Table of Contents

Chapter 1. Module Interface

SIP Express Router features modular architecture which allows us to split SIP Express Router's functionality across several modules. This approach gives us greater flexibility, only required set of functions can be loaded upon startup which minimizes the server's memory footprint. Modules can be also provided by 3rd party developers and distributed separately from the main server. Most of the functionality that SIP Express Router provides is available through modules, the core itself contains only minimum set of functions that is essential for proper server's behaviour or that is needed by all modules.

This chapter provides detailed information on module interface of SIP Express Router, which is used to pass information on available functions and parameters from the modules to the core.

1.1. Shared Objects

First it would be good to know how SIP Express Router loads and uses modules before we describe the module interface in detail. This section gives a brief overview of SIP Express Router's module subsystem.

SIP Express Router modules are compiled as "shared objects". A file containing a shared object has usually .so suffix. All modules (shared objects) will be stored in one directory after installation. For example tm module, which contains code essential for stateful processing, will be stored in file named `tm.so`. By default these files are stored in `/usr/lib/ser/modules` directory.

You can later load the modules using **loadmodule** command in your configuration file. If you want to load previously mentioned `tm.so` module, you can do it using **loadmodule "/usr/lib/ser/modules/tm.so"** in your configuration file. This command invokes dynamic linker provided by the operating system which opens `tm.so` file, loads it into memory and resolves all symbol dependencies (a module might require symbols from the core, for example functions and variables).

As the last step of the module loading the core tries to find variable named `exports`, which describes all functions and parameters provided by the module. These functions and parameters are later available to the server and can be used either in the configuration file or by other modules.

1.2. Exporting Functions

Each module can provide zero or more functions, which can be used in the configuration file or by other modules internally. This section gives a detailed description of structure describing exported functions and passing this information to the core through the module interface.

Each function exported by a module must be described by cmd_export_t structure. Structures describing all exported functions are arranged into an array and pointer to the array is then passed to the core. The last element of the array must contain 0 in all it's fields, this element serves as the mark telling the core that this is the very last element and it must stop scanning the array.

Each exported function is described by the following structure:

```
struct cmd_export_ {
char* name;             /* null terminated command name */
cmd_function function;  /* pointer to the corresponding function */
int param_no;           /* number of parameters used by the function */
fixup_function fixup;   /* pointer to the function called to "fix" the parameters */
int flags;              /* Function flags */
};

typedef struct cmd_export_ cmd_export_t;
```

Meaning of the fileds:

- `char* name`

  This is the name under which the function will be visible to the core. Usually it is the same as the name of the corresponding function.

- `cmd_function function`

  cmd_function type is defined as follows:

  ```
  typedef int (*cmd_function)(struct sip_msg*, char*, char*);
  ```

  The first parameter is a SIP message being processed, the other 2 parameters are given from the configuration file.

  > **Note:** From time to time you might need to export a function that has different synopsis. This can happen if you export functions that are supposed to be called by other modules only and must not be called from the configuration script. In this case you will have to do type-casting otherwise the compiler will complain and will not compile your module.
  >
  > Simply put (cmd_function) just before the function name, for example `(cmd_function)my_function`. Don't use this unless you know what are you doing ! The server might crash if you pass wrong parameters to the function later !

- `int param_no`

  Number of parameters of the function. It can be 0, 1 or 2. The function will be not visible from the configuration script if you use another value.

- `fixup_function fixup`

  This is the function that will be used to "fixup" function parameters. Set this field to 0 if you don't need this.

  If you provide pointer to a fixup function in this field, the fixup function will be called for each occurence of the exported function in the configuration script.

  The fixup function can be used to perform some operation on the function parameters. For example, if one of the parameters is a regular expression, you can use the fixup to compile the regular expression. The fixup functions are called only once - upon the server startup and so the regular expression will be compiled before the server starts processing messages. When the server calls the exported function to process a SIP message, the function will be given the already compiled regular expression and doesn't have to compile it again. This is a significant performance improvement.

  Fixup functions can also be used to convert string to integer. As you have might noticed, the exported functions accept up to 2 parameters of type char*. Because of that it is not possible to pass integer parameters from the script files directly. If you want to pass an integer as a parameter, you must pass it as string (i.e. enclosed in quotes).

  Fixup function can be used to convert the string back to integer. Such a conversion should happend only once because the string parameter doesn't change when the server is running. Fixup is therefore ideal place for the conversion, it will be converted upon the server startup before the server starts processing SIP messages. After the conversion the function will get directly the converted value. See existing modules for example of such a fixup function.

- `int flags`

  Usage of each function can be restricted. You may want to write a function that can be used by other modules but cannot be called from the script. If you write a function that is supposed to process SIP requests only, you may want to restrict it so it will be never called for SIP replies and vice versa. That's what is flags field for.

  This field is OR value of different flags. Currently only REQUEST_ROUTE and REPLY_ROUTE flags are defined and used by the core. If you use REQUEST_ROUTE flag, then the function can be called from the main route block. If you use REPLY_ROUTE flag, then the function can be called from reply route blocks (More on this in the SER User's Guide). If this field is set to 0, then the function can be called internally (i.e. from other modules) only. If you want to make your function callable anywhere in the script, you can use REQUEST_ROUTE | REPLY_ROUTE.

1.3. Exporting Parameters

Each module can provide zero or more parameters, which can affect the module's behaviour. This section gives a detailed description of structures describing exported parameters and passing this information to the core through the module interface.

Each parameter exported by a module must be described by param_export_t structure. Structures describing all exported parameters are arranged into an array and pointer to the array is then passed to the core. The last element of the array must contain 0 in all it's fields, this element serves as the mark telling the core that this is the very last element and it must stop scanning the array (This is same as in array of exported functions).

Each exported parameter is described by the following structure:

```
struct param_export_ {
char* name;             /* null terminated param. name */
modparam_t type;        /* param. type */
void* param_pointer;    /* pointer to the param. memory location */
};

typedef struct param_export_ param_export_t;
```

Meaning of the fields:

- `char* name`

  This is null-terminated name of the parametes as it will be used in the scripts. Usually this is the same as the name of the variable holding the value.

- `modparam_t type`

  Type of the parameter. Currently only two types are defined. INT_PARAM for integer parameters (corresponding variable must be of type int) and STR_PARAM for string parameters (corresponding variable must be of type char*).

- `void* param_pointer`

  Pointer to the corresponding variable (stored as void* pointer, make sure that the variable has appropriate type depending on the type of the parameter !).

1.4. Module Initialization

If you need to initialize your module before the server starts processing SIP messages, you should provide initialization function. Each module can provide two initialization functions, main initialization function and child-specific initialization function. Fields holding pointers to both initialization functions are in main export structure (will be described later). Simply pass 0 instead of function pointer if you don't need one or both initialization functions.

The main initialization function will be called before any other function exported by the module. The function will be called only once, before the main process forks. This function is good for initialization that is common for all the children (processes). The function should return 0 if everything went OK and a negative error code otherwise. Server will abort if the function returns a negative value.

Per-child initialization function will be called *after* the main process forks. The function will be called for each child separately. The function should perform initialization that is specific for each child. For example each child process might open it's own database connection to avoid locking of a single connection shared by many processes. Such connections can be opened in the per-child initialization function. The function accepts one parameter which is rank (integer) of child for which the function is being executed. This allows developers to distinguish different children and perform different initialization for each child. The meaning of return value is same as in the main initialization function.

1.5. Module Clean-up

A module can also export a clean-up function that will be called by the main process when the server shuts down. The function accepts no parameters and return no value.

1.6. Module Callbacks

TBD.

1.7. exports Structure - Assembling the Pieces Together

We have already described how a module can export functions and parameters, but we haven't yet described how to pass this information to the core. Each module must have variable named `exports` which is structure module_exports. The variable will be looked up by the core immediately after it loads the module. The structure contains pointers to both arrays (functions, parameters), pointers to both initialization functions, destroy function and the callbacks. So the structure contains everything the core will need.

The structure looks like the follows:

```
struct module_exports{
    char* name;                     /* null terminated module name */
    cmd_export_t* cmds;             /* null terminated array of the exported commands */
    param_export_t* params;         /* null terminated array of the exported module parameters */
    init_function init_f;           /* Initilization function */
    response_function response_f;   /* function used for responses, returns yes or no; can be nul
    destroy_function destroy_f;     /* function called when the module should be "destroyed", e.g
    onbreak_function onbreak_f;
    child_init_function init_child_f;  /* function called by all processes after the fork */
};
```

Field description:

- `char* name`

Null terminated name of the module

- `cmd_exports* cmds`

  Pointer to the array of exported functions

- `param_export_t* params`

  Pointer to the array of exported parameters

- `init_function init_f`

  Pointer to the module initialization function

- `response_function response_f`

  Pointer to function processing responses

- `destroy_function destroy_f`

  Pointer to the module clean-up function

- `onbreak_function onbreak_f`

  TBD

- `child_init_function init_child_f`

  Pointer to the per-child initialization function

1.8. Example - Simple Module Interface

Let's suppose that we are going to write a simple module. The module will export two functions - `foo_req` which will be processing SIP requests and `foo_int` which is an internal function that can be called by other modules only. Both functions will take 2 parameters.

```
/* Prototypes */
int foo_req(struct sip_msg* msg, char* param1, char* param2);
int foo_res(struct sip_msg* msg, char* param1, char* param2);

static cmd_export cmds[] = {
    {"foo_req", foo_req, 2, 0, ROUTE_REQUEST},
```

```
    {"foo_int", foo_int, 2, 0, 0            },
    {0, 0, 0, 0}
};
```

The module will also have two parameters, foo_bar of type integer and bar_foo of type string.

```
int foo_bar = 0;
char* bar_foo = "default value";

static param_export params[] = {
    {"foo_bar", INT_PARAM, &foo_bar},
    {"bar_foo", STR_PARAM, bar_foo     },
    {0, 0, 0}
};
```

We will also create both initialization functions and a clean-up function:

```
static int mod_init(void)
{
    printf("foo module initializing\n");
}

static int child_init(int rank)
{
    printf("child nr. %d initializing\n", rank);
    return 0;
}

static void destroy(void)
{
    printf("foo module cleaning up\n");
}
```

And finally we put everything into the exports structure:

```
struct module_exports exports = {
    "foobar",   /* Module name */
    cmds,       /* Exported functions */
    params,     /* Exported parameters */
    mod_init,   /* Module initialization function */
    0,          /* Response function */
    destroy,    /* Clean-up function */
    0,          /* On Cancel function */
    child_init  /* Per-child init function */
};
```

And that’s it.