Creating New Modules

About

This page outlines some of the structure available to build a new FreeSWITCH™ module.

Before You Begin...

- Ensure that the functionality you're looking for has not already been implemented somewhere else.
- Collaborate with the original authors if there is something very close that just needs a minor change to accommodate what you need.
- Review the bundled modules in freeswitch/src/mod/applications (particularly mod_skel) to get an understanding of how modules interface with FreeSWITCH and function.

Building Modules

Within Git checkout

It is possible to build your modules at the same time as FreeSWITCH.

Creating Your Module

- Start with a fresh git clone
- Create your own module directory under freeswitch/src/mod/applications
- Copy the skeleton from freeswitch/src/mod/applications/mod_skel/mod_skel to mod_yourmodule
- Edit as appropriate
- Add your module to the list of modules to build in freeswitch/modules.conf

Customizing Include/Library Flags

The first time you perform a 'make' from freeswitch, a Makefile will be generated for your module

If you wish to use functionality from the libraries bundled with FreeSWITCH, you might add something like this to your module's Makefile:

```
LOCAL_CFLAGS=-I$(switch_srcdir)/libs/libteletone/src
LOCAL_LDFLAGS=$(switch_srcdir)/libs/libteletone/libteletone.la
```

Working with multiple source files

If you want to work with multiple C or C++ files, you might add something like this to your module's Makefile:

```
LOCAL_OBJS=file.o file2.o file3.o
local_depend: $(LOCAL_OBJS)
```

Please note that the file list should not contain the source file which has the same name as the module.

Using development headers (libfreeswitch-dev)

It is also possible to build your module using only the libfreeswitch-dev development header package from the Debian repository.

This means you do not need to recompile FreeSWITCH, only your module(s). This means a much faster build.

- Create a directory for storing your module
- Copy the skeleton from freeswitch/src/mod/applications/mod_skel/mod_skel.c to mod_yourmodule.c
- Edit as appropriate
- Create this Makefile (more complex ones may be required as your project demands it):

```
```
You can now create your own Debian package (more information to be added later - Steven Ayre)

Module Interface

A module is expected to define the following "hooks":

**SWITCH_MODULE_LOAD_FUNCTION**

Using this macro defines the load function for the module, in this function you should initialize any global structures, hook up any event or state handlers, etc. If you return anything other than SWITCH_STATUS_SUCCESS, the module will not continue to be loaded.

**SWITCH_MODULE_RUNTIME_FUNCTION**

This is the runtime loop of the module, from here you can listen on sockets, spawn new threads to handle requests, etc.

**SWITCH_MODULE_SHUTDOWN_FUNCTION**

This is where you do any cleanup for unloading or reloading the module, you should release state handlers, event reservations, etc. You should also synchronize with shutting down the runtime thread (usually using something like a shared 'running' variable that the shutdown function sets to some value that the runtime function notices, sets to a third value and then exits).

Iterative Development

The build system is set up to conveniently support rapidly iterative development of your module. Lets look at how you can make changes to your module source, build those changes, and then install and reload your library in a running instance of FreeSWITCH; all with 3 quick and easy commands.

In these examples the FS source will be in a directory called `freeswitch` under our home directory. The module we'll use in our examples will be `mod_even_t_zmq` which would be located at `~/freeswitch/src/mod/event_handlers/mod_event_zmq`.

Building Your Module

The build system automatically generates a target for each module which happens to be the module's full name.

You've just made a change to the source code and now want to build just the module and not the whole of the switch:

```
josh@overt ~/freeswitch $ make mod_event_zmq
```
Installing Your Module

Once the build is done there should be an updated library file located in the module source directory. The library could be manually copied to the FS installation directory, but there are a number of other ancillary tasks that need to be done such as setting permissions and updating the runtime linker. The build system will handle all of this with an autogenerated target; each module has a distinct install target which is it's name appended with `-install`.

You've now built the module library and are ready to install the new library to the FS installation directory (on nix platforms this does not require shutting down a running switch instance):

```
josh@overt ~/freeswitch $ sudo make mod_event_zmq-install
```

Reloading Your Module

FS has the capability to reload modules which will cause the newly installed library to get loaded. N.B. your module's Shutdown function must accomplish a complete reversion of any setup that happens in it's Load function for a reload to work properly.

You've now built the module, installed it and now want to see how the new changes work; the module can be reloaded using the FreeSWITCH console:

```
freeswitch@overt> reload mod_event_zmq
```

XML API

XML is used a lot, every module gets its configuration from XML objects and some (e.g. directory) expose functions for modules to query information. This is how you query users, domains, etc. to get specific information.

Those function will take a pointer to a switch_xml_t structure, you need to pass a preallocated pointer so that the functions can write the proper XML object in there.

Here are a few ways to open a configuration section.

**Note:** XML objects that come from a locate or open function MUST be freed. To do so, call the switch_xml_free function. (note it doesn't take a pointer as argument)

```c
void switch_xml_free(switch_xml_t xml)
```

**Domains**

```c
switch_status_t switch_xml_locate_domain (const char *domain_name, switch_event_t *params, switch_xml_t *root, switch_xml_t *domain)
```

**Users**

```c
switch_status_t switch_xml_locate_user (const char *key, const char *user_name, const char *domain_name, const char *ip, switch_xml_t *root, switch_xml_t *domain, switch_xml_t *user, switch_event_t *params)
```

**Module Configuration**

```c
switch_xml_t switch_xml_open_cfg (const char *file_path, switch_xml_t *node, switch_event_t *params)
```

- `file_path` the name of the config section e.g. modules.conf
- `node` a pointer to point to the node if it is found
- `params` optional URL formatted params to pass to external gateways

Your module configuration file should resemble the following:

```xml
<configuration name="mymodule.conf" description="MyModule configuration">
  ...
</configuration>
```

You can then use freeswitch's XML functions to access your configuration data. **Note:** Don't forget to call switch_xml_free(&node) after you are done!

Gathering data from XML objects

Now that you have your favorite configuration element opened, let's get some data out of it.

The switch_xml_child(switch_xml_t xml, const char *name) function will return you the first node with the name passed as argument. You can then use the switch_xml_attr function to get the value of an attribute.

For example, to parse the following structure

```xml
<configuration name="mymodule.conf" description="MyModule configuration">
  <settings>
    <param name="key" value="value"/>
  </settings>
</configuration>
```

```c
switch_xml_t xml = switch_xml_open_cfg("mymodule.conf");
switch_xml_t settings = switch_xml_child(xml, "settings");
switch_xml_t param = switch_xml_child(settings, "param");
printf("key: %s, value: %s\n", switch_xml_attr(param, "name"), switch_xml_attr(param, "value"));
```
You could do the following:

```c
switch_xml_t xml = NULL, x_lists = NULL, x_list = NULL, cfg = NULL;
if ((xml = switch_xml_open_cfg("mymodule.conf", &cfg, NULL)) {  
  if (x_lists = switch_xml_child(cfg, "settings")) {  
    for (x_list = switch_xml_child(x_lists, "param"); x_list; x_list = x_list->next) {  
      const char *name = switch_xml_attr(x_list, "name"); // This needs to be const  
      const char *value = switch_xml_attr(x_list, "value");  
      // Ignore empty/missing attributes  
      if (switch_strlen_zero(name)) {  
        continue;
      }  
      if (switch_strlen_zero(value)) {  
        continue;
      }  
      if (!strcmp(name, "myattribute")) {  
        // Do something with value
      } else if (!strcmp(name, "myotherattribute")) {  
        // ...
      } else {  
        switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Unknown attribute %s\n", name);
      }
    }
  } else {
    switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Failed to load MyModule's config!\n");
  }
}
```

Reacting

Once your module is loaded, you want to react to what's happening, here is a few way to do so:

**Subscribing to events**

Since we have an event based architecture, you can subscribe to various events to get notified whenever something happens in the switch. The event system is going to call a callback function every time an event occurs.

```c
switch_event_bind(const char *id, switch_event_types_t event, const char *subclass_name, switch_event_callback_t callback, void *user_data);
```

- id is an identifier token of the binder, use your module's name
- event is which event you would like to receive, see the list
- subclass_name is obviously the name of the subclass
- user_data is a pointer which is going to be passed back to you whenever the callback is called.

If you want to receive all events, you can use the SWITCH_EVENT_ALL event type and the SWITCH_EVENT_SUBCLASS_ANY subclass.

```c
if (switch_event_bind("mod_mymodule", SWITCH_EVENT_ALL, SWITCH_EVENT_SUBCLASS_ANY, mymodule_event_handler, NULL) != SWITCH_STATUS_SUCCESS) {
  switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "Cannot bind to event handler!\n");
}
```

**Handling events**

Now that you registered a callback, here's how it works. Your callback procedure must have the following prototype:

```c
void mymodule_event_handler(switch_event_t *event);
```

Let's take a look at switch_event_t
/!

`switch_event` {  
    /*! the event id (descriptor) */  
    switch_event_types_t event_id;  
    /*! the priority of the event */  
    switch_priority_t priority;  
    /*! the owner of the event */  
    char *owner;  
    /*! the subclass of the event */  
    char *subclass_name;  
    /*! the event headers */  
    switch_event_header_t *headers;  
    /*! the event headers tail pointer */  
    switch_event_header_t *last_header;  
    /*! the body of the event */  
    char *body;  
    /*! user data from the subclass provider */  
    void *bind_user_data;  
    /*! user data from the event sender */  
    void *event_user_data;  
    /*! unique key */  
    unsigned long key;  
    struct switch_event *next;
};

Now most of you will need to filter on the event type, and do something with the headers the event contains. Here is an example:

```c
void mymodule_event_handler(switch_event_t *event) {
    switch_assert(event); // Just a sanity check
    switch(event->event_id) {
        case SWITCH_EVENT_CHANNEL_CREATE:
            switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "A new channel is born, its
called \"%s\"\n", switch_event_get_header_nil(event, "channel-name"); // This function isn't case-sensitive
            break;
        case SWITCH_EVENT_CHANNEL_DESTROY:
            switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR, "A channel named \"%s\" has been
destroyed\n" switch_event_get_header_nil(event, "channel-name");
            break;
        }
}
```

Setting hooks

**Channel States**

A description and explanation of Channel States is on a separate page.

**State handlers**

You have the option of getting a callback whenever your channel changes state. Your function signature should be:

```c
switch_status_t my_callback(switch_core_session_t *);
```

The way you expose state handlers is by passing a `switch_state_handler_table_t` structure containing functions pointers to your handlers. Here is the structure:

```c
struct switch_state_handler_table {  
    switch_state_handler_t on_init;  
    switch_state_handler_t on_routing;  
    switch_state_handler_t on_execute;  
    switch_state_handler_t on_hangup;  
    switch_state_handler_t on_exchange_media;  
    switch_state_handler_t on_reset;  
    switch_state_handler_t on_park;  
    switch_state_handler_t on_soft_execute;  
    switch_state_handler_t on_consume_media;  
    switch_state_handler_t on_hibernate;  
    switch_state_handler_t on_reporting;  
    void *padding[10];
};
```
The ones you do not want can be safely set to NULL, all others are going to be called when the state changes.

There is a few way to register state handlers, all using this structure.

Globally

```c
int switch_core_add_state_handler(const switch_state_handler_table_t *state_handler);
void switch_core_remove_state_handler(const switch_state_handler_table_t *state_handler);
```

On a specific channel

```c
int switch_channel_add_state_handler(switch_channel_t *channel, const switch_state_handler_table_t *state_handler);
void switch_channel_clear_state_handler(switch_channel_t *channel, const switch_state_handler_table_t *state_handler);
```

You can also set them when calling `switch_ivr_originate()` (more later)

PS: Callbacks will occur on the session's thread itself.
Note: If you register state handlers and support module unloading, make sure you unbind them to avoid segfaults.

Making calls

### Analysis of a FreeSWITCH call

(talk about call states, session, channels, private data, etc)

### Making calls from within your module

Below is my current understanding. I'm not core developer, so all is guessed --Sathieu 11:14, 18 February 2010 (UTC)

When `originate your_module/dest vmain` is called from the console:

- your_io_routines.outgoing_channel is called
  - create a new session (with `switch_core_session_request()`)
  - parse `outbound_profile->destination_number` to get called number
  - qualify the `session`, attached `channel` and attached `private data`
  - The channel is currently in `CS_NEW`
  - Change channel state to `CS_INIT`
  - Make the phone ring (and call `switch_channel_mark_ring_ready()`)
- your_state_handler.on_init is called
  - Change channel state to `CS_ROUTING`
  - your_state_handler.on_routing is called
  - What to do here?
- The call is answered
  - `switch_channel_mark_answered(channel)`

### Session Objects

You can get the session object of a session if you know the uuid. Consider:

```c
switch_core_session_t *session;
if ((session = switch_core_session_locate(uuid_here))) {
  /* do stuff with session */

  switch_core_session_rwlock(session);
}
```

**NOTE:** `switch_core_session_locate()` will automatically lock the session. When you are done with the session object it is MANDATORY to call `switch_core_session_rwlock()` or bad things will happen.

### Memory Management

In addition to the normal ways of managing memory in C, FreeSWITCH provides an interface to APR memory pools. You can allocate from a memory pool using `switch_core_alloc()` or allocate from a session's memory pool using `switch_core_session_alloc()`. You cannot free memory from a pool, all the memory allocated from a pool is freed when the pool is destroyed.

Memory pools are also hierarchical, in FreeSWITCH™ all the pools are a child of the core's master pool. You can create your own child pools using `switch_core_new_memory_pool()` or you can use a session's memory pool (not recommended).

However, since you can't free memory from a pool without destroying the entire pool, there's still times when using traditional C memory management is a better plan.

### Useful Macros
FreeSWITCH™ provides a number of very helpful macros to aid development.

**SWITCH_DECLARE_GLOBAL_STRING_FUNC**

This macro lets you define a dynamic global string (which is actually static to your module) setting function. The macro will free the previous value (if any) and then strdup the new value into it. The first value is the name of the function to be defined and the second parameter is the char * pointer to store the pointer to the allocated string.

**switch_malloc**

Macro to attempt a regular malloc(), which will abort() the process (eg. exit with SIGABRT), printing an error message to stderr containing the filename and line number where the error occurred.

```c
#define switch_malloc(ptr, len) (void)( (!!(ptr = malloc(len))) || (fprintf(stderr,"ABORT! Malloc failure at: %s:%s", __FILE__, __LINE__),abort(), 0), ptr )
```

**switch_zmalloc**

As per `switch_malloc`, but uses calloc() so that the contents of the buffer are filled with byte value zero before use.

```c
#define switch_zmalloc(ptr, len) (void)( (!!(ptr = calloc(1, (len)))) || (fprintf(stderr,"ABORT! Malloc failure at: %s:%s", __FILE__, __LINE__),abort(), 0), ptr)
```

**switch_strdup**

Similarly to `switch_malloc`, the macro will abort() if strdup() returns a null pointer (since strdup() is internally little more than a malloc() and a strcpy()).

```c
#define switch_strdup(ptr, s) (void)( (!!(ptr = strdup(s))) || (fprintf(stderr,"ABORT! Malloc failure at: %s:%s", __FILE__, __LINE__),abort(), 0), ptr)```

**switch_safe_free**

Macro to avoid freeing a null pointer. Pointer is set to null after free()ing to avoid pointing to a stale buffer.

```c
#define switch_safe_free(it) if (it) {free(it);it=NULL;}
```

**New Module Checklist**

When proposing a new module:

- Add a Makefile.am for the module
- Add the module to the FS configure.ac
- Add the module to build/modules.conf.in (commented out)
- Describe the module in as much detail as possible in the comments at the top of the module
- Add our whitespace footer to the module files; ensure the '.[ch]' module files use tabs for indentation and are free of trailing whitespace (eg. in Emacs, run `M-x whitespace-mode`, then `M-x whitespace-cleanup`)
- Remove any unused code left over from debugging
- Ensure symbols not meant to be exported are declared static
- Don't add any files to conf/vanilla
- Write a commit message body describing the module, why it's useful, what it does, how it works, and any parts not yet implemented

**See Also:**

- Developer Potpourri