libmodule Documentation

Release 5.0.0

Federico Di Pierro

Sep 07, 2019

Contents:

CHAPTER 1

Concepts

1.1 Module

A module is core entity of libmodule: it is a single and indipendent logical unit that reacts to certain events, both pubsub and socket ones. It can be seen as an Actor with the power of managing socket events. It offers some callbacks that are used by libmodule to manage its life. It is initialized through MODULE macro:

```
MODULE("test")
```

This macro creates a "test" module. MODULE macro also creates a constructor and destructor that are automatically called by libmodule at start and at end of program. Finally, this macro declares all of needed callbacks and returns an opaque handler for the module, that will be transparently passed with each call to libmodule API while using *Easy API*.

1.2 Context

A context is a way to create subnets of modules. You can loop on events from each context, and each context behaves independently from others. This can be particularly useful when dealing with 2+ threads; ideally, each thread has its own module's context and thus its own events to be polled. A context is automatically created for you first time a module that binds on that context is registered; so, multi-context API is very similar to single context one. To initialize a module binding it to its context, use MODULE_CTX macro:

```
MODULE_CTX("test", "myCtx")
```

This macro firstly creates a "myCtx" context, then a "test" module using same MODULE macro as before. Indeed, MODULE macro is only a particular case of MODULE_CTX macro, where myCtx is automatically setted to "default". This makes sense, as you can expect: single context API is a multi context API with only 1 context. Modules can only see and reach (through PubSub messaging) other modules from same context.

1.3 Loop

Libmodule offers an internal loop, started with modules_ctx_loop(); note that each context has its own loop. Moreover, you can even easily integrate it into your own loop: modules_ctx_get_fd() will retrieve a pollable fd and POLLIN events will be raised whenever a new message is available. Remember that before starting your loop, modules_ctx_dispatch() should be called, to dispatch initial "LoopStarted" messages to each module. Then, whenever POLLIN data is available on libmodule's fd, you only need to call modules_ctx_dispatch() again. Finally, remember to close() libmodule's fd retrieved through modules_ctx_get_fd().

CHAPTER 2

Data Structures

2.1 Macros

```
#define MODULE_VERSION_MAJ @PROJECT_VERSION_MAJOR@
#define MODULE_VERSION_MIN @PROJECT_VERSION_MINOR@
#define MODULE_VERSION_PAT @PROJECT_VERSION_PATCH@

#define MODULE_DEFAULT_CTX  "default"
#define MODULE_MAX_EVENTS   64
```

2.2 Types

```
/* Incomplete structure declaration to self handler */
typedef struct _self self_t;

/* Modules states */
enum module_states { IDLE = 0x1, RUNNING = 0x2, PAUSED = 0x4, STOPPED = 0x8 };

/* PubSub message types */
enum msg_type { USER, LOOP_STARTED, LOOP_STOPPED, MODULE_STARTED, MODULE_STOPPED,␣
↪MODULE_POISONPILL };

typedef struct {
    const char *topic;
    const void *message;
    ssize_t size;
    const self_t *sender;
    enum msg_type type;
} ps_msg_t;

typedef struct {
```

(continues on next page)

(continued from previous page)

```
    int fd;
    const void *userptr;
} fd_msg_t;

typedef struct {
    bool is_pubsub;
    union {
        const fd_msg_t      *fd_msg;
        const ps_msg_t      *ps_msg;
    };
} msg_t;

/* Callbacks typedefs */
typedef void (*init_cb)(void);
typedef bool (*evaluate_cb)(void);
typedef void (*recv_cb)(const msg_t *const msg, const void *userdata);
typedef void (*destroy_cb)(void);

/* Logger callback */
typedef void (*log_cb)(const self_t *self, const char *fmt, va_list args, const void␣
↪*userdata);

/* Memory management user-passed functions */
typedef void *(*malloc_fn)(size_t size);
typedef void *(*realloc_fn)(void *ptr, size_t size);
typedef void *(*calloc_fn)(size_t nmemb, size_t size);
typedef void (*free_fn)(void *ptr);

/* Struct that holds user defined callbacks */
typedef struct {
    init_cb init;                           // module's init function (should return␣
↪a FD)
    evaluate_cb evaluate;                   // module's state changed function
    recv_cb recv;                           // module's recv function
    destroy_cb destroy;                     // module's destroy function
} userhook_t;

/* Struct that holds user defined memory functions */
typedef struct {
    malloc_fn _malloc;
    realloc_fn _realloc;
    calloc_fn _calloc;
    free_fn _free;
} memhook_t;
```

2.3 Return Codes

```
typedef enum {
    MOD_WRONG_PARAM = -8,
    MOD_NO_MEM,
    MOD_WRONG_STATE,
    MOD_NO_PARENT,
    MOD_NO_CTX,
    MOD_NO_MOD,
```

(continues on next page)

(continued from previous page)

```
    MOD_NO_SELF,
    MOD_ERR,
    MOD_OK
} module_ret_code;
```

CHAPTER 3

Callbacks

Every module needs 5 functions that must be defined by developer. If using *Easy API*, they are automatically declared by MODULE macro. Moreover, a module_pre_start function is declared too, but it is not needed by libmodule interface, ie: it can be left undefined. Your compiler may warn you about that though.

```
static void module_pre_start(void);
static void init(void);
static bool check(void);
static bool evaluate(void);
static void receive(const msg_t *const msg, const void *userdata);
static void destroy(void);
```

module_pre_start (void)
: This function will be called before any module is registered. It is the per-module version of *modules_pre_start* function.

init (void)
: Initializes module state; useful to register any fd to be polled or to register any topic. Note that init() is only called first time module is started.

check (void)
: Startup filter to check whether this module should be actually created and managed by libmodule.

  Returns true if the module should be registered, false otherwise.

evaluate (void)
: Similar to check() function but at runtime: this function is called for each IDLE module after evey state machine update and it should check whether a module is now ready to be start (ie: init should be called on this module and its state should be set to RUNNING). Use this to check intra-modules dependencies or any other env variable.

  Returns true if module is now ready to be started, else false.

receive (msg, userdata)
: Poll callback, called when any event is ready on module's fd or when a PubSub message is received by a module. Use msg->is_pubsub to decide which internal message should be read (ie: ps_msg_t or fd_msg_t).

  Param `const msg_t * const` msg: pointer to msg_t struct.

Param `const void *` userdata: pointer to userdata as set by m_set_userdata.

destroy(void)

Destroys module, called automatically at module deregistration. Please note that module's fds set as autoclose will be closed.

CHAPTER 4

Lifecycle

4.1 Easy API

We previously saw that every module has to expose at least 5 functions. We will now go deeper, better understanding a module's lifecycle. First of all, module lifecycle is automatically managed by libmodule; moreover, when using *Easy API*, module registration/deregistration is completely automated and transparent to developer. This means that when using easy API (or multicontext API), you will only have to declare a source file as a module and define needed functions.

Before any module is registered into libmodule, each module's module_pre_start() function is called. This function can be useful to set each module pre-starting state: this may be needed if any check() function depends on some global state (eg: as read by config file).

After every module_pre_start() function is called, libmodule will start checking which module needs to be registered, eventually registering them. For every module, check() function will be called; if and only if that function returns true, the module will be registered. An unregistered module is just dead code; none of its functions will ever be called. Once a module is registered, it will be initially set to an IDLE state. Idle means that the module is not started yet, thus it cannot receive any PubSub msg nor any event from fds.

As soon as its context starts looping, a module's evaluate() function will be called, trying to start it right away. Evaluate() will be then called at each state machine change, for each idle module.

As soon as module's evaluate() returns true, the module is started. It means its init() function is finally called and its state is set to RUNNING. A single module can poll on multiple fds: just call module_register_fd() multiple times. When a module reaches RUNNING state, modules_loop()/modules_ctx_loop() functions will finally receive events from its fds.

Whenever an event triggers on a module's fd, or the module receives a PubSub message from another one, its receive() callback is called. Receive callback will receive userdata too as parameter, as set by module_set_userdata().

Finally, when leaving program, each module's destroy() function is automatically called during the process of automatic module's deregistration.

4.2 Complex API

When dealing with libmodule's *Complex API*, no modules is automatically started for you, ie: you must manually call module_register()/module_deregister() on each module. When using complex API, you are responsible to register/deregister modules, and thus initing/destroying them. Note that with Complex API, module_pre_start() function is not available (it would be useless), and you won't need to define check() function. You will still have to define evaluate(), init(), receive() and destroy() functions (but you can freely name them!).

Everything else but module's (de)registration is same as Easy API.

4.3 Module States

As previously mentioned, a registered module, before being started, is in IDLE state. IDLE state means that it has still no source of events; it won't receive any PubSub message and even if it registers any fd, they won't be polled. When module is started, thus reaching RUNNING state, all its registered fds will start being polled; moreover, it can finally receive PubSub messages. Fds registered while in RUNNING state, are automatically polled. If a module is PAUSED, it will stop polling on its fds and PubSub messages, but PubSub messages will still be queued on its write end of pipe. Thus, as soon as module is resumed, all PubSub messages received during PAUSED state will trigger receive() callback. If a module gets STOPPED, it will stop polling on its fds and PubSub messages, and every autoclose fd will be closed. Moreover, all its registered fds are freed and its enqueued pubsub messages are destroyed. STOPPED state is the same as IDLE state, but it means that module was started at least once and module's init() callback has already been called. If you instead wish to stop a module letting it receive any enqueued pubsub message, you need to send a POISONPILL message to it, through module_poisonpill() function.

module_start() needs to be called on a IDLE or STOPPED module. module_pause() needs to be called on a RUNNING module. module_resume() needs to be called on a PAUSED module. module_stop() needs to be called on a RUNNING or PAUSED module.

CHAPTER 5

Memory

Libmodule, even in its _easy (high level) APIs, is targeting a smart and hopefully large audience. Consequently, it is up to developers to understand and observe any pointers' scope. It means that any data passed to its API is trusted; it's up to implementer to actually be sure that eg: a registered topic pointer stays alive until it is needed. This is often accomplished by using global/static variables or heap-allocations.

Libmodule chose to minimize internal heap allocations to avoid performance issues or enforcing a behaviour to developers, that in my opinion is much worse. Note that instead map and stack APIs are more flexible about it, offering to eg: duplicate hashmap keys.

CHAPTER 6

PubSub

6.1 Concepts

Those unfamiliar with actor like messaging, may wonder what a pubsub messaging is. PubSub (Publisher-Subscriber) messaging is much like a producer/consumer architecture: some entities publish messages on a topic; other entities can subscribe to the topic to receive those messages.

6.2 Implementation

Since libmodule 2.1, pubsub implementation is async and makes use of unix pipes. When sending a message to other modules, a pubsub message is allocated and its address is written in recipient module's writable end of pipe. The message will then get caught by modules_loop, the address read from readable end of pipe and callback called with the message.

Since libmodule 4.0.0, module_tell() makes use of module references: it means recipient should be ref'd through module_ref(). Note that you cannot call any function on a module's reference as you cannot impersonate another module. Only module_is(), module_get_name/context() and module_poisonpill() functions can be called through a module's reference.

6.3 System messages

Beside USER messages (ps_msg_t.type), there are some system messages, respectively: LOOP_STARTED, LOOP_STOPPED, MODULE_STARTED, MODULE_STOPPED. These pubsub messages are automatically sent by libmodule when matching functions are called, eg: * LOOP_STARTED(STOPPED) is sent whenever a loop starts(stops) looping. It is useful to actually start(stop) your pubsub messagging (eg: one module waits on LOOP_STARTED to send a pubsub message to another module, and so on...). It won't have any valued fields, except for type. * MODULE_STARTED(STOPPED) is sent whenever a module starts/resumes(stops/pauses). Again this is useful to inform other modules that a new module is available. It will have valued type and sender fields; sender will be set to started(stopped) module; it will be NULL though when a module is stopped by deregistering it.

Finally, note that system messages with valued sender won't be sent to modules that actually generated the message.

6.4 Topics

Topics are just strings. Thus, if subscribing to "foo" topic, a module will receive: broadcasted messages, messages told directly to it and messages published on "foo" topic. Note that you can subscribe to regex patterns too, eg: "f[o+]" would receive messages published on "fo", "foo", "fooo" etc etc. When you unsubscribe from a regex topic, you should pass same regex to which you subscribed.

6.5 Notes

Note that a context must be looping to receive any pubsub message. Moreover, when a context stops looping, all pubsub messages will be flushed and thus delivered to each RUNNING module. Pubsub message sent while context is not looping or module is PAUSED are buffered until context starts looping/module gets resumed. For more information, see pipe capacity. Finally, please be aware that data pointer sent through pubsub messaging is trusted, ie: you should pay attention to its scope.

CHAPTER 7

Module API

Module API denotes libmodule interface functions to manage each module lifecycle. It is splitted in two APIs.

7.1 Easy API

Module easy API consists of a single-context-multi-modules set of macros. It can be found in <module/module_easy.h>. These macros make it easy and transparent to developer all of the module's internal mechanisms, providing a very simple way to use libmodule. It enforces correct modularity too: each module must have its own source file. Where not specified, these functions return a *module_ret_code*.

MODULE (name)

Creates "name" module with a default context: declares all needed functions and creates both constructor and destructor that will automatically register/deregister this module at startup. Finally, it declares a `const self_t *_self` global variable that will be automatically used in every function call. Note that default context will be automatically created if this is the first module to bind on it.

Parameters

- **name** (`const char *`) – name of the module to be created

Returns void

MODULE_CTX (name, ctxName)

Creates "name" module in ctxName context: declares all needed functions and creates both constructor and destructor that will automatically register/deregister this module at startup. Finally, it declares a `const self_t *_self` global variable that will be automatically used in every function call. Note that ctxName context will be automatically created if this is the first module to bind on it.

Parameters

- **name** (`const char *`) – name of the module to be created
- **ctxName** (`const char *`) – name of the context in which the module has to be created

Returns void

self()
Returns _self handler for the module.

Returns `const self_t *`

m_load(path)
Attaches a new module from a .so file to "default" context. If module.so has a different context, this will be an error.

Parameters

- **path** (`const char *`) – shared object path.

m_unload(path)
Detaches a module loaded from a .so file.

Parameters

- **path** (`const char *`) – shared object path.

m_is(state)
Check current module's state

Parameters

- **state** (`const enum module_states`) – state we are interested in; note that it can be an OR of states (eg: IDLE | RUNNING)

Returns false if module' state is not 'state', true if it is and MOD_ERR on error.

m_start(void)
Start module's polling

m_pause(void)
Pause module's polling

m_resume(void)
Resume module's polling

m_stop(void)
Stop module's polling by closing its fds. Note that module is not destroyed: you can add new fds and call m_start on it. Moreover, its enqueued pubsub messages are destroyed.

m_become(new_recv)
Change receive callback to receive_$(new_recv)

Parameters

- **new_recv** (`const recv_cb`) – new module's receive callback; the function has prefix receive_ concatenated with new_recv

m_unbecome(void)
Reset to default receive poll callback

m_set_userdata(userdata)
Set userdata for this module; userdata will be passed as parameter to receive callback

Parameters

- **userdata** (`const void *`) – module's new userdata.

m_register_fd(fd, autoclose, userptr)
Registers a new fd to be polled by a module

Parameters

- **fd** (const int) – fd to be registered.
- **autoclose** (const bool) – whether to automatically close the fd on module stop/fd deregistering.
- **userptr** (const void *) – data to be passed in receive() callback msg->fd_msg_t when an event happens on this fd.

m_deregister_fd(fd)
Deregisters a fd from a module

Parameters

- **fd** (const int) – module's old fd.

m_log(fmt, args)
Logger function for this module. Call it the same way you'd call printf.

Parameters

- **fmt** (const char *) – log's format.
- **args** (variadic) – variadic argument.

m_dump()
Dump current module's state. Diagnostic API.

m_ref(name, modref)
Takes a reference from another module; it can be used in pubsub messaging to tell a message to it. It must not be freed.

Parameters

- **name** (const char *) – name of a module.
- **modref** (const self_t **) – variable that holds reference to module

m_subscribe(topic)
Subscribes the module to a topic. If module is already subscribed to topic, MODULE_ERR will be returned. Note that a regex is a valid topic too.

Parameters

- **topic** (const char *) – topic to which subscribe.

m_unsubscribe(topic)
Unsubscribes the module from a topic. If module is not subscribed to topic, MODULE_ERR will be returned.

Parameters

- **topic** (const char *) – topic to which unsubscribe.

m_tell(recipient, msg, size, autofree)
Tell a message to another module.

Parameters

- **recipient** (const self_t *) – module to whom deliver the message.
- **msg** (const void *) – actual data to be sent.
- **size** (const ssize_t) – size of data to be sent.
- **autofree** (const bool) – whether to autofree msg after last recipient's received it.

m_publish(topic, msg, size, autofree)
Publish a message on a topic.

Parameters

- **topic** (const char *) – topic on which publish message. Note that only topic creator can publish message on topic.
- **msg** (const void *) – actual data to be published.
- **size** (const ssize_t) – size of data to be published.
- **autofree** (const bool) – whether to autofree msg after last recipient's received it.

m_broadcast (msg, size, autofree, global)
Broadcast a message.

Parameters

- **msg** (const void *) – data to be delivered to all modules in a context.
- **size** (const ssize_t) – size of data to be delivered.
- **autofree** (const bool) – whether to autofree msg after last recipient's received it.
- **global** (const bool) – whether to broadcast to every context.

m_tell_str (recipient, msg)
Tell a string message to another module. Size is automatically computed through strlen, and autofree is set to false.

Parameters

- **recipient** (const self_t *) – module to whom deliver the message.
- **msg** (const char *) – message to be sent.

m_publish_str (topic, msg)
Publish a string message on a topic. Size is automatically computed through strlen, and autofree is set to false.

Parameters

- **topic** (const char *) – topic on which publish message. Note that only topic creator can publish message on topic.
- **msg** (const char *) – message to be published.

m_broadcast_str (msg, global)
Broadcast a string message. Same as calling m_publish(NULL, msg). Size is automatically computed through strlen, and autofree is set to false.

Parameters

- **msg** (const char *) – message to be delivered to all modules in a context.
- **global** (const bool) – whether to broadcast to every context.

m_poisonpill (recipient)
Enqueue a POISONPILL message to recipient. This allows to stop another module after it flushes its pubsub messages.

Parameters

- **recipient** (const self_t *) – RUNNING module to be stopped.

7.2 Complex API

Complex (probably better to say less-easy) API consists of **'Module easy API'_** internally used functions. It can be found in <module/module.h> header. Sometime you may avoid using easy API; eg: if you wish to use same source file for different modules, or if you wish to manually register a module. Again, where not specified, these functions return a *module_ret_code*.

module_register (name, ctx_name, *self*, hook)
Register a new module

Parameters

- **name** (`const char *`) – module's name.
- **ctx_name** (`const char *`) – module's context name. A new context will be created if it cannot be found.
- **self** (`self_t **`) – handler for this module that will be created by this call.
- **hook** (`const userhook *`) – struct that holds this module's callbacks.

module_deregister (*self*)
Deregister module

Parameters

- **self** (`self_t **`) – pointer to module's handler. It is set to NULL after this call.

module_load (path, ctx_name)
Attaches a new module from a .so file to ctx_name context. If module.so has a different context, this will be an error.

Parameters

- **path** (`const char *`) – shared object path.
- **ctx_name** (`const char *`) – module's context name.

module_unload (path)
Detaches a module loaded from a .so file.

Parameters

- **path** (`const char *`) – shared object path.

module_is (*self*, state)
Check current module's state

Parameters

- **self** (`const self_t *`) – pointer to module's handler.
- **state** (`const enum module_states`) – state we are interested in; note that it can be an OR of states (eg: IDLE | RUNNING)

Returns false if module'state is not 'state', true if it is and MOD_ERR on error.

module_start (*self*)
Start module's polling

Parameters

- **self** (`const self_t *`) – pointer to module's handler

module_pause (*self*)
Pause module's polling

Parameters

- **self** (const self_t *) – pointer to module's handler

module_resume (*self*)
Resume module's polling

Parameters

- **self** (const self_t *) – pointer to module's handler

module_stop (*self*)
Stop module's polling by closing its fds. Note that module is not destroyed: you can add new fds and call module_start on it. Moreover, its enqueued pubsub messages are destroyed.

Parameters

- **self** (const self_t *) – pointer to module's handler

module_become (*self*, new_receive)
Change receive callback to new_receive

Parameters

- **self** (const self_t *) – pointer to module's handler
- **new_receive** (const recv_cb) – new module's receive.

module_set_userdata (*self*, userdata)
Set userdata for this module; userdata will be passed as parameter to receive callback.

Parameters

- **self** (const self_t *) – pointer to module's handler
- **userdata** (const void *) – module's new userdata.

module_register_fd (*self*, fd, autoclose, userptr)
Register a new fd to be polled by a module

Parameters

- **self** (const self_t *) – pointer to module's handler
- **fd** (const int) – fd to be registered.
- **autoclose** (const bool) – whether to automatically close the fd on module stop/fd deregistering.
- **userptr** (const void *) – data to be passed in receive() callback msg->fd_msg_t when an event happens on this fd.

module_deregister_fd (*self*, fd)
Deregister a fd from a module

Parameters

- **self** (const self_t *) – pointer to module's handler
- **fd** (const int) – module's old fd.

module_get_name (*self*, name)
Get module's name from his self pointer

Parameters

- **self** (const self_t *) – pointer to module's handler

- **name** (`char **`) – pointer to storage for module's name. Note that this must be freed by user.

module_get_context (*self*, ctx)
Get module's name from his self pointer.

Parameters

- **self** (`const self_t *`) – pointer to module's handler
- **ctx** (`char **`) – pointer to storage for module's ctx. Note that this must be freed by user.

module_log (*self*, fmt, args)
Logger function for this module. Call it the same way you'd call printf

Parameters

- **self** (`const self_t *`) – pointer to module's handler
- **fmt** (`const char *`) – log's format.
- **args** (`variadic`) – variadic argument.

module_dump (*self*)
Dump current module's state. Diagnostic API.

Parameters

- **self** (`const self_t *`) – pointer to module's handler

module_ref (*self*, name, modref)
Takes a reference from another module; it can be used in pubsub messaging to tell a message to it. It must not be freed.

Parameters

- **self** (`const self_t *`) – pointer to module's handler
- **name** (`const char *`) – name of a module.
- **modref** (`const self_t **`) – variable that holds reference to module

module_subscribe (*self*, topic)
Subscribes the module to a topic. If module is already subscribed to topic, MODULE_ERR will be returned. Note that a regex is a valid topic too.

Parameters

- **self** (`const self_t *`) – pointer to module's handler
- **topic** (`const char *`) – topic to which subscribe.

module_unsubscribe (*self*, topic)
Unsubscribes the module from a topic. If module is not subscribed to topic, MODULE_ERR will be returned.

Parameters

- **self** (`const self_t *`) – pointer to module's handler
- **topic** (`const char *`) – topic to which unsubscribe.

module_tell (*self*, recipient, msg, size, autofree)
Tell a message to another module.

Parameters

- **self** (`const self_t *`) – pointer to module's handler

- **recipient** (`const self_t *`) – module to whom deliver the message.
- **msg** (`const void *`) – actual data to be sent.
- **size** (`const ssize_t`) – size of data to be sent.
- **autofree** (`const bool`) – whether to autofree msg after last recipient's received it.

module_publish (*self*, topic, msg, size, autofree)
Publish a message on a topic.

Parameters

- **self** (`const self_t *`) – pointer to module's handler
- **topic** (`const char *`) – topic on which publish message. Note that only topic creator can publish message on topic.
- **msg** (`const void *`) – actual data to be published.
- **size** (`const ssize_t`) – size of data to be published.
- **autofree** (`const bool`) – whether to autofree msg after last recipient's received it.

module_broadcast (*self*, msg, size, autofree, global)
Broadcast a message.

Parameters

- **self** (`const self_t *`) – pointer to module's handler
- **msg** (`const void *`) – data to be delivered to all modules in a context.
- **size** (`const ssize_t`) – size of data to be delivered.
- **autofree** (`const bool`) – whether to autofree msg after last recipient's received it.
- **global** (`const bool`) – whether to broadcast to every context.

module_poisonpill (*self*, recipient)
Enqueue a POISONPILL message to recipient. This allows to stop another module after it flushes its pubsub messages.

Parameters

- **self** (`const self_t *`) – pointer to module's handler
- **recipient** (`const self_t *`) – RUNNING module to be stopped.

CHAPTER 8

Modules API

Modules API denotes libmodule interface functions to manage context loop. Like Module API, it has an easy, single-context API. Moreover, it has an equivalent multi-context API. All these functions but modules_pre_start() return a *module_ret_code*.

Moreover, there are a bunch of functions that are common to every context (thus not needing ctx_name param).

modules_pre_start (void)
: This function will be called by libmodule before doing anything else. It can be useful to set some global state/read config that are needed to decide whether to start a module. You only need to define this function and it will be automatically called by libmodule.

main (argc, argv)
: Libmodule's provide its own main implementation. This means that, on simplest cases, you'll only have to write your modules and compile linking to libmodule. Libmodule's main() is quite simple: it just runs modules_ctx_loop() on every registered context (in different pthreads for multiple contexts). Note that this is a weak symbol, thus it is obviously overridable by users.

  Parameters

  - **argc** (`int`) – number of cmdline arguments.
  - **argv** (`char *[]`) – cmdline arguments.

modules_set_memhook (hook)
: Set memory management functions. By default: malloc, realloc, calloc and free are used.

  Parameters

  - **hook** (`const memhook_t *`) – new memory management hook.

8.1 Easy API

Modules easy API should be used in conjunction with *Easy API*. It abstracts all of libmodule internals mechanisms to provide an easy-to-use and simple API. It can be found in <module/modules_easy.h> header.

modules_set_logger(logger)
Set a logger. By default, module's log prints to stdout.

Parameters

- **logger** (const log_cb) – logger function.

modules_loop(void)
Start looping on events from modules. Note that as soon as modules_loop is called, a message with type == LOOP_STARTED will be broadcasted to all context's modules. Moreover, loop is automatically stopped (with return code 0) if there are no more RUNNING modules in its context.

modules_quit(quit_code)
Leave libmodule's events loop. Note that as soon as it is called, a message with type == LOOP_STOPPED will be broadcasted to all context's modules.

Parameters

- **quit_code** (const uint8_t) – exit code that should be returned by modules_loop.

modules_get_fd(fd)
Retrieve internal libmodule's events loop fd. Useful to integrate libmodule's loop inside client's own loop.

Parameters

- **fd** (int *) – pointer in which to store libmodule's fd

modules_dispatch(ret)
Dispatch libmodule's messages. Useful when libmodule's loop is integrated inside an external loop. This is a non-blocking function (ie: if no data is available to be dispatched, it will return).

Parameters

- **ret** (int *) – ret >= 0 and MOD_OK returned -> number of dispatched messages. ret >= 0 and MOD_ERR returned -> loop has been quitted by a modules_quit() code, thus it returns quit_code. Ret < 0 and MOD_ERR returned: an error happened.

8.2 Multi-context API

Modules multi-context API let you manage your contexts in a very simple way. It is exposed by <module/modules.h> header. It exposes very similar functions to single-context API (again, single-context is only a particular case of multi-context), that now take a "context_name" parameter.

modules_ctx_set_logger(ctx_name, logger)
Set a logger for a context. By default, module's log prints to stdout.

Parameters

- **ctx_name** (const char *) – context name.
- **logger** (const log_cb) – logger function.

modules_ctx_loop(ctx_name)
Start looping on events from modules. Note that this is just a macro that calls modules_ctx_loop_events with MODULE_MAX_EVENTS (64) events. Moreover, loop is automatically stopped (with return code 0) if there are no more RUNNING modules in its context.

Parameters

- **ctx_name** (const char *) – context name.

modules_ctx_loop_events(ctx_name, maxevents)

Start looping on events from modules, on at most maxevents events at the same time. Note that as soon as modules_loop is called, a message with type == LOOP_STARTED will be broadcasted to all context's modules. Moreover, loop is automatically stopped (with return code 0) if there are no more RUNNING modules in its context.

Parameters

- **ctx_name** (`const char *`) – context name.
- **maxevents** (`const int`) – max number of fds wakeup that will be managed at the same time.

modules_ctx_quit(ctx_name, quit_code)

Leave libmodule's events loop. Note that as soon as it is called, a message with type == LOOP_STOPPED will be broadcasted to all context's modules.

Parameters

- **ctx_name** (`const char *`) – context name.
- **quit_code** (`const uint8_t`) – exit code that should be returned by modules_loop.

modules_ctx_get_fd(ctx_name, fd)

Retrieve internal libmodule's events loop fd. Useful to integrate libmodule's loop inside client's own loop.

Parameters

- **ctx_name** (`const char *`) – context name.
- **fd** (`int *`) – pointer in which to store libmodule's fd

modules_ctx_dispatch(ctx_name, ret)

Dispatch libmodule's messages. Useful when libmodule's loop is integrated inside an external loop. This is a non-blocking function (ie: if no data is available to be dispatched, it will return).

Parameters

- **ctx_name** (`const char *`) – context name.
- **ret** (`int *`) – ret >= 0 and MOD_OK returned -> number of dispatched messages. ret >= 0 and MOD_ERR returned -> loop has been quitted by a modules_quit() code, thus it returns quit_code. Ret < 0 and MOD_ERR returned: an error happened.

CHAPTER 9

Map API

Libmodule offers an easy to use hashmap implementation, provided by <module/map.h> header. It is used internally to store context's modules and modules' subscriptions/topics.

9.1 Structures

```
typedef enum {
    MAP_EPERM = -6,
    MAP_WRONG_PARAM,
    MAP_ERR,
    MAP_MISSING,
    MAP_FULL,
    MAP_OMEM,
    MAP_OK
} map_ret_code;

/* Callback for map_iterate */
typedef map_ret_code (*map_cb)(void *, const char *, void *);

/* Fn for map_set_dtor */
typedef void (*map_dtor)(void *);

/* Incomplete struct declaration for hashmap */
typedef struct _map map_t;

/* Incomplete struct declaration for hashmap iterator */
typedef struct _map_itr map_itr_t;
```

9.2 API

Where not specified, these functions return a map_ret_code.

map_new(const bool *keysdup*, const map_dtor *fn*)
Create a new map_t object.

Parameters

- **keysdup** (const bool) – whether keys lifetime should be managed by map
- **fn** (const map_dtor) – callback called on value destroy. If NULL, values won't be automatically destroyed.

Returns pointer to newly allocated map_t.

map_itr_new(m)
Create a new map iterator object. Note that it can be freed with free().

Parameters

- **m** (const map_t *) – pointer to map_t

Returns pointer to newly allocated map_itr_t.

map_itr_next(itr)
Get next iterator. If next iterator is past last element, iterator will be automatically freed.

Parameters

- **itr** (map_itr_t *) – pointer to map_itr_t

Returns pointer to next iterator.

map_itr_remove(itr)
Remove current iterator {key, value} from map.

Parameters

- **itr** (map_itr_t *) – pointer to map_itr_t

map_itr_get_key(itr)
Get current iterator's key.

Parameters

- **itr** (const map_itr_t *) – pointer to map_itr_t

Returns current iterator's key

map_itr_get_data(itr)
Get current iterator's data.

Parameters

- **itr** (const map_itr_t *) – pointer to map_itr_t

Returns current iterator's data

map_itr_set_data(itr)
Set current iterator's data.

Parameters

- **itr** (const map_itr_t *) – pointer to map_itr_t

map_iterate(m, fn, userptr)
Iterate an hashmap calling cb on each element until MAP_OK is returned (or end of hashmap is reached). Returns MAP_MISSING if map is NULL or empty. If fn() returns a value != MAP_OK, iteration will stop and: if value < MAP_OK, value will be returned, else MAP_OK will be returned.

Parameters

- **m** (`map_t *`) – pointer to map_t
- **fn** (`const map_cb`) – callback to be called
- **userptr** (`void *`) – userdata to be passed to callback as first parameter

map_put (m, key, val)
Put a value inside hashmap. If key is already present, old value will be updated (and, if dtor is set, destroyed). Note that if new value and old value are the same pointer, nothing will be done.

Parameters

- **m** (`map_t *`) – pointer to map_t
- **key** (`const char *`) – key for this value
- **val** (`void *`) – value to be put inside map

map_get (m, key)
Get an hashmap value.

Parameters

- **m** (`map_t *`) – pointer to map_t
- **key** (`const char *`) – key for this value

Returns void pointer to value, on NULL on error.

map_has_key (m, key)
Check if key exists in map.

Parameters

- **m** (`map_t *`) – pointer to map_t
- **key** (`const char *`) – desired key

Returns true if key exists, false otherwise.

map_remove (m, key)
Remove a key from hashmap.

Parameters

- **m** (`map_t *`) – pointer to map_t
- **key** (`const char *`) – key to be removed

map_clear (m)
Clears a map object by deleting any object inside map.

Parameters

- **s** (`map_t *`) – pointer to map_t

map_free (m)
Free a map object (it internally calls map_clear too).

Parameters

- **m** (`map_t *`) – pointer to map_t

map_length (m)
Get map length.

Parameters

- **m** (`map_t *`) – pointer to map_t

Returns map length or a map_ret_code if any error happens (map_t is null).

CHAPTER 10

Stack API

Libmodule offers an easy to use stack implementation, provided by <module/stack.h> header. It is used internally to store module's stack for become/unbecome methods.

10.1 Structures

```
typedef enum {
    STACK_WRONG_PARAM = -4,
    STACK_MISSING,
    STACK_ERR,
    STACK_OMEM,
    STACK_OK
} stack_ret_code;

/* Callback for stack_iterate */
typedef stack_ret_code (*stack_cb)(void *, void *);

/* Fn for stack_set_dtor */
typedef void (*stack_dtor)(void *);

/* Incomplete struct declaration for stack */
typedef struct _stack stack_t;

/* Incomplete struct declaration for stack iterator */
typedef struct _stack_itr stack_itr_t;
```

10.2 API

Where not specified, these functions return a stack_ret_code.

stack_new(fn)
Create a new stack_t object.

Parameters

- **fn** (const stack_dtor) – callback called on value destroy. If NULL, values won't be automatically destroyed.

Returns pointer to newly allocated stack_t.

stack_itr_new(s)
Create a new stack_itr_t object. Note that it can be freed with free().

Parameters

- **s** (const stack_t *) – pointer to stack_t

Returns pointer to newly allocated stack_itr_t.

stack_itr_next(itr)
Get next iterator. If next iterator is past last element, iterator will be automatically freed.

Parameters

- **itr** (stack_itr_t *) – pointer to stack_itr_t

Returns pointer to next iterator.

stack_itr_get_data(itr)
Get iterator's data.

Parameters

- **itr** (const stack_itr_t *) – pointer to stack_itr_t

Returns pointer to current iterator's data.

stack_itr_set_data(itr)
Set iterator's data.

Parameters

- **itr** (const stack_itr_t *) – pointer to stack_itr_t

stack_iterate(s, fn, userptr)
Iterate a stack calling cb on each element until STACK_OK is returned (or end of stack is reached). Returns STACK_MISSING if stack is NULL or empty. If fn() returns a value != STACK_OK, iteration will stop and: if value < STACK_OK, value will be returned, else STACK_OK will be returned.

Parameters

- **s** (stack_t *) – pointer to stack_t
- **fn** (const stack_cb) – callback to be called
- **userptr** (void *) – userdata to be passed to callback as first parameter

stack_push(s, val)
Push a value on top of stack.

Parameters

- **s** (stack_t *) – pointer to stack_t
- **val** (void *) – value to be put inside stack

stack_pop(s)
Pop a value from top of stack, removing it from stack.

Parameters

- **s** (stack_t *) – pointer to stack_t

Returns void pointer to value, on NULL on error.

stack_peek(s)
Return top-of-stack element, without removing it.

Parameters

- **s** (stack_t *) – pointer to stack_t

Returns void pointer to value, on NULL on error.

stack_clear(s)
Clears a stack object by deleting any object inside stack.

Parameters

- **s** (stack_t *) – pointer to stack_t

stack_free(s)
Free a stack object (it internally calls stack_clear too).

Parameters

- **s** (stack_t *) – pointer to stack_t

stack_length(s)
Get stack length.

Parameters

- **s** (stack_t *) – pointer to stack_t

Returns stack length or a stack_ret_code if any error happens (stack_t is null).

CHAPTER 11

Queue API

Libmodule offers an easy to use queue implementation, provided by <module/queue.h> header.

11.1 Structures

```
typedef enum {
    QUEUE_WRONG_PARAM = -4,
    QUEUE_MISSING,
    QUEUE_ERR,
    QUEUE_OMEM,
    QUEUE_OK
} queue_ret_code;

/* Callback for queue_iterate */
typedef queue_ret_code (*queue_cb)(void *, void *);

/* Fn for queue_set_dtor */
typedef void (*queue_dtor)(void *);

/* Incomplete struct declaration for queue */
typedef struct _queue queue_t;

/* Incomplete struct declaration for queue iterator */
typedef struct _queue_itr queue_itr_t;
```

11.2 API

Where not specified, these functions return a queue_ret_code.

queue_new(fn)
: Create a new queue_t object.

Parameters

- **fn** (const queue_dtor) – callback called on value destroy. If NULL, values won't be automatically destroyed.

Returns pointer to newly allocated queue_t.

queue_itr_new(q)
Create a new queue_itr_t object. Note that it can be freed with free().

Parameters

- **s** (const queue_t *) – pointer to queue_t

Returns pointer to newly allocated queue_itr_t.

queue_itr_next(itr)
Get next iterator. If next iterator is past last element, iterator will be automatically freed.

Parameters

- **itr** (queue_itr_t *) – pointer to queue_itr_t

Returns pointer to next iterator.

queue_itr_get_data(itr)
Get iterator's data.

Parameters

- **itr** (const queue_itr_t *) – pointer to queue_itr_t

Returns pointer to current iterator's data.

queue_itr_set_data(itr)
Set iterator's data.

Parameters

- **itr** (const queue_itr_t *) – pointer to queue_itr_t

queue_iterate(q, fn, userptr)
Iterate a queue calling cb on each element until QUEUE_OK is returned (or end of queue is reached). Returns QUEUE_MISSING if queue is NULL or empty. If fn() returns a value != QUEUE_OK, iteration will stop and: if value < QUEUE_OK, value will be returned, else QUEUE_OK will be returned.

Parameters

- **s** (queue_t *) – pointer to queue_t
- **fn** (const queue_cb) – callback to be called
- **userptr** (void *) – userdata to be passed to callback as first parameter

queue_enqueue(q, val)
Push a value on top of queue.

Parameters

- **s** (queue_t *) – pointer to queue_t
- **val** (void *) – value to be put inside queue

queue_dequeue(q)
Pop a value from top of queue, removing it from queue.

Parameters

- **s** (queue_t *) – pointer to queue_t

Returns void pointer to value, on NULL on error.

queue_peek(q)
Return queue's head element, without removing it.

Parameters

- **s** (queue_t *) – pointer to queue_t

Returns void pointer to value, on NULL on error.

queue_clear(q)
Clears a queue object by deleting any object inside queue.

Parameters

- **s** (queue_t *) – pointer to queue_t

queue_free(q)
Free a queue object (it internally calls queue_clear too).

Parameters

- **s** (queue_t *) – pointer to queue_t

queue_length(q)
Get queue length.

Parameters

- **s** (queue_t *) – pointer to queue_t

Returns queue length or a queue_ret_code if any error happens (queue_t is null).

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Index

Q

R

S