvent()`	P/S
	`emgrAddTaggedDataToEvent()`	P
	`emgrAddTaggedFileToEvent()`	P
	`emgrAddDataToEvent()`	P
	`emgrAddFileToEvent()`	P
	`emgrAllocEvent()`	P
	`emgrCloneEvent()`	C
	`emgrCloneGb()`	C
	`emgrFreeEvent()`	P
	`emgrGetEventItem()`	C

Table 2-3 API Function Categories **(continued)**

Category	Function	Used By[a]
Event manipulation (creation/definition) (cont.)	`emgrGetFirstEventGb()`	C
	`emgrGetFirstEventItem()`	C
	`emgrGetNextEventGb()`	C
	`emgrGetNextEventItem()`	C
	`emgrNewQuery()`	P/S
	`emgrSetToForward()`	P
	`emgrShmCliInitEvent()`	C
	`emgrShmInitEvent()`	C
Event manipulation (access)	`emgrBuildQSearch()`	C
	`emgrPrintEvent()`	P/S/C
	`emgrSearchGb()`	C
	`emgrCheckEvent()`	P/S/C
	`emgrAddSubscribe()`	S
	`emgrAddUnsubscribe()`	S
	`emgrNewSubscribe()`	S
	`emgrNewUnsubscribe()`	S
	`emgrSubscribeSpecCntFreq()`	S
	`emgrSubscribeSpecDsoConsumer()`	S
	`emgrSubscribeSpecExecConsumer()`	S
	`emgrSubscribeSpecExecShMemConsumer()`	S
	`emgrSubscribeSpecFacility()`	S
	`emgrSubscribeSpecForwardConsumer()`	S
	`emgrSubscribeSpecPriority()`	S

Table 2-3 API Function Categories **(continued)**

Category	Function	Used By[a]
Subscription manipulation (cont.)	`emgrSubscribeSpecRegexpMap()`	S
	`emgrSubscribeSpecTimeFreq()`	S
Transmission/execution	`emgrForwardEvent()`	P
	`emgrRunQuery()`	P/S
	`emgrRunSubscribe()`	S
	`emgrRunUnSubscribe()`	S
	`emgrSendEvent()`	P/S
Configuration	`emgrIsDaemonInstalled()`	P/S
	`emgrIsDaemonStarted()`	P/S
	`getConfigValue()`	P/S

a. The "Used By" column indicates the type of application that uses each function (P = producer, S = subscriber, and C = consumer).

The following sections describe the API functions that are available.

emgrAddDataToEvent()

```
int emgrAddDataToEvent(EmgrEvent_t *pEvent,
                       const void *databuf,
                       size_t size);
```

The `emgrAddDataToEvent()` function adds binary data to an event.

Parameters:

**pEvent*	pointer to an event structure
**databuf*	pointer to a data buffer (The pointer should be valid while you use the event; producers should free the data buffer memory.)
size	size of the data buffer (in bytes)

Return value:

- Success: 0
- Failure:

-1	A processing error occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrAddFileToEvent()

```
int emgrAddFileToEvent(EmgrEvent_t *pEvent,
                       const char *path);
```

The `emgrAddFileToEvent()` function adds the contents of a file to an event.

Parameters:

**pEvent*	pointer to an event structure
**path*	pointer to a character string that contains the full pathname of the file

Return value:

- Success: 0
- Failure:

-1	A processing error occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.
17	The file does not exist or could not be opened.

emgrAddGbToEvent()

```
int emgrAddGbToEvent(EmgrEvent_t *pEvent,
                     struct GeneralBlock *pNewGB);
```

The `emgrAddGbToEvent()` function adds a GeneralBlock structure to an event.

Parameters:

**pEvent*	pointer to an event structure
**pNewGB*	pointer to a GeneralBlock structure to add to the event

Return value:

- Success: 0
- Failure:

-1	An unspecified error occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrAddIntIemToEvent()

```
int emgrAddIntItemToEvent(EmgrEvent_t *pEvent,
                          const char *name,
                          long value);
```

The `emgrAddIntItemToEvent()` function converts an integer to a string and adds it to an event.

Parameters:

**pEvent*	pointer to the event structure
**name*	pointer to a character string that contains the name of the item
value	integer value to add

Return value:

- Success: 0
- Failure:

-1	An unspecified error occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrAddItemToEvent()

```
int emgrAddItemToEvent(EmgrEvent_t *pEvent,
                       const char *name,
                       const char *value);
```

The `emgrAddItemToEvent()` function adds an item (named value) to an event.

Parameters:

**pEvent*	pointer to the event structure
**name*	pointer to a character string that contains the name of the item
**value*	pointer to a character string that contains the value of the item

Return value:

- Success: 0
- Failure:

-1	An unspecified error occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrAddSubscribe()

```
EmgrEvent_t *emgrAddSubscribe(EmgrEvent_t *pEvent,
                              const char *appname,
                              int evClass,
                              int evType,
                              const char *source,
                              const char *origin);
```

The `emgrAddSubscribe()` function adds the next subscription specification to an event that was already allocated with the `emgNewSubscribe()` function. The `emgrAddSubscribe()` function also sets specific information for batch subscription processing.

Parameters:

**appname*	pointer to a character string that contains the name of the application that owns the event (has domain over the event class and type) (Set this string to NULL to select events from any application.)
evClass	event class to subscribe (Set this parameter to -1 to select all classes.)
evType	event type to subscribe (Set this parameter to -1 to select all event types.)
**source*	pointer to a character string that contains the name of the host or hosts from which to subscribe events (If the string contains more than one host, separate the hosts with spaces and/or commas. If the string is empty or NULL, events are subscribed from the localhost.)
**origin*	pointer to a character string that contains the name of the application that logs the event (If the application that sends the events also owns the events, set the *origin* and *appname* parameters to the same value or pass an empty string or NULL character pointer to the *origin* parameter.)

Return value:

- Success: Pointer to the event structure
- Failure: NULL pointer

emgrAddTaggedDataToEvent()

```
int emgrAddTaggedDataToEvent(EmgrEvent_t *pEvent,
                             const char *tag,
                             const void *pBuffer,
                             size_t size);
```

The `emgrAddTaggedDataToEvent()` function adds the contents of a data buffer to an event. It also names the data with a tag that you can specify to quickly access the data again. (The tag acts as an item name.)

Parameters:

**pEvent*	pointer to an event structure
**tag*	pointer to a character string that contains the tag
**pBuffer*	pointer to the buffer of data (This pointer must be valid the entire time that the event is in use.)
size	number of bytes of data in the buffer

Return value:

- Success: 0
- Failure:

-1	An unspecified error occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrAddTaggedFileToEvent()

```
int emgrAddTaggedFileToEvent(EmgrEvent_t *pEvent,
                             const char *tag,
                             const char *path);
```

The `emgrAddTaggedFileToEvent()` function adds the contents of a file to an event. It also names the data block with a tag that you specify so you can quickly access the data again. (The tag acts as an item name.)

Parameters:

**pEvent*	pointer to an event structure
**tag*	pointer to a character string that contains the file tag
**path*	pointer to a character string that contains the path to the file

Return value:

- Success: 0
- Failure:

-1	An unspecified error occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrAddUnsubscribe()

```
EmgrEvent_t *emgrAddUnSubscribe(EmgrEvent_t *pEvent,
                                const char *appname,
                                int evClass,
                                int evType,
                                const char *source,
                                const char *origin);
```

The `emgrAddUnSubscribe()` function adds the next unsubscription specification to an event that was already allocated with the `emgrNewUnSubscribe()` function. The `emgrAddUnSubscribe()` function also sets specific information for batch event processing.

Parameters:

**pEvent*	pointer to an event structure
**appname*	pointer to a character string that contains the name of the application that owns the event (has domain over the event class and type) **Note:** Do not set this parameter to an empty string or a NULL character pointer.
evClass	event class to subscribe (Set this parameter to -1 to select all classes.)
evType	event type to subscribe (Set this parameter to -1 to select all event types.)
**source*	pointer to a character string that contains the name of the host or hosts from which to subscribe events (If the string contains more than one host, separate the hosts with spaces and/or commas. If the string is empty or NULL, events are unsubscribed from the localhost.)
**origin*	pointer to a character string that contains the name of the application that logs the event (If the application that sends the events also owns the events, set the *origin* and *appname* parameters to the same value. Set this parameter to empty string or the NULL character pointer to specify any application.)

Return value:

- Success: Pointer to the event structure
- Failure: NULL pointer

emgrAllocEvent()

```
EmgrEvent_t *emgrAllocEvent(const char *appname,
                            int evClass,
                            int evType,
                            int version,
                            char *origin);
```

The `emgrAllocEvent()` function allocates memory for an event.

Parameters:

**appname*	pointer to a character string that contains the name of the application that owns the event
evClass	application-specific event class number (set to 0 if you do not want to specify a class)
evType	application-specific event type number (do not set to 0 or -1)
version	application-specific event version for consumer or producer use (set to 0 if you do not want to specify a version)
**origin*	pointer to a character string that contains the name of the application that logs the event (If it is the same as the application that owns the event, set the string to the same string as *appname*.)

Return value:

- Success: Pointer to the event structure
- Failure: NULL pointer

emgrBuildQSearch()

```
int emgrBuildQSearch(EmgrEvent_t *pEvent);
```

The `emgrBuildQSearch()` function builds the internal search table for an event to enable searches based on item tags. Normally, you do not need to use the function because the Event Manager calls it when necessary on the consumer side.

Parameters:

**pEvent*	pointer to the event structure

Return value:

- Success: 0
- Failure:

-1	An unspecified error occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrCheckEvent()

```
int emgrCheckEvent(const EmgrEvent_t *pEvent);
```

The `emgrCheckEvent()` function verifies that an event structure is valid.

Parameter:

**pEvent*	pointer to an event structure

Return value:

- Success: 0 (event structure is valid)
- Failure:

-1	The event structure is not valid.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrCloneEvent()

`int *emgrCloneEvent(const EmgrEvent_t *`*pEvent*`);`

The `emgrCloneEvent()` function clones all parameters and data from an event, except the eventID, source, timestamp, and uid attributes. This function is useful for DSO consumers that must modify an event and pass it elsewhere for further processing.

Parameters:

**pEvent*	pointer to the event structure

Return value:

- Success: 0
- Failure: NULL pointer (The **pEvent* pointer that was passed to the function points to corrupted memory, or a memory allocation failure occurred.)

emgrCloneGb()

`GeneralBlock_t *emgrCloneGb(const EmgrEvent_t *`*pGb*`);`

The `emgrCloneGb()` function clones the contents of a GeneralBlock structure.

Parameters:

**pGb*	pointer to the GeneralBlock structure

Return value:

- Success: 0
- Failure: NULL pointer (The **pGB* pointer that was passed to the function points to corrupted memory, or a memory allocation failure occurred.)

emgrForwardEvent()

```
int emgrForwardEvent(EmgrEvent_t *pEvent,
                     const char *forwardPath);
```

The `emgrForwardEvent()` function specifies that the Event Manager should forward an event to one or more remote hosts; this function uses the `emgrSendEvent()` function to forward the event to the first host in the forward path.

Parameters:

**pEvent*	pointer to the event structure
**forwardPath*	pointer to a character string
	The character string contains the path of hosts that should receive an event and has the following format:
	hostname1[:*port*]>*hostname2*>[:*port*]>...*hostnameN*[:*port*]
	Example: `host1.sgi.com>host2.sgi.com:5553>host3.sgi.com`

Return value:

- Success: 0
- Failure:

-1	The forward path is invalid or there is a communication error with the first host in the path.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrFreeEvent()

```
int emgrFreeEvent(EmgrEvent_t *pEvent);
```

The `emgrFreeEvent()` function frees up all memory resources allocated to an event.

Parameters:

**pEvent*	pointer to an event structure

Return value:

- Success: 0
- Failure:

-1	An unspecified error occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrGetEventItem()

```
const void *emgrGetEventItem(const EmgrEvent_t *pEvent,
                             const char *name,
                             int *pType,
                             int *pLength);
```

This `emgrGetEventItem()` function returns the value of an item.

Parameters:

**pEvent*	pointer to the event structure
**name*	pointer to a character string that contains the name of the requested item
**pType*	pointer to an integer that holds the type of data that the function retrieves The integer can have the following values: 1 = file data 2 = binary data 3 = string data
**pLength*	pointer to an integer that holds the length of the data that the function retrieves (the length of the string if the function returns a string, the length of the binary large object (BLOB) of data if the function returns binary data, or the file length if the function returns a file)

Return value:

- Success: Pointer to a character array if the item is a string, pointer to a binary array if the item is binary data, or pointer to a `GbFileValue` structure (type `GbFileValue_t`) if the item is a file
- Failure: NULL pointer

emgrGetFirstEventGb()

```
const struct GeneralBlock *emgrGetFirstEventGb(EmgrEvent_t *pEvent);
```

The `emgrGetFirstEventGb()` function returns the first GeneralBlock structure that is attached to an event and initializes an iterator to use with the `emgrGetNextEventGb()` function.

Parameters:

**pEvent*	pointer to the event structure

Return value:

- Success: Pointer to the GeneralBlock structure
- Failure: NULL pointer

emgrGetFirstEventItem()

```
int emgrGetFirstEventItem(EmgrEvent_t *pEvent,
                          const char **pName,
                          const void **pValue,
                          int *pType,
                          int *pLength);
```

The `emgrGetFirstEventItem()` function traverses event data and returns the first item and value of the event data initialized into *pName* and *pValue*. It initializes the iterator.

Parameters:

**pEvent*	pointer to the event structure
***pName*	pointer to the location where the returned item name pointer is stored
***pValue*	pointer to a character array if the item is a string, pointer to a binary array if the item is binary data, or pointer to a `GbFileValue` structure (type `GbFileValue_t`) if the item is a file
**pType*	pointer to an integer that holds the type of data that the function retrieves The integer can have the following values: 1 = file data 2 = binary data 3 = string data
**pLength*	pointer to an integer that holds the length of the data that the function retrieves (the length of the string if the function returns a string, the length of the BLOB of data if the function returns binary data, or the file length if the function returns a file)

Return value:

- Success: 0
- Failure:

-1	An unspecified error occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrGetNextEventGb()

```
const struct GeneralBlock *emgrGetNextEventGb(EmgrEvent_t *pEvent);
```

The `emgrGetFirstEventGb()` function returns the next GeneralBlock structure that is attached to an event.

Parameters:

**pEvent*	pointer to the event structure

Return value:

- Success: Pointer to the GeneralBlock structure
- Failure: NULL pointer

emgrGetNextEventItem()

```
int emgrGetNextEventItem(EmgrEvent_t *pEvent,
                         const char **pName,
                         const void **pValue,
                         int *pType,
                         int *pLength);
```

The `emgrGetNextEventItem()` function traverses the event data and returns the item and value of the next item in the event data to *pName* and *pValue*.

Parameters:

**pEvent*	pointer to the event structure
***pName*	pointer to the location where the returned item name pointer is stored
***pValue*	pointer to a character array if the item is a string, pointer to a binary array if the item is binary data, or pointer to a `GbFileValue` structure (type `GbFileValue_t`) if the item is a file
**pType*	pointer to an integer that holds the type of data that the function retrieves The integer can have the following values: 1 = file data 2 = binary data 3 = string data
**pLength*	pointer to an integer that holds the length of the data that the function retrieves (the length of the string if the function returns a string, the length of the BLOB of data if the function returns binary data, or the file length if the function returns a file)

Return value:

- Success: 0
- Failure:

-1	An unspecified error occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrIsDaemonInstalled()

```
int emgrIsDaemonInstalled();
```

The `emgrIsDaemonInstalled()` function identifies whether the Event Manager server is installed on the local system.

Parameters: none

Return value:

- 0: Event Manager server is installed
- 1: Event Manager server is not installed

Note: This function works only with the default configuration. If you modify how the Event Manager is configured or installed, this function fails.

emgrIsDaemonStarted()

`int emgrIsDaemonStarted(const char *`*server*`);`

The `emgrIsDaemonStarted()` function identifies whether the Event Manager daemon is running on the specified system.

Parameters:

**server*	pointer to a character string that contains the name of the server to check

Return value:

- 0: Event Manager daemon is running on the specified system
- 1: Event Manager daemon is not running on the specified system

Note: This function works only with the default configuration. If you modify how the Event Manager is configured or installed, this function fails.

emgrNewQuery()

```
EmgrEvent_t *emgrNewQuery(const char *appname,
                          int evClass,
                          int evType,
                          const char *source,
                          const char *origin);
```

The `emgrNewQuery()` function is a wrapper to the `emgrAllocEvent()` function. The `emgrNewQuery()` function allocates an event structure and initializes the event header to conduct a query of events that are currently subscribed.

Parameters:

**appname*	pointer to a character string that contains the name of the application that owns the event (has domain over the event class and type) Set this parameter to the NULL character pointer to specify any application
evClass	event class to match (Set this parameter to -1 to select all classes.)
evType	event type to match (Set this parameter to -1 to select all event types.)
**source*	pointer to a character string that contains the name of the host or hosts from which to query events (If the string contains more than one host, separate the hosts with spaces and/or commas. If the string is empty or NULL, events from any source are used.)
**origin*	pointer to a character string that contains the name of the application that logs the event. If the string is empty or NULL, events from any origin are used

Return value:

- Success: Pointer to the event structure
- Failure: NULL pointer

emgrNewSubscribe()

```
EmgrEvent_t *emgrNewSubscribe(const char *appname,
                              int evClass,
                              int evType,
                              const char *source,
                              const char *origin);
```

The `emgrNewSubscribe()` function is a wrapper to the `emgrAllocEvent()` function. The `emgrNewSubscribe()` function allocates an event structure and initializes the event header with the specified data.

Parameters:

**appname*	pointer to a character string that contains the name of the application that owns the event (has domain over the event class and type) (Set this string to NULL to select events from any application.)
evClass	event class to subscribe (Set this parameter to -1 to select all classes.)
evType	event type to subscribe (Set this parameter to -1 to select all event types.)
**source*	pointer to a character string that contains the name of the host or hosts from which to subscribe events (If the string contains more than one host, separate the hosts with spaces and/or commas. If the string is empty or NULL, events from any source are used.)
**origin*	pointer to a character string that contains the name of the application that logs the event (If the application that sends the events also owns the events, set the *origin* and *appname* parameters to the same value. Set this parameter to empty string or the NULL character pointer to specify any origin.)

Return value:

- Success: Pointer to the event structure
- Failure: NULL pointer

emgrNewUnsubscribe()

```
EmgrEvent_t *emgrNewUnSubscribe(const char *appname,
                                int evClass,
                                int evType,
                                const char *source,
                                const char *origin);
```

The `emgrNewUnSubscribe()` function is a wrapper to the `emgrAllocEvent()` function. The `emgrNewUnSubscribe()` function allocates an event structure and initializes the event header for unsubscription using the data provided.

Parameters:

**appname*	pointer to a character string that contains the name of the application that owns the event (has domain over the event class and type) **Note:** Do not set this parameter to an empty string or a NULL character pointer.
evClass	event class to subscribe (Set this parameter to -1 to select all classes.)
evType	event type to subscribe (Set this parameter to -1 to select all event types.)
**source*	pointer to a character string that contains the name of the host or hosts from which to subscribe events (If the string contains more than one host, separate the hosts with spaces and/or commas. If the string is empty or NULL, events are unsubscribed from any host.)
**origin*	pointer to a character string that contains the name of the application that logs the event (If the application that sends the events also owns the events, set the *origin* and *appname* parameters to the same value. Set this parameter to empty string or the NULL character pointer to specify any host.)

Note: If you want to unsubscribe an event, the specified parameters must match the subscription parameters (including the consumer definition).

Return value:

- Success: Pointer to the event structure
- Failure: NULL pointer

emgrPrintEvent()

```
void emgrPrintEvent(const EmgrEvent_t *pEvent,
                    FILE *out);
```

The `emgrPrintEvent()` function prints an event to a FILE stream.

Parameters:

**pEvent*	pointer to the event structure
**out*	pointer to the FILE stream

Return value: None

emgrRunQuery()

```
int emgrRunQuery(EmgrEvent_t *pQueryEvent,
                 const char *host,
                 EmgrEvent_t ***ppRetEvents,
                 int *pEvCount,
                 int timeout);
```

The `emgrRunQuery()` function executes a subscription query.

Parameters:

**appname*	pointer returned by the `emgrNewQuery()` function
**host*	hostname (and optionally port number) of the system to query
evType	event type to match (Set this parameter to -1 to select all event types.)
****ppRetEvents*	array of pointers to events that match the query
**pEvCount*	number of returned events

Note: The calling program must free the memory that is used to store the number of returned events and the array of pointers.

timeout	number of seconds to wait for a return from the Event Manager

Return value:

- Success: 0 (with *ppRetEvents* and *pEvCount* set to values)
- Failure:

-1	An unspecified error occurred.
4	The **pQueryEvent* pointer that was passed to the function points to corrupted memory.

emgrRunSubscribe()

```
int emgrRunSubscribe(EmgrEvent_t *pSubscrEvent,
                     const char *host,
                     int timeout,
                     char **pRetEventMask);
```

The `emgrRunSubscribe()` function adds subscription attributes to an event (Use this function instead of the `emgrSendEvent()` function if operation completion status is needed.) This function can subscribe multiple events at a time.

Parameters:

**pSubscrEvent*	pointer to the event structure
**host*	pointer to a character string that contains the hostname
timeout	number of seconds to wait for a return from the Event Manager
`**`*pRetEventMask*	address of a variable that returns the subscription status (each element contains '0' plus the subscription status)

Return value:

- Success: 0 (with *pRetEventMask* set to a value)
- Failure:

-1	An unspecified error occurred.
4	The **pSubscrEvent* pointer that was passed to the function points to corrupted memory.

emgrRunUnSubscribe()

```
int emgrRunUnSubscribe(EmgrEvent_t *pSubscrEvent,
                       const char *host,
                       int timeout,
                       char **pRetEventMask);
```

The `emgrRunUnSubscribe()` function unsubscribes events. (Use this function instead of the `emgrSendEvent()` function if operation completion status is needed.) This function can unsubscribe multiple events at a time.

Parameters:

**pSubscrEvent*	pointer to the event structure
**host*	pointer to a character string that contains the hostname
timeout	number of seconds to wait for a return from the Event Manager
`**`*pRetEventMask*	address of a variable that returns the subscription status (each element contains ‘0’ plus the subscription status)

Return value:

- Success: 0 (with *pRetEventMask* set to a value)
- Failure:

-1	An unspecified error occurred.
4	The **pSubscrEvent* pointer that was passed to the function points to corrupted memory.

emgrSearchGb()

```
const struct GeneralBlock *emgrSearchGb(EmgrEvent_t *pEvent,
                                        const char *tag);
```

The `emgrSearchGb()` function locates the GeneralBlock referenced by a tag that you specify.

Parameters:

**pEvent*	pointer to the event structure
**tag*	pointer to a character string that contains the tag for the GeneralBlock that you want to locate

Return value:

- Success: Pointer to the GeneralBlock structure
- Failure: NULL pointer

emgrSendEvent()

```
int *emgrSendEvent(EmgrEvent_t *pEvent,
                   const char *host);
```

The `emgrSendEvent()` sends an event to the Event Manager on the specified host.

Parameters:

**pEvent*	pointer to the event structure
**host*	pointer to a character string that contains the hostname of the Event Manager used to subscribe events. The character string uses the following format: <*hostname*>[:<*port_number*>] If you specify NULL or an empty string, the function subscribes the event with the Event Manager on the local system.

Return value:

- Success: 0
- Failure:

-1	An unspecified error occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrSetToForward()

```
int emgrSetToForward(EmgrEvent_t *pEvent,
                     const char *forwardPath);
```

The `emgrSetToForward()` function specifies that the Event Manager should forward an event to one or more remote hosts.

Parameters:

**pEvent*	pointer to the event structure
**forwardPath*	pointer to a character string The character string contains the path of hosts that should receive an event and has the following format: *hostname1*[:*port*]>*hostname2*>[:*port*]>...*hostnameN*[:*port*] Example: `host1.sgi.com>host2.sgi.com:5553>host3.sgi.com`

Return value:

- Success: 0
- Failure:

-1	An unspecified error occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrShmCliInitEvent()

```
EmgrEvent_t *emgrShmCliInitEvent(int argc,
                                 const char *argv[],
                                 int *pError);
```

The `emgrShmCliInitEvent()` function is a wrapper to the `emgrShmInitEvent()` function. It hides the details and simplifies command-line option parsing to `main()`, searches for a shared memory option, and calls the `emgrShmInitEvent()` function.

Parameters:

argc	number of arguments
**argv*[]	pointer to the array of arguments
**pError*	pointer to the error code when an error occurs

Return value:

- Success: Pointer to the initialized event structure
- Failure: NULL pointer (**pError* points to the error code.)

emgrShmInitEvent()

```
EmgrEvent_t *emgrShmInitEvent(int shmId,
                              int *pError);
```

The `emgrShmInitEvent()` function initializes an event from shared memory that the Event Manager allocated.

Parameters:

shmId	shared memory ID (that was passed to the consumer process as a command-line parameter)
**pError*	pointer to the error code when an error occurs

Return value:

- Success: Pointer to the initialized event structure
- Failure: NULL pointer (**pError* points to the error code.)

  The error is the system error number from the *errNO* global variable. (Refer to the `/usr/include/errno.h` and `/usr/include/linux/errno.h` files for more information.)

emgrSubscribeSpecCntFreq()

```
int emgrSubscribeSpecCntFreq(EmgrEvent_t *pEvent,
                             int freq);
```

The `emgrSubscribeSpecCntFreq()` function is a wrapper to the `emgrAddItemToEvent()` function. The `emgrSubscribeSpecCntFreq()` function adds a tagged item to a subscription event to specify how often the Event Manager should send a matching event to a matching subscriber.

Parameters:

**pEvent*	pointer to the event structure
freq	count frequency value (The Event Manager sends one of this number of events to the subscriber; for example, if you set *freq* to 5, the Event Manager sends every fifth matching event to the subscriber.)

Return value:

- Success: 0
- Failure:

-1	A memory allocation failure occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrSubscribeSpecDsoConsumer()

```
int emgrSubscribeSpecDsoConsumer(EmgrEvent_t *pEvent,
                                 const char *dsoPath,
                                 const char *callName,
                                 const char *prmSpec);
```

The `emgrSubscribeSpecDsoConsumer()` function subscribes events from consumers that are implemented as dynamic shared object (DSO) libraries that are called from the Event Manager server.

Parameters:

**pEvent*	pointer to the event structure
**dsoPath*	pointer to a character string that contains the pathname of the consumer DSO library
**callName*	pointer to a character string that contains the name of the main consumer function to call
**prmSpec*	pointer to a character string of parameters for the consumer

Return value:

- Success: 0
- Failure:

-1	An unspecified error occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrSubscribeSpecExecConsumer()

```
int emgrSubscribeSpecExecConsumer(EmgrEvent_t *pEvent,
                                  const char *execPath,
                                  const char *prmSpec);
```

The `emgrSubscribeSpecExecConsumer()` function subscribes events from applications that are launched with the `fork()` and `exec()` commands. Event parameters are passed to the consumer on the command line.

Parameters:

**pEvent*	pointer to the event structure
**execPath*	pointer to a character string that contains the pathname of the consumer application to launch
**prmSpec*	pointer to a string of parameters for the application

Return value:

- Success: 0
- Failure:

-1	An unspecified error occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrSubscribeSpecExecShMemConsumer()

```
int emgrSubscribeSpecExecShMemConsumer(EmgrEvent_t *pEvent,
                                       const char *execPath,
                                       const char *prmSpec);
```

The `emgrSubscribeSpecExecShMemConsumer()` function subscribes events from consumer applications that are launched via `fork()` and `exec()` commands. Event parameters are passed to the consumer applications via shared memory handoffs handled by the API layer.

Parameters:

**pEvent*	pointer to the event structure
**execPath*	pointer to a character string that contains the pathname of the consumer application to launch
**prmSpec*	pointer to a string of parameters for the application

Return value:

- Success: 0
- Failure:

-1	An unspecified error occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrSubscribeSpecFacility()

```
int emgrSubscribeSpecFacility(EmgrEvent_t *pEvent,
                              int facility);
```

The `emgrSubscribeSpecFacility()` function is a wrapper to the `emgrAddItemToEvent()` function. The `emgrSubscribeSpecFacility()` function adds a tagged item to a subscription or unsubscription event to specify an optional event facility filter (used for subscription matching).

Parameters:

**pEvent*	pointer to the event structure
facility	facility ID value (same as `syslog` facility value)

Return value:

- Success: 0
- Failure:

-1	A memory allocation failure occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrSubscribeSpecForwardConsumer()

```
int emgrSubscribeSpecForwardConsumer(EmgrEvent_t *pEvent,
                                     const char *forwardPath);
```

The `emgrSubscribeSpecForwardConsumer()` function specifies that the Event Manager should forward an event to another host for processing.

Parameters:

**pEvent*	pointer to the event structure
**forwardPath*	pointer to a character string The character string contains the path of hosts that should receive an event and has the following format: *hostname1*[:*port*]>*hostname2*>[:*port*]>...*hostnameN*[:*port*] Example: `host1.sgi.com>host2.sgi.com:5553>host3.sgi.com`

Return value:

- Success: 0
- Failure:

-1	An unspecified error occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrSubscribeSpecPriority()

```
int emgrSubscribeSpecPriority(EmgrEvent_t *pEvent,
                              int pri);
```

The `emgrSubscribeSpecPriority()` function is a wrapper to the `emgrAddItemToEvent()` function. The `emgrSubscribeSpecPriority()` function adds a tagged item to a subscription or unsubscription event to specify an optional event priority filter for subscription matching.

Parameters:

**pEvent*	pointer to the event structure
pri	priority value (same as `syslog` priority value)

Return value:

- Success: 0
- Failure:

-1	A memory allocation failure occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrSubscribeSpecRegexpMap()

```
int emgrSubscribeSpecRegexpMap(EmgrEvent_t *pEvent,
                               const char *regExp,
                               int evMapClass,
                               int evMapType);
```

The `emgrSubscribeSpecRegexpMap()` function is a wrapper to the `emgrAddItemToEvent()` function. The `emgrSubscribeSpecRegexpMap()` function adds a tagged item to a subscription event to specify an optional untagged event's class and type mapping before forwarding it to a subscribed consumer.

Parameters:

**pEvent*	pointer to the event structure
**regExp*	regular expression to compare to an event’s message body
evMapClass	class ID to add to the event
evMapType	type ID to add to the event

Return value:

- Success: 0
- Failure:

-1	A memory allocation failure occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

emgrSubscribeSpecTimeFreq()

```
int emgrSubscribeSpecTimeFreq(EmgrEvent_t *pEvent,
                              int freq);
```

The `emgrSubscribeSpecTimeFreq()` function is a wrapper to the `emgrAddItemToEvent()` function. The `emgrSubscribeSpecTimeFreq()` function adds a tagged item to a subscription event that specifies how often (in events/second) a matching event should be sent to a subscribed consumer. This function enables you to limit (or throttle) the number of events that are sent to a consumer each second.

The Event Manager divides the actual number of events that it receives in a second by the frequency value (*freq*) and rounds the value down to the nearest integer value; the Event Manager sends the resulting number of events to the subscribed consumer each second. Table 2-3 shows the number of events that are sent to a consumer for various example frequency (*freq*) values.

Table 2-4 Event Frequency Examples

Number of Events Received by the Event Manager (Per Second)	**Number of Events Sent to Subscribed Consumer (Per Second)** *freq* **= 1**	*freq* **= 2**	*freq* **= 3**	*freq* **= 4**	*freq* **= 5**
0	0	0	0	0	0
1	1	0	0	0	0
2	2	1	0	0	0
3	3	1	1	0	0
4	4	2	1	1	0
5	5	2	1	1	1
6	6	3	2	1	1
7	7	3	2	1	1
8	8	4	2	2	1
9	9	4	3	2	1
10	10	5	3	2	2
11	11	5	3	2	2

Table 2-4 Event Frequency Examples **(continued)**

Number of Events Received by the Event Manager (Per Second)	**Number of Events Sent to Subscribed Consumer (Per Second)** *freq* **= 1**	*freq* **= 2**	*freq* **= 3**	*freq* **= 4**	*freq* **= 5**
12	12	6	4	3	2
13	13	6	4	3	2
14	14	7	4	3	2
15	15	7	5	3	3
16	16	8	5	4	3
17	17	8	5	4	3
18	18	9	6	4	3
19	19	9	6	4	3
20	20	10	6	5	4

Parameters:

**pEvent*	pointer to the event structure
freq	frequency value

Return value:

- Success: 0
- Failure:

-1	A memory allocation failure occurred.
4	The **pEvent* pointer that was passed to the function points to corrupted memory.

getConfigValue()

```
typedef void *SearchMarker_t;

SearchMarker_t getConfigValue(const char *key,
                              const char **value,
                              SearchMarker_t from);
```

The `getConfigValue()` function retrieves the configuration value for a specified item.

Parameters:

**key*	pointer to a character string that contains the search key
***value*	pointer to the location where the result string pointer is stored
from	token that specifies where the function should begin its search (Set this parameter to NULL for the first `getConfigValue()` function call and set it to the return value from the previous `getConfigValue()` function call to continue searching from the point where the previous search ended.)

Return value:

- Success: A token that indicates the location to begin the search the next time that the function is called.
- Failure: None

Creating Producer, Subscriber, and Consumer Applications

This chapter covers the following topics:

- Creating a producer application
- Creating a subscriber application
- Creating a consumer application

Creating a Producer Application

Producer applications must include Event Manager application programming interface (API) function calls that create and submit events to the Event Manager.

Figure 3-1 summarizes the steps necessary to create an event and send it to the Event Manager. The text following the figure provides detailed information about each step. Refer to Chapter 2, "Event Manager API," for specific information about the individual functions.

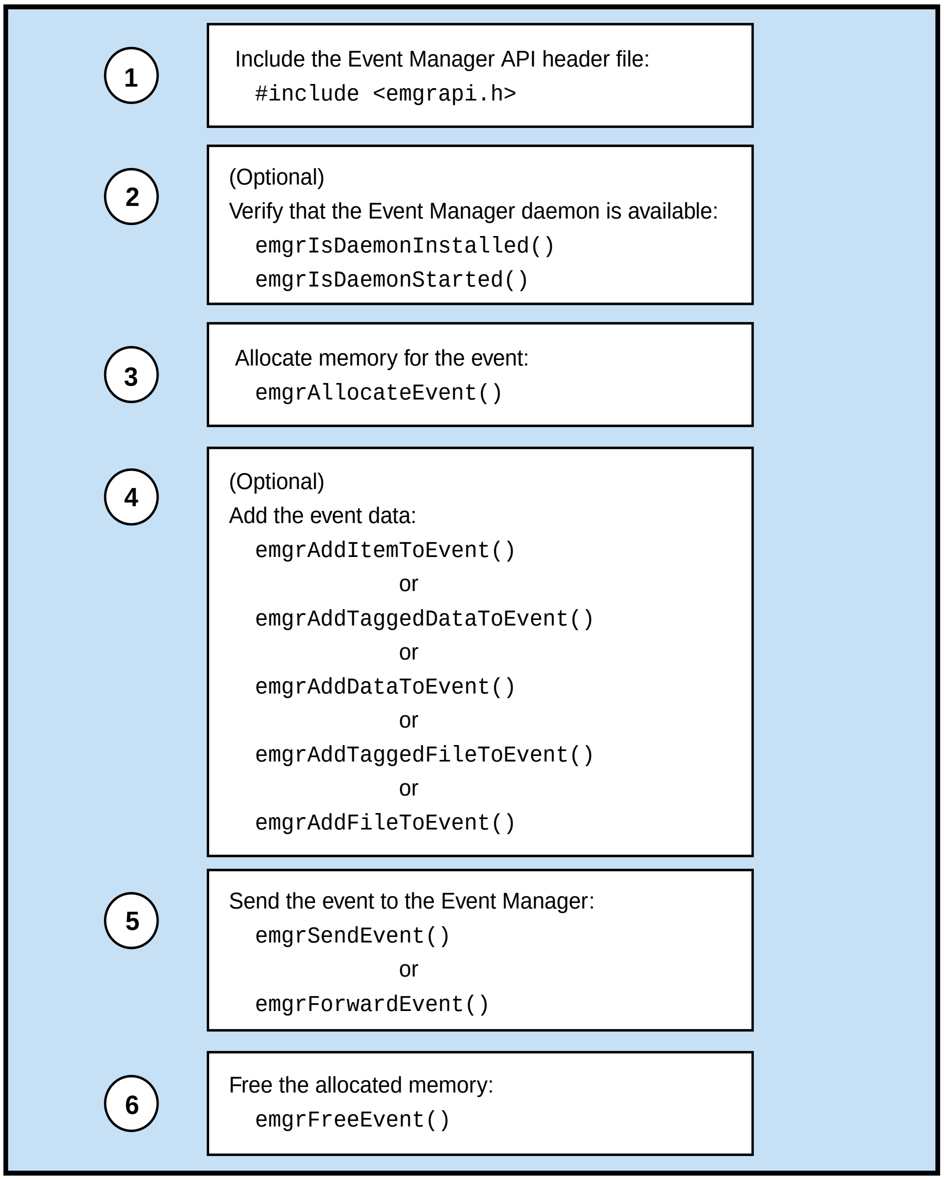


Figure 3-1 Creating and Submitting an Event from a Producer Application

1. Include the Event Manager API header file so that you can access the Event Manager API functions:

   ```
   #include <emgrapi.h>
   ```

2. Verify that the Event Manager daemon is available:

   - Use the `emgrIsDaemonInstalled()` function to verify that the `eventmond` daemon is installed on the system.
   - Use the `emgrIsDaemonStarted()` function to verify that the emgr daemon is running so that the producer application can send event data to it.

   Note: This step is optional. These functions work only with the default configuration; if you modify how the Event Manager is installed or configured, these functions may fail.

   ---

3. Use the `emgrAllocEvent()` function to allocate memory for the event.

4. Add information to the event:

   Use the `emgrAddItemToEvent()` function to add a character name-value.

   or

   Use the `emgrAddTaggedDataToEvent()` or `emgrAddDataToEvent()` function to add binary data to the event.

   or

   Use the `emgrAddTaggedFileToEvent()` or `emgrAddFileToEvent()` function to add a file from the local filesystem.

   Tips:

   Normally, you should use the `Tagged` version of the commands because tagged data can be accessed faster.

   Be careful with data that you added to an event using the `emgrAddTaggedDataToEvent()` and `emgrAddDataToEvent()` functions. The API does not free any passed pointers; you must keep the pointers valid until you send the event information and free memory.

   Note: This step is optional. You can create an event that has no data; however, normally, you should attach data to an event before you send the event to the Event Manager.

   ---

5. Use the `emgrSendEvent()` or `emgrForwardEvent()` function to send the event to the Event Manager daemon (`eventmond`).
6. Use the `emgrFreeEvent()` function to free the memory that you allocated for the event.

Example 3-1 shows an example producer that allocates an event, adds several types of data to it, sends the event to Event Manager, and frees the memory allocated to the event.

Example 3-1 Example Producer Code

```
#include <stdio.h>
#include <string.h>

#include <emgrapi.h>

main()
{
        EmgrEvent_t *e;
        int ret;
        char *err, *val;

        /*--- Define data to send ---*/

        char *NAME1 = "HDSIZE", *NAME2 = "MEMSIZE";
        char *VALUE1 = "30GB", *VALUE2 = "256MB";
        int class=123,type=456,version=0;
        char *origin = "syslog";
        char *appname = "unix";
        char *databuf;
        int datasz;

        /*---- Initialize the event header and body----*/

        e = emgrAllocEvent(class,type,version,origin,appname);

        /*--- Add data to event---*/

        if(emgrAddItemToEvent(e,NAME1, NULL)!= 0) {
               printf("Error.\n");
        }

        if(emgrAddItemToEvent(e,NAME2, VALUE2)!= 0) {
               printf("Error.\n");
        }


        /*--- Add file to event ---*/

        if(ret = emgrAddFileToEvent(e,"/tmp/testfile"))
                printf("\n Failure adding file.\n");

        /*--- Add binary data to event ---*/

        /* NOTE: databuf memory will not be freed by emgrFreeEvent()*/

        if(ret = emgrAddDataToEvent(e,databuf, datasz))
                printf("\n  Failure adding binary data.\n");


        /*--- Send the event to the Event Manager ---*/

        if((err = emgrSendEvent(e,NULL)) != 0) {
                printf("\n Failure sending data: errcode %d\n",ret);


           /*--- Free the memory allocated to the event ---*/

        emgrFreeEvent(e);
}
```

Creating a Subscriber Application

Subscriber applications perform the following functions:

- Creating, updating, and submitting subscription events
- Creating, updating, and submitting unsubscription events

Subscription events indicate that the Event Manager should send information about a specific event to a specific consumer application. Subscription events specify how the Event Manager should notify the consumer about the event (load a consumer DSO, send event information to an executable application via shared memory, send event information to an executable application via command-line options, or forward the event to a consumer application on another system). Several Event Manager API functions are available to configure how and when the Event Manager should send event information to consumer applications.

Unsubscription events indicate that a consumer no longer needs to receive information about a specific event. When the Event Manager receives an unsubscription event, it stops sending information about the specified event to the specified consumer.

Creating, Modifying, and Submitting Subscription Events

You must subscribe a consumer to an event to enable the consumer application to receive event information from the `eventmond` daemon. You do this by creating a subscription event and submitting it to the `eventmond` daemon.

Figure 3-2 summarizes the steps needed to create, modify, and submit subscription events. The text following the figure provides detailed information about each step. Refer to Chapter 2, “Event Manager API,” for specific information about the individual functions.

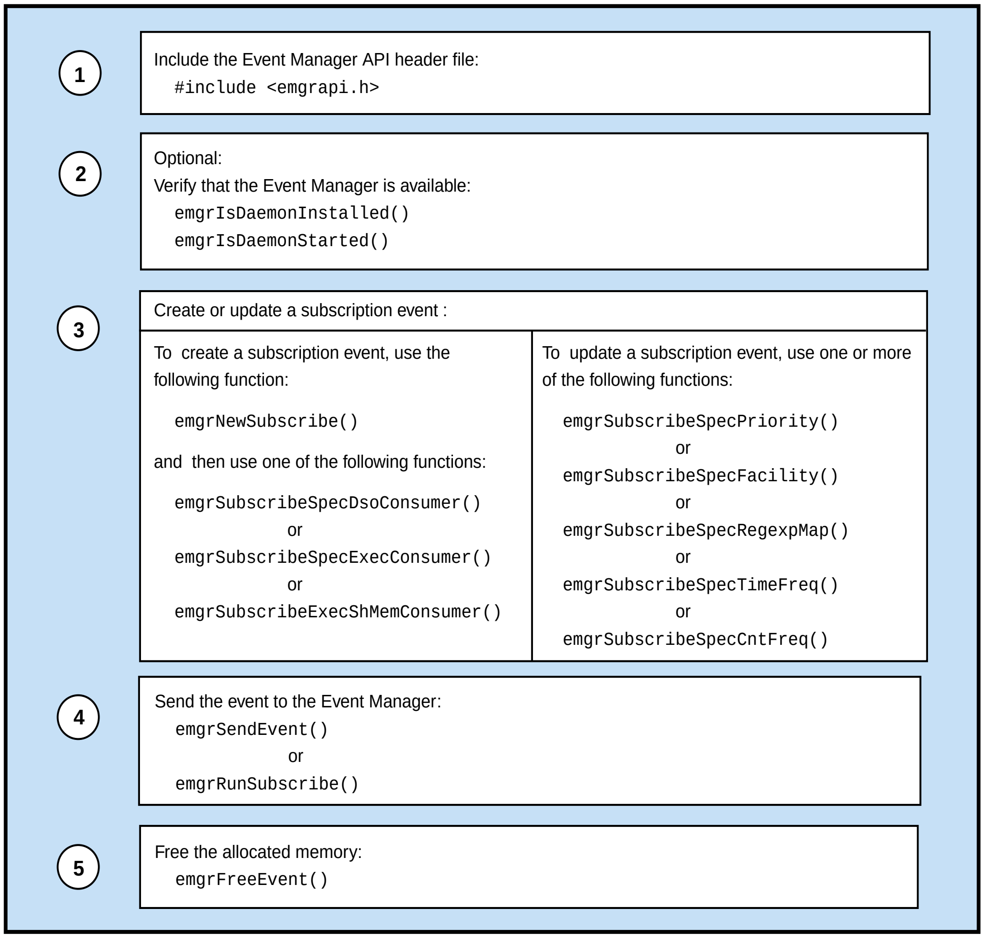

1
Include the Event Manager API header file:
#include <emgrapi.h>
2
Optional:
Verify that the Event Manager is available:
emgrIsDaemonInstalled()
emgrIsDaemonStarted()
3
Create or update a subscription event :
To create a subscription event, use the following function:
emgrNewSubscribe()
and then use one of the following functions:
emgrSubscribeSpecDsoConsumer()
or
emgrSubscribeSpecExecConsumer()
or
emgrSubscribeExecShMemConsumer()
To update a subscription event, use one or more of the following functions:
emgrSubscribeSpecPriority()
or
emgrSubscribeSpecFacility()
or
emgrSubscribeSpecRegexpMap()
or
emgrSubscribeSpecTimeFreq()
or
emgrSubscribeSpecCntFreq()
4
Send the event to the Event Manager:
emgrSendEvent()
or
emgrRunSubscribe()
5
Free the allocated memory:
emgrFreeEvent()

Figure 3-2 Creating/Updating and Submitting a Subscription Event from a Subscriber Application

1. Include the Event Manager API header file so that you can access the Event Manager API functions:

   `#include <emgrapi.h>`

2. Verify that the Event Manager daemon is available:

   - Use the `emgrIsDaemonInstalled()` function to verify that the `eventmond` daemon is installed on the system.
   - Use the `emgrIsDaemonStarted()` function to verify that the `eventmond` daemon is running so the producer application can send event data to it.

   Note: This step is optional. These functions work only with the default configuration; if you modify how the Event Manager is installed or configured, these functions may fail.

   ---

3. Create or update the subscription event:

   To create a subscription event, perform the following actions:

   - Use the `emgrNewSubscribe()` function to allocate a new subscription event structure and initialize the event header with data.
   - Perform one of the following actions to subscribe a consumer to the event:

     Use the `emgrSubscribeSpecDsoConsumer()` function to subscribe events from consumers that are implemented as distributed shared object (DSO) libraries that are called from the Event Manager server.

     or

     Use the `emgrSubscribeSpecExecConsumer()` function to subscribe events from applications that execute through the `fork()` or `exec()` command. (Event parameters pass to the consumer through the command line.)

     or

     Use the `emgrSubscribeExecShMemConsumer()` function to subscribes events from consumer applications that execute through the `fork()` or `exec()` commands and use shared memory. (Event parameters pass to the consumer applications via shared memory handoffs handled by the API layer.)

To update a subscription event, perform one or more of the following actions:

- Use the `emgrSubscribeSpecPriority()` function to add a tagged item to a subscription event to specify an optional event priority filter for subscription matching.
- Use the `emgrSubscribeSpecFacility()` function to add a tagged item to a subscription event to specify an optional event facility filter for subscription matching.
- Use the `emgrSubscribeSpecRegexpMap()` function to add a tagged item to a subscription event to specify an optional untagged event's class and type mapping before forwarding it to a subscribed consumer.
- Use the `emgrSubscribeSpecTimeFreq()` function to add a tagged item to a subscription event that specifies how often (events/second) a matching event should be sent to a matching subscriber.
- Use the `emgrSubscribeSpecCntFreq()` function to add a tagged item to a subscription event to specify how often (one out of *n* events) a matching event should be sent to a matching subscriber.

4. Use the `emgrSendEvent()` or `emgrRunSubscribe()` function to send the event to the Event Manager daemon (`eventmond`).
5. Use the `emgrFreeEvent()` function to free the memory that you allocated for the event.

Examples

The following examples show how to subscribe various types of consumer applications to events.

Example 3-2 Example Code to Subscribe a DSO Consumer

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

#include <string.h>

#include "emgrapi.h"

main()
{
   int i = 0;

   const char *host = "localhost";

   const char *sAppName = "tstApp";
   int         sClass   = 123;
   int         sType    = 345;
   const char *sSource  = NULL;
   const char *sOrigin  = NULL;

   const char *sDsoPath = "./libtstdso.so";
   const char *sDsoFunc = "TstDso";
   const char *sDsoPrms = "p1,p2,p3";


   EmgrEvent_t *pSubscrEvent =
    emgrNewSubscribe(sAppName, sClass, sType, sSource, sOrigin );

   emgrSubscribeSpecDsoConsumer(pSubscrEvent,
   sDsoPath, sDsoFunc, sDsoPrms);

   emgrSendEvent(pSubscrEvent, host);

   emgrFreeEvent(pSubscrEvent);
}
```

Example 3-3 Example Code to Subscribe an Executable Consumer

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

#include <string.h>

#include <emgrapi.h>

main()
{

   int i = 0;

   const char *host = "localhost";

   const char *sAppName = "tstApp";
   int         sClass   = 123;
   int         sType    = 345;
   const char *sSource  = NULL;
   const char *sOrigin  = NULL;

   const char *sExecPath = "/bin/ls";
   const char *sExecPrms = "-l";


   EmgrEvent_t *pSubscrEvent =
    emgrNewSubscribe( sAppName, sClass, sType, sSource, sOrigin );

    emgrSubscribeSpecExecConsumer( pSubscrEvent, sExecPath, sExecPrms);

   emgrSendEvent(pSubscrEvent, host);

   emgrFreeEvent(pSubscrEvent);
}
```

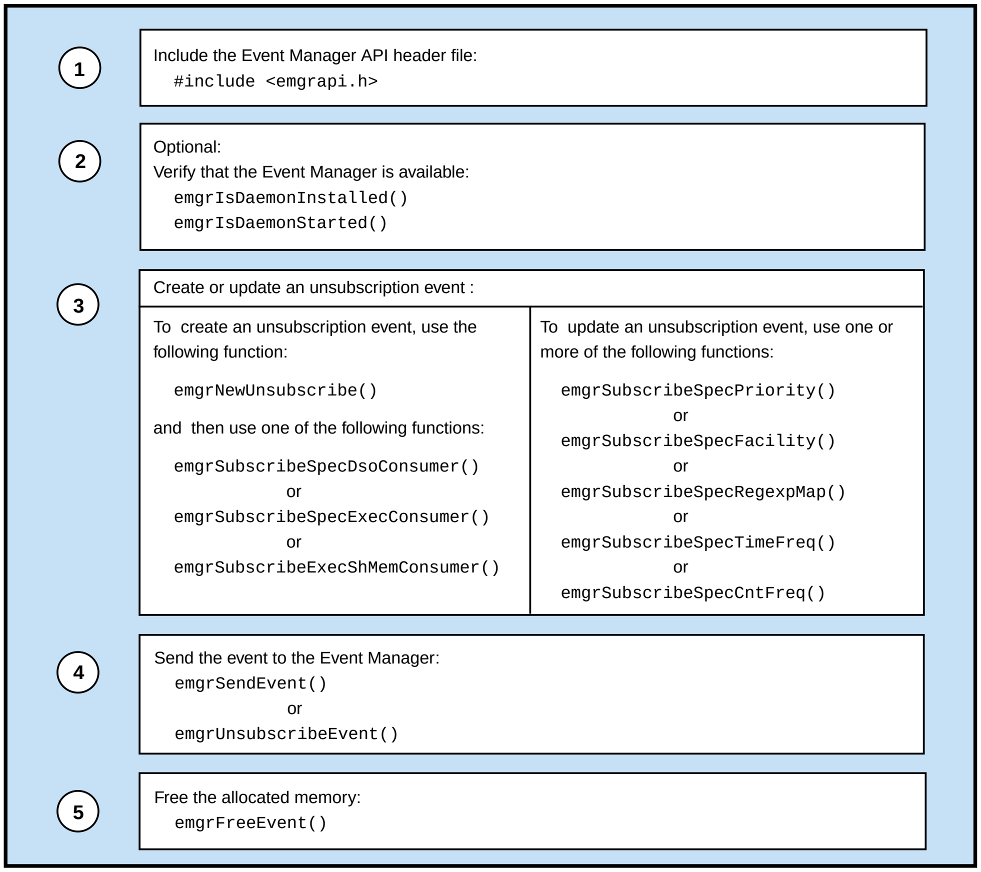
Example 3-4 Example Code to Subscribe a Shared Memory Consumer

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

#include <string.h>

#include <emgrapi.h>

main()
{
   int i = 0;

   const char *sAppName = "tstApp";
   int         sClass   = 123;
   int         sType    = 345;
   const char *sSource  = "minsk-linux.csd.sgi.com";
   const char *sOrigin  = "tstApp";

   const char *sExecPath = "tstShmExec";


   EmgrEvent_t *pSubscrEvent =
    emgrNewSubscribe(sAppName, sClass, sType, sSource, sOrigin ) ;
    emgrSubscribeSpecExecShMemConsumer(pSubscrEvent, sExecPath, NULL);

   emgrSendEvent(pSubscrEvent, NULL);

   emgrFreeEvent(pSubscrEvent);
}
```

Creating, Modifying, and Submitting Unsubscription Events

When a consumer no longer requires information about an event from the Event Manager, you should unsubscribe the event for that consumer. You do this by creating an unsubscription event and sending it to the `eventmond` daemon.

Figure 3-3 summarizes the steps necessary to create, modify, and submit unsubscription events. The text following the figure provides detailed information about each step. Refer to Chapter 2, "Event Manager API," for specific information about the individual functions.



1 Include the Event Manager API header file:

```
#include <emgrapi.h>
```

2 Optional:
Verify that the Event Manager is available:

```
emgrIsDaemonInstalled()
emgrIsDaemonStarted()
```

3 Create or update an unsubscription event :

To create an unsubscription event, use the following function:	To update an unsubscription event, use one or more of the following functions:

`emgrNewUnsubscribe()`
and then use one of the following functions:
`emgrSubscribeSpecDsoConsumer()`
or
`emgrSubscribeSpecExecConsumer()`
or
`emgrSubscribeExecShMemConsumer()`
	`emgrSubscribeSpecPriority()`
	or
	`emgrSubscribeSpecFacility()`
	or
	`emgrSubscribeSpecRegexpMap()`
	or
	`emgrSubscribeSpecTimeFreq()`
	or
	`emgrSubscribeSpecCntFreq()`

4 Send the event to the Event Manager:

```
emgrSendEvent()
        or
emgrUnsubscribeEvent()
```

5 Free the allocated memory:

```
emgrFreeEvent()
```

Figure 3-3 Creating/Updating and Submitting an Unsubscription Event from a Subscriber Application

1. Include the Event Manager API header file so that you can access the Event Manager API functions:

   ```
   #include <emgrapi.h>
   ```

2. Verify that the Event Manager daemon is available:

   - Use the `emgrIsDaemonInstalled()` function to verify that the `eventmond` daemon is installed on the system
   - Use the `emgrIsDaemonStarted()` function to verify that the emgr daemon is running so the producer application can send event data to it.

   Note: This step is optional. These functions work only with the default configuration; if you modify how the Event Manager is installed or configured, these functions may fail.

3. Create/update the unsubscription event:

   To create an unsubscription event, perform the following actions:

   - Use the `emgrNewUnsubscribe()` function to allocate a new unsubscription event structure and initialize the event header with data.
   - Perform one of the following actions to unsubscribe a consumer from an event:

     Use the `emgrSubscribeSpecDsoConsumer()` function to unsubscribe events from consumers that are implemented as distributed shared object (DSO) libraries that are called from the Event Manager server.

     or

     Use the `emgrSubscribeSpecExecConsumer()` function to unsubscribe events from applications that execute through the `fork()` or `exec()` command. (Event parameters pass to the consumer through the command line.)

     or

     Use the `emgrSubscribeExecShMemConsumer()` function to unsubscribe events from consumer applications that execute through the `fork()` or `exec()` commands and use shared memory. (Event parameters pass to the consumer applications via shared memory handoffs handled by the API layer.)

To update an unsubscription event, perform one or more of the following actions:

- Use the `emgrSubscribeSpecPriority()` function to add a tagged item to an unsubscription event to specify an optional event priority filter for subscription matching.
- Use the `emgrSubscribeSpecFacility()` function to add a tagged item to an unsubscription event to specify an optional event facility filter for subscription matching.
- Use the `emgrSubscribeSpecRegexpMap()` function to add a tagged item to an unsubscription event to specify an optional untagged event's class and type mapping before forwarding it to a subscribed consumer.
- Use the `emgrSubscribeSpecTimeFreq()` function to add a tagged item to an unsubscription event that specifies how often (events/second) a matching event should be sent to a matching subscriber.
- Use the `emgrSubscribeSpecCntFreq()` function to add a tagged item to an unsubscription event to specify how often (1/n) a matching event should be sent to a matching subscriber.

4. Use the `emgrSendEvent()` or `emgrRunUnsubscribe()` function to send the event to the Event Manager daemon (`eventmond`).

Examples

The unsubscription code must contain the same components as the subscribe code (except that the `emgrNewUnsubscribe()` function replaces the `emgrNewSubscribe()` function). The following examples show how to unsubscribe various types of consumer applications from events.

Example 3-5 Example Code to Unsubscribe a DSO Consumer

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

#include <string.h>

#include "emgrapi.h"

main()
{
   int i = 0;

   const char *host = "localhost";

   const char *sAppName = "tstApp";
   int         sClass   = 123;
   int         sType    = 345;
   const char *sSource  = NULL;
   const char *sOrigin  = NULL;

   const char *sDsoPath = "./libtstdso.so";
   const char *sDsoFunc = "TstDso";
   const char *sDsoPrms = "p1,p2,p3";

   EmgrEvent_t *pUnsubscrEvent =
    emgrNewUnsubscribe( sAppName, sClass, sType, sSource, sOrigin );

   emgrSubscribeSpecDsoConsumer( pUnsubscrEvent,
   sDsoPath, sDsoFunc, sDsoPrms);

   emgrSendEvent(pUnsubscrEvent, host);

   emgrFreeEvent(pUnsubscrEvent);
}
```

Example 3-6 Example Code to Unsubscribe an Executable Consumer

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

#include <string.h>

#include <emgrapi.h>

main()
{

  int i = 0;

  const char *host = "localhost";

  const char *sAppName = "tstApp";
  int         sClass   = 123;
  int         sType    = 345;
  const char *sSource  = NULL;
  const char *sOrigin  = NULL;

  const char *sExecPath = "/bin/ls";
  const char *sExecPrms = "-l";


  EmgrEvent_t *pUnsubscrEvent =
  emgrNewUnsubscribe( sAppName, sClass, sType, sSource, sOrigin );

  emgrSubscribeSpecExecConsumer( pUnsubscrEvent, sExecPath, sExecPrms);

  emgrSendEvent(pUnsubscrEvent, host);

  emgrFreeEvent(pUnsubscrEvent);

}
```

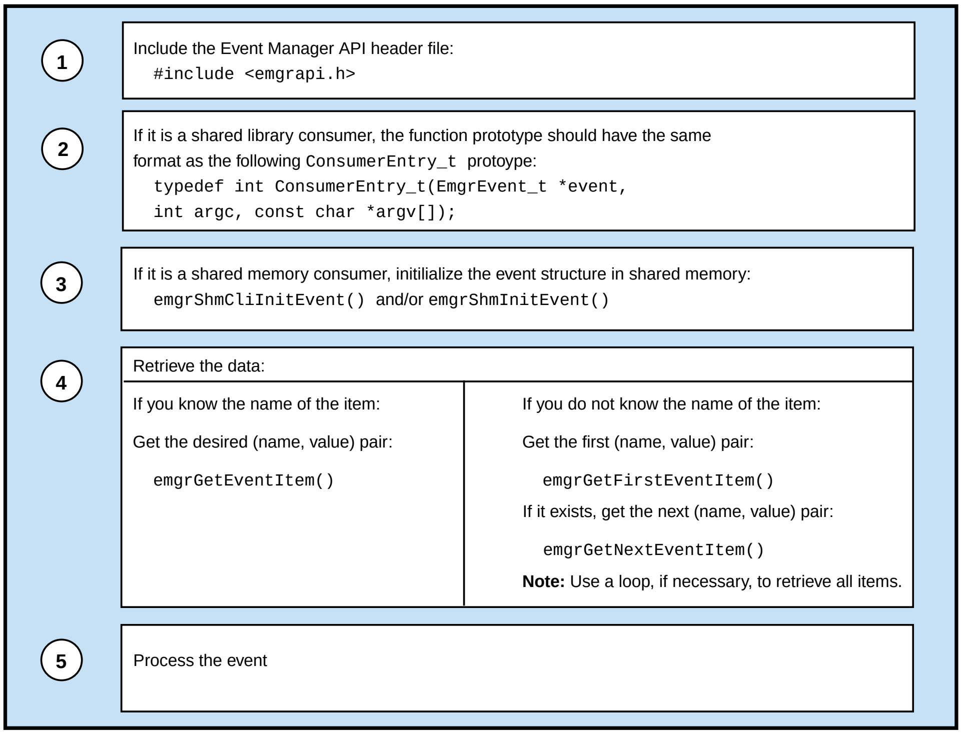
Example 3-7 Example Code to Unsubscribe a Shared Memory Consumer

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

#include <string.h>

#include <emgrapi.h>

main()
{
  int i = 0;

  const char *sAppName = "tstApp";
  int         sClass   = 123;
  int         sType    = 345;
  const char *sSource  = "minsk-linux.csd.sgi.com";
  const char *sOrigin  = "tstApp";

  const char *sExecPath = "tstShmExec";

  EmgrEvent_t *pUnsubscrEvent =
  emgrNewUnsubscribe(sAppName, sClass, sType, sSource, sOrigin ) ;
  emgrSubscribeSpecExecShMemConsumer(pUnsubscrEvent, sExecPath, NULL);

  emgrSendEvent(pUnsubscrEvent, NULL);

  emgrFreeEvent(pUnsubscrEvent);
}
```

Creating a Consumer Application

When the Event Manager detects an event, it compares the event with the current subscription parameters; if there is a match, the Event Manager executes the proper consumer (using the method specified in the subscription event for the consumer) to send the event to it. Then, the consumer can use API functions to access the event payload (data).

Figure 3-4 summarizes the steps necessary to access the event payload. The text following the figure provides detailed information about each step. Refer to Chapter 2, "Event Manager API," for specific information about the individual functions.



1 Include the Event Manager API header file:

```
#include <emgrapi.h>
```

2 If it is a shared library consumer, the function prototype should have the same format as the following `ConsumerEntry_t` protoype:

```
typedef int ConsumerEntry_t(EmgrEvent_t *event,
int argc, const char *argv[]);
```

3 If it is a shared memory consumer, initilialize the event structure in shared memory:
`emgrShmCliInitEvent()` and/or `emgrShmInitEvent()`

4 Retrieve the data:

If you know the name of the item:

Get the desired (name, value) pair:

`emgrGetEventItem()`

If you do not know the name of the item:

Get the first (name, value) pair:

`emgrGetFirstEventItem()`

If it exists, get the next (name, value) pair:

`emgrGetNextEventItem()`

Note: Use a loop, if necessary, to retrieve all items.

5 Process the event

Figure 3-4 Accessing an Event from a Consumer Application

1. Include the Event Manager API header file so that you can access the Event Manager API functions:

   ```
   #include <emgrapi.h>
   ```

2. If the consumer is a shared library consumer, the function prototype must use the same format as the following `ConsumerEntry_t` prototype to enable the Event Manager to call it:

   ```
   typedef int ConsumerEntry_t(EmgrEvent_t *event,
   int argc,
   const char *argv[]);
   ```

   For example:

   ```
   int TstDso(EmgrEvent_t *event,
   int argc,
   char *argv[]);
   ```

3. If the consumer is a shared memory consumer, use the `emgrShmInitEvent()` or `emgrShmCliInitEvent()` function to initialize the event structure from shared memory.

4. Retrieve the data using one of the following methods:

   - If you know the name of the item:

     Use the `emgrGetEventItem()` function to get the value of the item. You must specify the name of the item as a parameter to the function.

   - If you do not know the name of the item:

     Use the `emgrGetFirstEventItem()` function to get a (name, value) pair. If there is more than one (name, value) pair, use the `emgrGetNextItem()` function in a loop to load all of the (name, value) pairs.

5. Process the event.

Example

The following example shows shared library consumer code that accesses all (name, value) pairs in an event.

Example 3-8 Example Code to Access Event Data from a Shared Library Consumer Application

```
<#include emgrapi.h>

int TstDso(EmgrEvent_t *event, int argc, char *argv[]) {
   int i;
   const char *name;
   const void *value;
   int type = 0, length = 0;

   printf("consumer_main\n");
   printf("   type=%d;class=%d;version=%d",
  event->header.evType,event->header.evClass,event->header.version);

   printf("   ORIGIN=%s;APPNAME=%s;SOURCE=%s\n",
  event->origin,event->appname,event->source);

   i = emgrGetFirstEventItem(event, &name, &value, &type, &length);
   while (i == 0) {
      printf("   %s=[%s];\n",name,(char *) value);
      i = emgrGetNextEventItem(event, &name, &value, &type, &length);
   }

   printf("  Number of args: %d\n",argc-1);
   for(i=0; i< argc; i++)
      printf("   arg[%d] = '%s'\n",i,argv[i]);

   return 0;
}
```

The following example code accesses event data from a shared memory consumer.

Example 3-9 Example Code to Access Event Data from a Shared Memory Consumer

```
int main(int argc, const char *argv[]) {

   int i = 0;

   for(i = 0; i < argc; i++) {
      printf("Arg[%d] = `%s'\n", i, argv[i]);
   }

   {
      int error = 0;
      EmgrEvent_t *pEvent = emgrShmCliInitEvent( argc, argv, &error);

      if ( pEvent != NULL ) {
 emgrPrintEvent(pEvent, stdout);

 emgrFreeEvent(pEvent);
      } else {
 fprintf(stderr,
 "Error %d initializing event from the shared memory\n",
 error);
      }

   }
   return 0;
}
```

eventmond Command-line Options

Use the `eventmond` command to configure the `eventmond` daemon or to send commands to tasks that `eventmond` is running.

Note: The task interface is complex and remains proprietary until SGI thoroughly tests it. When the task interface becomes available for general use, this document will be revised to provide more information about the task interface and how to write tasks that use it. SGI recommends that you do not attempt to create and load custom tasks at this time.

Configuring the Daemon

Use the following command-line options to configure how the `eventmond` daemon behaves:

`eventmond [-p` <*port_number*>`] [-s` <*socket_name*>`] [-c` <*subscription_file_name*>`]`
`[-B-]`

Table 4-1 eventmond Command-line Options to Configure the Daemon

Option	Description
`-p` <*port_number*>	Specifies the TCP/IP port that `eventmond` uses to send and receive event information (default: 5553)
`-s` <*socket_name*>	Specifies the UNIX domain socket that `eventmond` uses for the command execution interface (load task, run task, and so on) (default: `/tmp/s.eventmond`)
`-c` <*subscription_file_name*>	Specifies a file in which `eventmond` saves the current subscription data so consumers do not have to resubscribe events if the `eventmond` daemon is stopped and restarted
`-B-`	Specifies that `eventmond` should not run as a daemon

Sending Commands to Tasks

Use the following command-line options to send commands to tasks that the Event Manager is running:

```
eventmond [-L <taskname> -P <parameters>] [-U <taskname> -P <parameters>]
  [-S <taskname> -P <parameters>] [-Q <taskname> -P <parameters>]
  [-I <taskname> -P <parameters>] [-C <taskname> -P <parameters>]
  [-M <taskname> -P <parameters>] [-T [A|I]]
```

Table 4-2 eventmond Command-line Options to Start and Stop Tasks

Option[a]	Description[b]
`-C` *<taskname>* `-P` *<parameters>*	Sends a command to a running task
`-I` *<taskname>* `-P` *<parameters>*	Returns information about a running task
`-L` *<taskname>* `-P` *<parameters>*	Loads a task into the Event Manager and starts it
`-M` *<taskname>* `-P` *<parameters>*	Sends a message to a running task
`-Q` *<taskname>* `-P` *<parameters>*	Stops a running task
`-S` *<taskname>* `-P` *<parameters>*	Starts a loaded task
`-T [A|I]`	Lists the loaded tasks and shows the current status of each task Use `-T A` to show all active tasks. Use `-T I` to show all idle tasks. Use `-T` to show all tasks.
`-U` *<taskname>* `-P` *<parameters>*	Stops a running task and unloads it

a. *<taskname>* is either the name of the task (for example, `syslog`) or the full DSO name (for example, `libsyslog.so`). The full pathname (for example, `/usr/lib/syslog`) of the DSO is not required.

b. The commands are not applicable to all tasks. For example, the `syslog` task uses only the load task and unload task commands.

Displaying Help

Use the `-h` command-line option to display information about the command-line options that are available:

```
eventmond -h
```