# EMG plugin API

## Overview

The functionality in EMG can be extended by writing a plugin. These should be written in C, or in some language that can produce a shared library with "C" linkage. Each plugin implements one or more of the API functions, each one called at a specific place in the life cycle of each message.

All exported functions are available to all plugins that are loaded by the same EMG process, so use the static keyword on all functions that shouldn't be used by EMG. If that is not possible, for example if the plugin consists of several files, make sure to give them a unique prefix.

## API functions

There are eight API functions you can implement. If they exist in the plugin they will be called. Any missing functions will be ignored, so just implement the functions needed.

The functions are of three different types:

### Load and unload

When EMG starts, the shared library that contains the plugin is loaded and this API function is called:

```
void* create_config(char* name, char* filename);
```

The name is the name of the plugin, as given in the server.cfg file by the PLUGIN myplugin < line, and the filename is the value of the keyword CONFIG in that section. There is nothing that says that this has to be a filename, it might just as well be the name of a database profile to use, or something entirely different. Using it as the filename of the configuration file for the plugin is very convenient though. From EMG's point of view it is simply a string, so it is up to the plugin to interpret it.

The return value is passed as the config parameter to the other API functions.

When EMG is stopped (emgd –stop), or refreshed (emgd -reload or emgd -refresh), the plugin is unloaded. At this point, this API function is called:

```
void destroy_config(void* config);
```

The parameter is the pointer that create_config() returned. The function should release all resources that were acquired by the create_config() function.

Due to linker namespace problems there is nothing that ensures that create_config() finds the destroy_config() from the correct plugin file, so if you use more than one plugin you MUST implement destroy_config in all your plugins as a wrapper which in turn calls a static “destroy” function like in the example below.

```
void* create_config(char* name, char* filename)
{
    config_t* config = calloc(1, sizeof(config_t));
    // populate config
    if (isvalid(config)) return config;
    do_destroy_config(config);
    return NULL;
}
static void do_destroy_config(void* config)
{
```

```
    // clear config
  free(config);
}
void destroy_config(void* config)
{
   do_destroy_config(config);
}
```

### Thread start and stop

For each thread that is created to run the plugin, this function is called first:

```
void* plugin_thread_start(void* config);
```

The config is the configuration object returned by create_config(). This function should create an object that contains any thread specific data that is needed. If the plugin wants to communicate with a database, this is a good place to perform the connection, initialize any prepared statements, etc.

The returned pointer will be used as the thread_local parameter to plugin_thread_stop() below. The message functions will get the pointer in the request parameter in the field threadlocal.

If the plugin will only be used with a single instance, these operations can instead be done in create_config().

When the plugin thread is about to exit, this function is called:

```
void plugin_thread_stop(void* threadlocal);
```

The threadlocal pointer is the one returned by plugin_thread_start() above. Just as for the pair create_config() and destroy_config(), this function should
free all resources aquired by plugin_thread_start().

### For each message

There are four points in the life cycle of each message that check for an external plugin.

The first point is after the message has arrived to EMG. All DEFAULT and FORCE options are considered, all address rewrites are done, any contents mapping are performed, etc. The only thing left to do is the routing. At this point the function before_receive() is called. This function can set the ROUTE option to force the message to be routed somewhere, modify any other option, and even prohibit the message from being accepted. Normally the function should return 0, but if it returns anything else the message is rejected with the error code set from response->result + OFFSET (is set in the plugin configuration).

The second point is right before the message is being sent. At this point the function before_send() is called. This function can be used to temporarily stop messages on a connector based on some external criteria (e.g. the time of day), run the first phase in a "two phase commit" scenario, etc. The return code should be 0 to accept the operation and anything else to stop the message.

After the message has been sent and a return code has been received from the SMSC at the other end, the function after_send() is called. The SMSC return code is put in the result field in the request structure.

This function should always return 0.

Some messages get a delivery receipt back after some time. When this happens, the function after_dlr() is called. The message given to this function is the new message entry containing the delivery report which is going back to the original sender.

All four functions above use the same parameters:

```
int before_receive(void* config, pluginrequest_t* request, pluginresponse_t* response);
int before_send(void* config, pluginrequest_t* request, pluginresponse_t* response);
int after_send(void* config, pluginrequest_t* request, pluginresponse_t* response);
int after_dlr(void* config, pluginrequest_t* request, pluginresponse_t* response);
```

The pluginrequest_t and pluginresponse_t structures are defined in the file plugintype.h.

## Message attributes

Most plugins want to examine the attributes in the message in some way. The message itself can be found in the request parameter, in the field qe. Each message contains a list of options, including the message body, the sender and recipient, etc.

The list of functions that can be used to examine and modify these values can be found in the section "Messages" below.

## Threads

All requests to plugins are queued, and one or more threads are used to perform the real function calls. The number of threads to use is taken from the keyword INSTANCES in the PLUGIN section. If the plugin cannot be made multithread safe, just set INSTANCES to 1 and all calls will be serialized for that plugin. This is independent from the number of connectors that use the plugin, so even if there are 10 connectors that use the same plugin, only one call will ever be made at the same time.

To make a multithread safe plugin, things are actually quite simple.

The main config structure is common for all plugin threads for a certain plugin, so if any fields there needs to be updated you must use mutexes of some sort to prevent data corruption. Feel free to use the functions listed in the "Mutexes" section below.

The rest of the data is private to the thread. Just do not return the same value as the threadlocal pointer in several threads, and you will be safe. Naturally you must also make sure you handle any static variables correctly.

## Configuration

### Configuring the plugin loader

First you must describe the plugin to the server. This is done in the server.cfg file, as a section called PLUGIN. There should be one PLUGIN section for each plugin. The section looks like this:

```
PLUGIN pluginname <
LIBRARY=...
...
>
```

#### LIBRARY

The keyword LIBRARY should point to the file containing the plugin, compiled as a shared library. For safety, always use a full path here.

This parameter is mandatory.

#### INSTANCES

The keyword INSTANCES says how many threads will be used for running the plugin. All requests to the same plugin are queued, and then those runner threads pick the jobs in the

order they came in. The "config" parameter will be the same, so you'll have to use the proper locks if you want to update anything there.

This parameter is mandatory. If the value is 0, no threads are started, and the plugin can't be used.

#### CONFIG

The keyword CONFIG contains the value that will be sent as the second parameter to create_config(). The most common use is to let it be a filename, containing the configuration options.

This parameter is optional.

#### OFFSET

The value of the parameter OFFSET should be an integer. This value is added to the response->result field, making it possible to get them into a unique range.

This parameter is optional.

### Configuring the connectors

Next there must be a reference from the connector(s) that should use the plugin. This is done by adding a line PLUGIN=pluginname for that connector. Any number of connectors can use the same plugin.

There is a second form of this reference, looking like this: PLUGIN=pluginname:parameter. This makes all calls from this connector having the field ccarg in the pluginrequest_t struct be set to the string parameter. The type of this field is a vbuf_t, and is NULL if the string is absent or empty.

### Configuring the plugin

Most plugins tend to need some sort of configuration file, to avoid having to recompile the plugin every time something in the environment changes. Read this file in the create_config() function. To help with this, you can call the function config_read(), with the following definition:

```
typedef int config_readline_t(void* cfg, vbuf_t* line);
int config_read(char* configname, void* config, config_readline_t* linereader);
```

The file is opened, and each line is read. Comments starting with '#' are removed, and empty lines are ignored. The rest is sent to the linereader() callback in a vbuf_t. Afterwards the file is closed.

## Logging

There are two functions you can use for logging. The first one should be used in the start and stop functions. It is defined like this:

```
void xlog(int loglevel, const char* format, ...);
```

The loglevel is one of SMSLOG_CRITICAL, SMSLOG_ERR, SMSLOG_WARNING, SMSLOG_INFO, SMSLOG_DEBUG, and SMSLOG_DEBUG2. The format is a normal printf-style format, followed by its parameters. If the level is at least as high as the level set in server.cfg, the line is added to the end of the $EMGDIR/log/general file. The string must end with a newline.

The second function shall be used in the four message functions. It looks like this:

```
CILOG(connectorinfo_t* ci, const char* pluginname, int loglevel,
```

int loglevel, const char* format, ...);

The connectorinfo_t parameter should be taken from the request parameter, in the ci field. The pluginname should be the name of the plugin, being sent as the first parameter to create_config(). This makes it easy to keep the log entries for different plugins separate.

The valid values for the loglevel are the same as for xlog(). To get the log entry to the correct file, make a bitwise or between that value and SMSLOG_PLUGIN (e.g. SMSLOG_PLUGIN | SMSLOG_INFO). The entries are sent to the file $EMGDIR/log/plugin.<pluginname>.

The loglevel comparison is done with the log level for the connector that is using the plugin. This value can be found in the loglevel field in the connectorinfo_t structure. This field can be checked before performing some expensive calculation, to avoid having the plugin spend lots of time producing some large log data, when the loglevel is set so that these lines would never appear. It can look like this:

```
connectorinfo_t* ci = request->ci;
if (ci->loglevel >= SMSLOG_DEBUG) {
  // some expensive operation
    @@...
  // then log the result
  CILOG(ci, config->name, SMSLOG_DEBUG | SMSLOG_PLUGIN, "...\n");
}
```

## Building the plugin

The commands needed to build a shared library are slightly different on the available platforms. In each case, add any -Idirectory or -llibrary as needed, depending on what your plugin does.

### Linux

```
gcc -c -DEMG myplugin.c
gcc -shared -Wl,-soname,myplugin.so.0 -o myplugin.so.0.0 myplugin.o
```

### Solaris

```
gcc -c -DEMG myplugin.c
gcc -G -h myplugin -Bdynamic -o myplugin.so.0.0 myplugin.o
```

### HP-UX

```
cc -c -fpic -DEMG myplugin.c
gcc -fpic -shared -o myplugin.sl.0.0 myplugin.o
```

## Troubleshooting

Since EMG is heavily multithreaded, single stepping through it with gdb is virtually pointless. Instead, use the logging functions to examine what the plugin is doing. A good way to start is to put this line at the beginning and end of each function. By putting it at the critical places, adding extra parameters as needed, one gets a good idea of what is going on.

```
xlog(SMSLOG_DEBUG2, "s:%d\n", __FILE__, __FUNCTION__, __LINE__);
```

By running emgd within gdb you get immediate feedback if there is a NULL pointer reference somewhere. Let it run, and then use bt to get a stack trace of where the problem was found. Just don't try to set any breakpoints.

If possible, you can also run emgd within Valgrind (http://www.valgrind.org). It has a rather high performance penalty, but is invaluable for finding memory related bugs.

## Functions

There are several functions within emgd that you can use from your plugin. These references will be resolved when the plugin is loaded, so you don't have to link with some EMG library or anything.

### Messages (qe)

Each message consists of a set of key-value pairs, called an "option". There are three functions to get the value of such an option.

```
char* qe_option_get(void* qe, int option);
char* qe_option_get2(void* qe, int option, int* len);
int qe_option_getint(void* qe, int option, int defaultvalue);
```

The function qe_option_get() returns a pointer to the string representation of the value. The function qe_option_get2() does the same, but also sets len to the length of the value.

The function qe_option_getint() returns a value as an integer. This should be used for options such as "type of number", "message type", and a few others. The string version will always work, but this function is more efficient.

To change the value of an option, use one of the the following functions:

```
void qe_option_replace(void* qe, int option, char* value);
void qe_option_replace2(void* qe, int option, char* value, int len);
void qe_option_replaceint(void* qe, int option, int value);
```

The parameter len is the length of the value. Use this function if the length is available, or when inserting data containing null bytes.

The list of options is available in the file mgp.h. All messsage functions are declared in pluginqueue.h.

### Mutexes

To make sure only one thread is accessing or modifying some common data, each section that uses that data must use some sort of lock. In EMG there are a couple of functions that can help with this. All of them are defined in the mutex.h file.

```
mutex_t *mutex_create(void);
```

Create a mutex_t object.

```
void mutex_destroy(mutex_t *);
```

Destroy the mutex_t object. The pointer can be NULL.

```
int mutex_lock(mutex_t *);
```

Lock the mutex. Any other threads that will try to lock the same mutex will block until the mutex is unlocked. Locking a NULL mutex will always succeed, so make sure the pointer is valid.

```
void mutex_unlock(mutex_t *);
```

Unlock the mutex. If another thread is waiting for this mutex, it will continue running.

### Lists

There are two types of list API's in EMG. The first one is list_t, which has a function for locking the list to be multithread safe. The second is pbuf_t, which is a little more lightweight. In a plugin both types are used.

### list_t

For a plugin, a list_t is only used when waiting for the reply from a database operation. As such, there are only two functions you'll have to use.

```
list_t* list_create(int listtype, void* dummy);
```

This function creates a new list.
The listtype should be LISTTYPE_LL. Send NULL for the dummy arg.

```
list_destroy(list_t* list, void* dummy);
```

Release all memory used by this list. Send NULL for the dummy arg.

### pbuf_t

A pbuf_t is a list of pointers. The functions are defined in the file pbuf.h. Several functions exists for traversing the list, both for calling some function on all entries as well as extracting one or more entries where the function returns non-zero.

### String buffers (vbuf_t)

A vbuf_t is a classic string type, dynamically growing when needed. The functions are defined in the file vbuf.h.

### Databases

The database profiles defined in server.cfg are available from plugins. The API requires the following steps. First you need to get the queue.

```
void* queue = db_findbyqueue("dbprofile-name");
```

From there, you need to get the "dbh".

```
void* dbh = db_finddbh(queue);
```

Next you need to build the SQL statement. For this, use a vbuf_t.
There are a couple of functions you can use for this.

To add a table name, prepending the TABLE_PREFIX setting if needed:

```
void db_appendtablename(vbuf_t* sql, const char* tablename);
```

To append a string, quoting any difficult characters such as single and double quotes:

```
void db_appendquoted(void* dbh, vbuf_t* sql, const char* string, int len);
```

To run the SQL statement:

```
void db_sendrequest(void* queue, list_t* responseq,
const char* sql, int encoding);
```

To get the char* version from a vbuf_t, use vbuf_tostring().
To create the responseq, do this:

```
list_t* responseq = list_create(LISTTYPE_LL, NULL);
```

Useful values for encoding are MGP_CHARCODE_LATIN1 and MGP_CHARCODE_UTF8.

Wait for the response like this:

```
dbresponse_t* dbresponse = db_readresponse(responseq);
```

If the statement as a SELECT, the result is extracted like this:

```
if (dbresponse) {
  dbrow_t dbrow;
  while ((dbrow = dbresult_fetchrow(dbresponse->dbresult)) != NULL) {
    // Values as char* in dbrow[0], dbrow[1] etc
```

```
  }
}
```

Finally clear the response:

```
db_freeresponse(dbresponse);
list_destroy(responseq, NULL);
```