LiteSpeed Web Server Module Developer's Guide

v 0.5

http://litespeedtech.com/
http://open.litespeedtech.com/mediawiki/

Date	Owner	Rev	Description
Feb 4, 2014	David Xu	0.1	Initial Draft
Jul 17, 2017	Ron Saad	0.2	Updates through OpenLiteSpeed 1.4
Dec 5, 2017	Bob Perper	0.3	Threading and other updates.
Jan 22, 2018	Ron Saad	0.4	Updates and cleanup
Jan 31, 2018	Ron Saad	0.5	Updates and cleanup

Abstract

This document provides instructions for developing LiteSpeed modules using the LiteSpeed Internal Application Programming Interface, LSIAPI. The intended audience includes software developers interested in creating LiteSpeed Web Server (LSWS) add-on modules to add or enhance the functionality of LiteSpeed Web Server.

Module support is available in OpenLiteSpeed 1.3.

Multi-threaded module support is available in OpenLiteSpeed 1.5 or greater.

Table of Contents

1. INTRODUCTION

LiteSpeed Web Server (LSWS) is a high performance event-driven web server. OpenLiteSpeed is an open source web server offering many of the features available in LSWS.

Typically web servers utilize external request handlers (cgi, script engines, etc.) to process and provide responses for client requests. LiteSpeed servers support LSAPI, CGI, FCGI, AJPv13 and proxy protocol server APIs for communicating with external request handlers.

The LiteSpeed Internal Application Programming Interface (LSIAPI) was introduced in OpenLiteSpeed 1.3 and LSWS Enterprise 6.0 to allow third party software developers to develop add-on modules that expand server functionality.

Unlike external request handlers, add-on modules execute within the server processes. Rather than executing relatively expensive inter-process communication (IPC) through sockets/pipes/shared-memory, etc., modules communicate with the server kernel via direct function calls, using the interface defined in `LSIAPI`. An add-on module can thus deliver the highest performance possible. Using the interface provided by `LSIAPI`, an add-on module can plug into several server processing steps and perform internal tasks which cannot be handled by an external request processor.

Add-on modules can be used for security-screening of requests / responses, embedding scripting language engines, optimizing content delivery, and more.

As add-on modules execute within the server process, module developers must utilize caution in module development and configuration, as a bug in a module may crash the whole server process. Additionally, some options (such as suEXEC, or changing user ID) are not readily available. Developers should be aware of such caveats before deciding to write an add-on module instead of using an external request processor.

1.1. Audience

This guide is for experienced C/C++ developers interested in adding functionality to the LiteSpeed Web Server via module code. Developers must be familiar with C as the interface assumes a basic understanding of the C development environment.

The LiteSpeed Internal Application Programming Interface (`LSIAPI`) is written in C and C++ and facilitates embedding applications into the LiteSpeed web server. Modules using the `LSIAPI` are loaded directly into the LiteSpeed web server, and crashes within module code can take the entire web server down. Developers are expected to have experience producing reliable, stable code in C and/or C++.

While web **server** development experience is not strictly necessary, it can certainly help in understanding the concepts, interfaces and framework covered in this guide.

OpenLiteSpeed v1.5 or later supports multi-threaded modules in addition to the single thread execution model. Developers opting for multithreading are expected to follow thread-safe code practices. The POSIX pthreads documentation may serve as a good starting point.

2. DEFINITIONS

This section provides definitions for acronyms and industry terminology.

- Callback function - A function passed by pointer to other code to be invoked for execution at another time, mainly associated with a Hook Point.
- Context - A set of resources grouped by URI that can be served by the server.
- Handler - A web server component that processes requests and returns responses.
- Hook Point - One of a multitude of events or states within the web server application at which a callback function may be registered.
- Hook Point Level - The request processing stage with available hook point. `LSIAPI` includes hooks at the TCP L4 level, the HTTP level, the main server level and the worker level, allowing modules to interact with the server at well defined processing stages.
- Filter, Data Filter - A callback function used to produce output based on input data.
- Filter Chain - A group of filters run sequentially based on priority. Each may process data in different ways.
- Handler Module- A module designed with a set of callback functions called by the server to process a certain type of request.
- HTTP - HyperText Transfer Protocol.
- HTTP Session - The process of servicing an HTTP request from beginning to end.
- LSAPI - LiteSpeed Server API protocol for processing specific data with external applications. For example, lsphp follows the LSAPI protocols and handles PHP data.
- `LSIAPI` - LiteSpeed Internal Application Programming Interface for interacting with modules.
- Module, Add-On Module - A dynamic library (.so) complied with the LSIAPI module definition that can then be used by the server.
- Module Configuration Parameters - parameters specified at the server or VHost level which allow configuration of the module, independent of any Request header or other information.
- Process - A heavyweight unit of execution where each program has independent data, handles and state that are not shared with other processes. A Process can have threads. OpenLiteSpeed 1.5 can operate be as a single process with multiple threads. Prior versions are solely multiple processes with one thread each.
- Query String - The parameters following a URI.
- Race Condition - In a threaded process, unguarded access of a shared resource by multiple threads. Unmitigated race conditions may result in corrupted data, undefined behavior, and crashes.
- Request - An HTTP request, comprised of request header and optional body.
- Request Body - The part following the request header of an HTTP request.

- Request Header - The header part of an HTTP request.
- Response - An HTTP response, comprised of response header and optional body.
- Response Body - The part following the response header of an HTTP response.
- Response Header - The header part of an HTTP response.
- Session - The combination of an HTTP session and TCP L4 session.
- TCP L4 - Transmission Control Protocol Layer 4 describing the socket connection transport layer.
- TCP L4 Session - The process of servicing a TCP Layer 4 socket connection from the point when it is opened until it is closed.
- Thread - a lightweight sequence of execution with data and handles shared with all of the threads in a process. In a thread, access to shared data must be protected to avoid race conditions.
- URI - Universal (Uniform) Resource Identifier. A virtual path to identify a resource.
- URL - Universal (Uniform) Resource Locator. Usually comprised of protocol, host, URI and query string.
- WebAdmin Console - A web-based user interface that allows the administrator to configure and control the web server application.

3. REFERENCES

Some of the external resources this document references include:

In the source (OpenLiteSpeed) directory:

- `addon/example/` - Examples of module development
- `include/` - `LSIAPI` include directory
- `include/ls.h` - main `LSIAPI` include file
- `src/modules/` - System configured modules.

Other resources:

- The LiteSpeed LSAPI Reference Manual
- The LiteSpeed Module Administrator's Guide
- LiteSpeed Server WebAdmin Console >Help – Instructions on adding and enabling modules for run-time inclusion.
- For multi-threaded coding: Wikipedia (https://en.wikipedia.org/wiki/Thread_(computing))

4. MODULE DEVELOPMENT OVERVIEW

4.1. Main Components

Working with modules involves three main components: the server core, the modules, and the LiteSpeed Internal Application Programming Interface (`LSIAPI`).

The server core is the main body of the web server application, which receives HTTP requests and sends back HTTP responses. It selects the appropriate *request handler* to process each request based on the server configuration. In addition to the server's built-in request handlers (*e.g.*, "static file handler", "CGI handler", "LSAPI handler"), the system provides for the addition of customized handlers through the configuration of *modules* as handlers.

A module is user-defined code which interfaces with the server, packaged as a dynamically loadable library. If registered via server configuration, modules are dynamically loaded into the server's address space at run-time. A module may function as a handler to service requests, or may serve as a *data filter* to process request/response data.

The `LSIAPI` serves both as a bridge layer, enabling communication between the module and the server core, and as an insulation layer between the two. It provides mechanisms for the server core and the modules to cooperate in performing tasks, while avoiding unnecessary exposure of internal implementation details. A module can be individually enabled or disabled without affecting functions of either the server core or of other modules.

Modules are loaded by the `LSIAPI` module manager per the directives in the server configuration. Through `LSIAPI`, a module can register *callback functions* to be called directly by the server core at defined request processing stages (*hook points*). Inside callback functions, a module can access specific elements of the server core internal data via the server API functions that are defined in `LSIAPI`.

Modules can have configuration parameters stored in the server configuration for the server as a whole or in each virtual host. Space separated, title/value pairs by line are stored in the configuration for the module itself. Accessing these parameters is described below.

Figure-1 below shows the relationship between the server core, the modules, and `LSIAPI`.

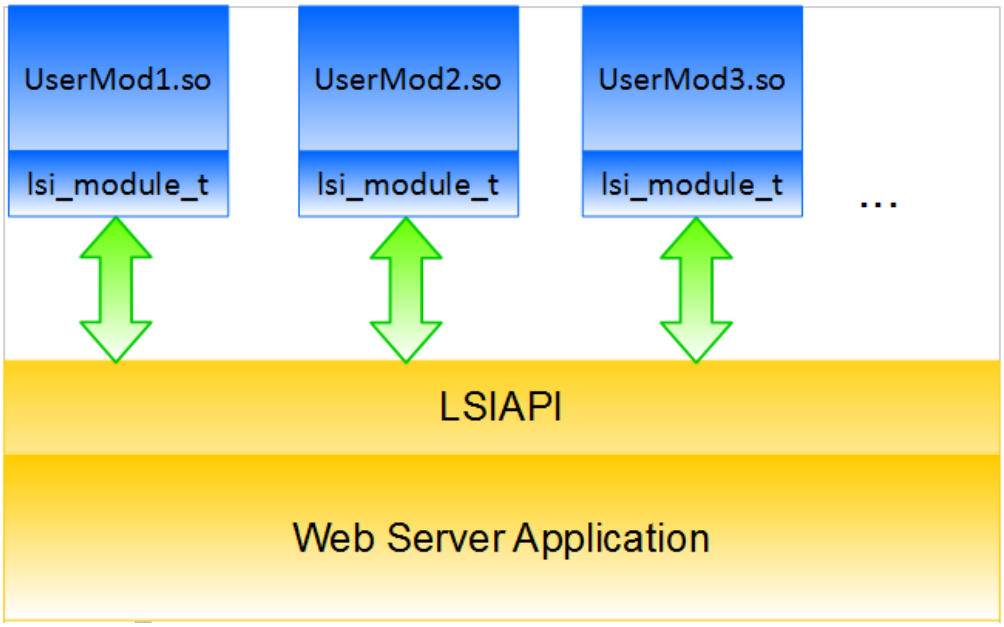


Figure 1: High Level System Diagram

4.2. HTTP Request Processing Overview

The server code processes an HTTP request in the following steps, handing off some of the processing to the corresponding request handler:

1. **Server**: Read the request headers (including request method, URL, protocol, and headers).
2. **Server**: Process the request headers to determine the corresponding request handler, looking up that information in the server configuration.
3. **Server**: Assign a request handler and have the handler perform initial processing of the request data (see 4.6. Request Handler Assignments).
4. **Server**: Optionally, if there is a request body, read the request body and forward it to the request handler.
5. **Request Handler**: The request handler sets the response headers.
6. **Request Handler**: The request handler generates the response body *chunks*, which are buffered by the server.
7. **Server**: The server sends the response header back to the client before sending the response body.
8. **Server**: The server flushes the buffered response body to the client. This is either initiated by the request handler or is triggered because the buffered body segment has passed the buffer's upper limit threshold.
9. **Request Handler**: The request handler generates more response body segments, to be buffered by the server, repeating step 8.

10. **Request Handler**: The request handler finishes generating the response body. The server releases the request handler.
11. **Server**: The server continues to flush all buffered response body segments to the client.
12. **Server**: Clean up resources associated with this request/response.

The above steps together represent an “HTTP session”. When working with modules, the processing steps are modified to include handing control to module defined code at a specific hook point, as explained in 4.5. Request Processing Call Flow.

4.3. Module - Server Interface

The interface between modules and the server code provided by `LSIAPI` includes a server core API, a module API, a hook point API, and a data storage API. A logging API is provided for adding module generated messages to the server logs.

4.3.1. Important Rules for Writing a Module

Modules are written to be shared libraries. That means that the code will be loaded into the OpenLiteSpeed server core and executed there. They must be stable as a crash in a module will take down the web server.

In multi-threaded modules, it is the module developer’s responsibility to guard critical sections (share resources) in module code (request and response functions, etc.) as module threads generally share data and other resources (address space, most data, open file descriptors, signals, current working directory, user and group id, etc.) with all other threads in the process. Generally, module code should limit data access to that provided by the specific `LSIAPI` calls, and should not attempt to change such shared elements as the current working directory, the user id, etc.

There are times when a module will return a character array and a length. It should not be assumed that the character array has a 0 terminator (a C string). If a C string is needed, copy the data to a separate memory array and add a 0 terminator. Data owned by the server should not be modified by modules.

The LiteSpeed Web Server does not expect exceptions and will crash should one occur. All errors need to be reported directly and returned to the server via return code.

4.3.2. Server Core API

The server core API allows modules to access data and features provided by the server core, including:

- Request header and body data
- Response header and body data
- TCP level connection data and control
- HTTP session data
- Environment data
- Global and module user-defined data
- Module configuration parameters
- Logging facilities
- Handler registration
- Hook point management
- Timer events
- Network event notification control

API functions can be accessed by a module via the global variable `g_api` declared in `ls.h`:

```
extern const lsi_api_t *g_api;
```

The OpenLiteSpeed API exports the g_api global which is used to access all OpenLiteSpeed functions. For example:

```
g_api->enable_hook( ... );
g_api->log( ... );
```

4.3.3. Module API

The module API provides the server with entry points into the module code. It is defined in the `lsi_module_t` data structure.

Every module must have a unique name. The module must declare its own global variable instance of `lsi_module_t` type. Recommended practice is to use a C preprocessor directive to define `MNAME` as the actual module name, and subsequently only refer to `MNAME` throughout the module code. The name selected for the module must match the name of the generated shared library file (without the `'.so'` extension).

```
#define         MNAME         mymodule
lsi_module_t MNAME;  // forward decl
...
module code
...
```

```
LSMODULE_EXPORT lsi_module_t MNAME = { LSI_MODULE_SIGNATURE, init, NULL, NULL, "",
serverHooks};
```

Aside from the mandatory `signature` element in `lsi_module_t`, the other elements include (optionally NULL) pointers to a module initialization function, a request handling control structure, a configuration parameter parsing function, a short module description string, and to the initial set of server hooks to be registered for the module. The presence of a non-NULL request handling pointer (`lsi_reqhdlr_t *`) enables the module to serve as a request handler, whereas many filter modules only need to provide the server hooks information.

The request handling type `lsi_reqhdlr_t` provides callback function pointers for handling specific phases of the request, as well configuration details for single vs. multi-threaded module operation.

The API uses the types defined in ls.h to pass information between modules and the server. Session related data is typically passed via an `lsi_session_t` pointer. These are typically provided to module code in calls to module functions, and are passed back to server code via module api calls as needed.

4.3.4. Hook Point API

The hook point API is used to facilitate calling module functions at specific processing steps, including specific Server and Worker related execution points (Server Processing Flow), and request handling stages (Request Processing Call Flow, below). The Hook point API includes the function pointer `lsi_callback_pf` and the callback parameter to be passed to the function, `lsi_param_t`, and `lsi_serverhook_t` the as defined in `ls.h`:

```
// typedef int  ( *lsi_callback_pf )( lsi_param_t * ); // in ls.h

int session_hook_func( lsi_param_t * param) {
        …
}

lsi_callback_pf pHf = session_hook_func;
int flag = LSI_FLAG_ENABLED;

static lsi_serverhook_t server_hooks[] =
{
…
{ LSI_HKPT_RECV_RESP_BODY, pHf, LSI_HOOK_NORMAL, flag },
…
```

```
}

static int init_module()
{
        ...
}

LSMODULE_EXPORT lsi_module_t MNAME =
{
LSI_MODULE_SIGNATURE, init_module, NULL, NULL, "", server_hooks, {0}
};
```

Callback functions must be registered in order to be called by the server core. They are registered using the `lsi_serverhook_t` array that is referenced in the `lsi_module_t` structure. Once registered, the callback functions of a module can be turned on and off using the `enable_hook` API function.

```
int hooks[] = {LSI_HKPT_RECV_RESP_BODY, LSI_HKPT_SEND_RESP_BODY};
int num = sizeof(hooks) / sizeof(int);
int enable = 1; // 0 to disable
g_api->enable_hook(session, &MNAME, enable, hooks, num);
```

See 5.1.5. Module Configuration and 5.4. Hook Points and Callback Function Summary for details.

4.3.5. Data Storage API

The data storage API provides a facility to store data for a module within the server core objects and make it available for the module's use when module code is invoked. This is done by storing a pointer to module allocated data in specific server objects, and calling the module supplied data release function (to deallocate the memory) when the object lifecycle is complete.

Access to the Data Storage API, as for all LiteSpeed functions is via the *_module_data() functions through the `g_api` pointer. See the OpenLiteSpeed `addon/example/testmoduledata.c` module for an example.

The module selects the object storing the module data by the level argument to `init_module_data`, defined by LSI_DATA_LEVEL (currently HTTP, FILE, IP, VHOST, and L4), and may store different data pointers at each of these levels.

Module data stored at VHOST level will remain available until the VHOST is released, typically at server shutdown time. This is an appropriate level for data that should remain valid while the VHOST is up. Once stored, the data is available to the module across all client requests handled by the same VHOST.

Module data stored at IP level will remain valid for the life cycle of an IP address in the LiteSpeed Web Server ClientCache. The release function will be called once the cache entry is cleared (typically due to inactivity timeout). This data is available to the module across multiple L4 sessions from the same client IP.

Module data stored at L4 level will not be released until the L4 connection is closed. Such data may be useful for information that should be available across multiple HTTP sessions in an L4 session (with keepalive connections).

HTTP level module data is released once the HTTP request handling has completed.

FILE level module data remains valid as long as the file entry remains in the file cache. When a file entry is evicted (typically due to usage timeout), the release callback function is executed.

A typical call sequence may look like:

```
// typedef int (*lsi_datarelease_pf)(void *); // in ls.h

LSMODULE_EXPORT lsi_module_t MNAME = // module definition

int my_release_func_http( void * dataPtr ) {
        ... cast dataPtr and release memory
}

int my_release_func_vhost( void * dataPtr ) {
        ... cast dataPtr and release memory
}

const lsi_session_t * session = //value passed into the function;

myHttpData = new MyType;                    // http data type
myVhostData = malloc(sizeof(VHostData));    // vhost data size
```

```
init_module_data(&MNAME, my_release_func_http, LSI_DATA_HTTP);
init_module_data(&MNAME, my_release_func_vhost, LSI_DATA_VHOST);
...
set_module_data(session, &MNAME, LSI_DATA_HTTP, (void *) myHttpData);
...
set_module_data(session, &MNAME, LSI_DATA_VHOST, (void *) myVhostData);


...
MyType * myHttpData = (MyType *)get_module_data(mySession, &MNAME, LSI_DATA_HTTP);
...
// memory will be freed by server calling the my_release_func_* functions
```

When working with file based data, an additional call must be made to the API function `init_file_type_mdata`. See the 5.5.3. Module Data reference section for details.

4.3.6. Logging API

LSIAPI provides mechanisms to add formatted output to the server log files.

The `LSM_*` macros defined in ls.h, including `LSM_ERR`, `LSM_WRN`, `LSM_NOT`, `LSM_INF` and `LSM_DBG` (for error, warning, notice, information, and debug logging levels, in order), are the recommended practice for module use, as they log the module name, session information (can be NULL is not available), thread id (if appropriate) and more. The log level used is defined in the server configuration.

For example, to log as an error, a return code from an external function into the error log:

```
LSM_ERR((&MNAME), session, "External function return code: %d\n", rc);
```

The macros accept printf variadic arguments and do not insert newlines, thus if a new line is desired it must be included in the format string.

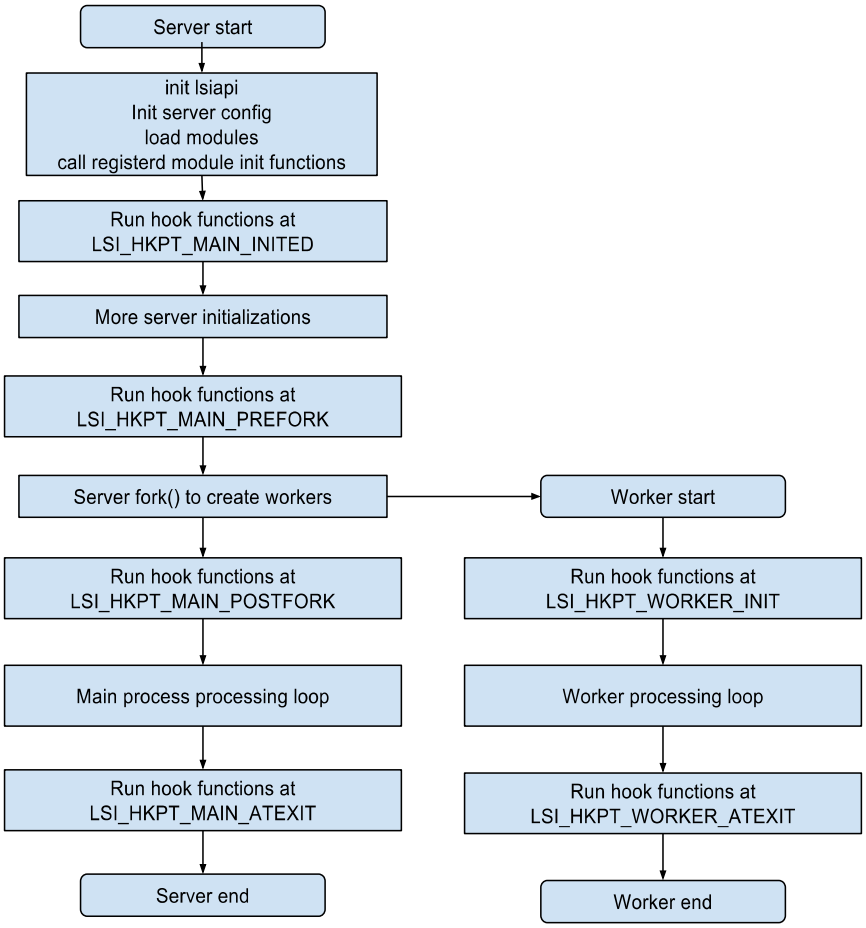
Direct access to the server logging facilities is available by calling the variadic log() function or providing vlog() with a va_list. When the pSession argument is not NULL, the session ID will be automatically added to the log message.

The lograw() function simply appends `len` bytes of `buf` to the log files without any processing.

4.4. Server Processing Flow

The LiteSpeed Web Server uses Worker processes when handling requests, and provides hook points for modules that need to execute actions related to the Server and Worker lifecycles (perhaps for efficiency to avoid per-request execution, or to maintain data over multiple requests, etc.).

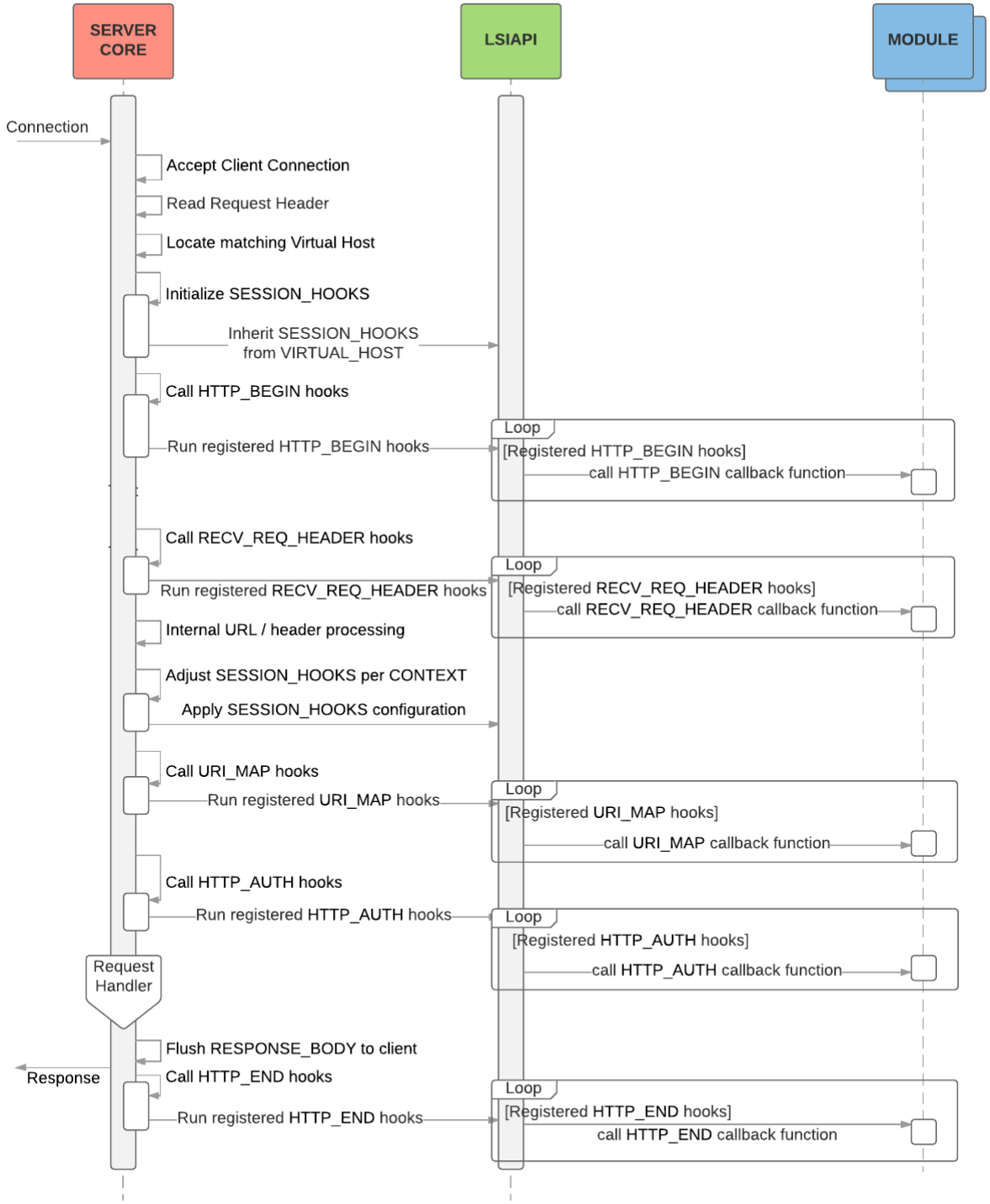
These Server and Worker hook points are handled as shown in the diagram below.



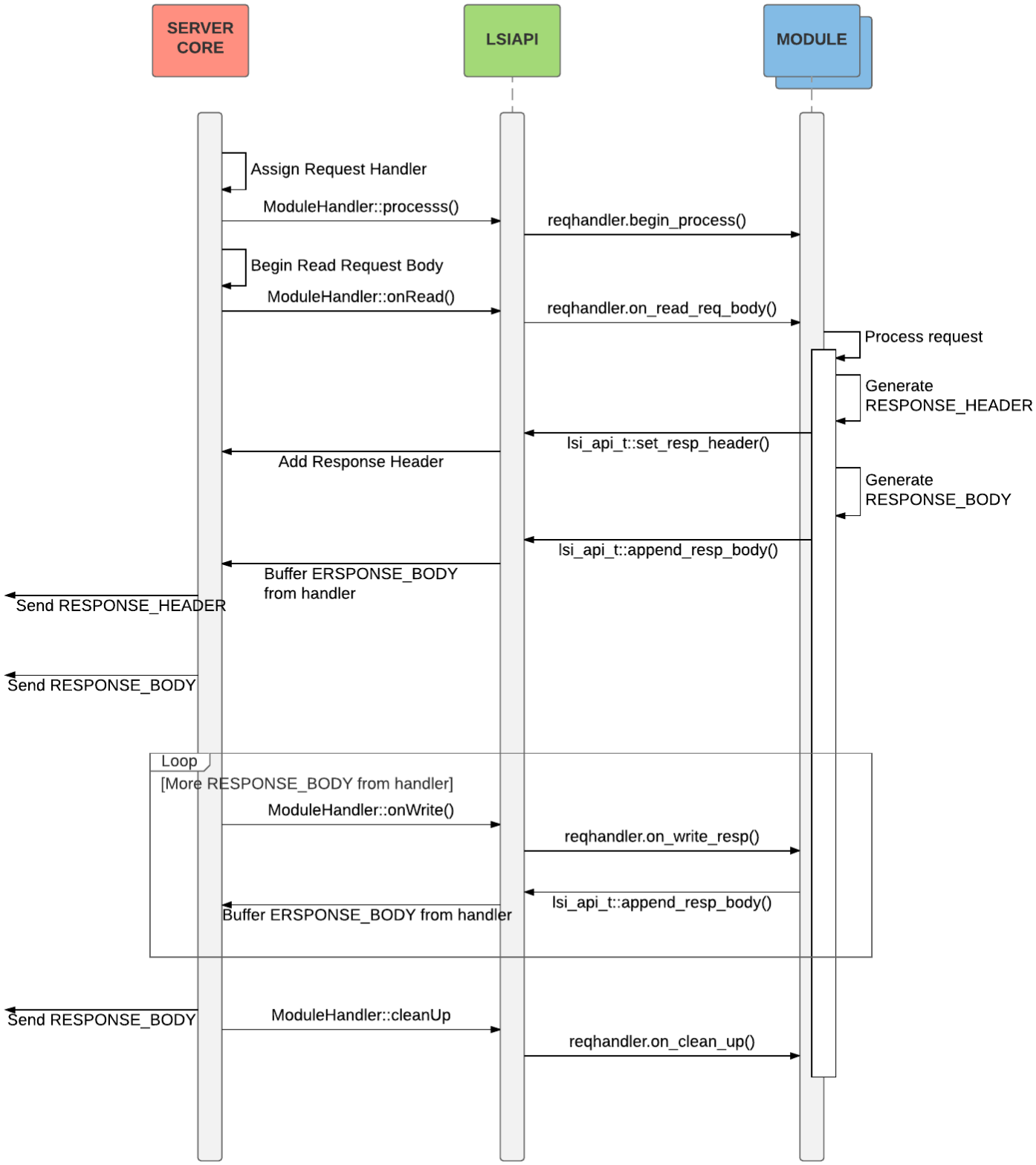
4.5. Request Processing Call Flow

The steps of processing an HTTP request are augmented when using modules as shown in the following diagrams, and shown in the searchable text table below.

REQUEST PROCESSING CALL FLOW



REQUEST HANDLER CALL FLOW



LiteSpeed Server Core Actions	LSIAPI Actions	Module Actions
Accept Client connection Read REQUEST_HEADER Locate matching VIRTUAL_HOST Initialize SESSION_HOOKS ➔		
	Inherit SESSION_HOOKS from VIRTUAL_HOST	
Call HTTP_BEGIN hooks ➔		
	Run registered HTTP_BEGIN hooks ➔	
		HTTP_BEGIN callback function invoked
Call RECV_REQ_HEADER hooks ➔		
	Run registered RECV_REQ_HEADER hooks ➔	
		RECV_REQ_HEADER callback function invoked
Internal URL/HEADER processing		
Adjust SESSION_HOOKS based on CONTEXT ➔		
	Apply CONTEXT SESSION_HOOKS configuration	
Call URI_MAP hooks ➔		
	Run registered URI_MAP hooks ➔	
		URI_MAP callback function invoked
Call HTTP_AUTH hooks ➔		
	Run registered HTTP_AUTH hooks ➔	
		HTTP_AUTH callback function invoked
Assign REQUEST_HANDLER		

Invoke MODULE_HANDLER ➔

ModuleHandler::process() ➔

reqhandler.begin_process() invoked

Read REQUEST_BODY

Call Handler OnRead() ➔

ModuleHandler::onRead() ➔

reqhandler.
on_read_req_body() invoked

←Generate RESPONSE_HEADER

←lsi_api_t::set_resp_header()

Add RESPONSE_HEADER

←Generate RESPONSE_BODY

←lsi_api_t::append_resp_body()

Buffer RESPONSE_BODY from handler

←Send RESPONSE_HEADER

←Send RESPONSE_BODY

More RESPONSE_BODY from handler?

ModuleHandler::onWrite() ➔

reqhandler.on_write_resp() invoked

←Send remaining RESPONSE_BODY

← lsi_api_t::append_resp_body()

Buffer RESPONSE_BODY from handler

←END_OF_RESPONSE

handler cleaning up

ModuleHandler::cleanUp() ➔

reqhandler.on_clean_up() invoked

←Flush RESPONSE_BODY to client

Call HTTP_END hooks ➔

Run registered HTTP_END hooks ➔

HTTP_END callback function invoked

4.6. Request Handler Assignments

The server core must choose one and only one handler to process a request. There are two ways this can be done: static assignment and dynamic assignment.

4.6.1. Static Assignment

Generally, the server core picks a request handler from the pool of internal handlers and module handlers based on server configuration. Assigning a module to handle a request through the configuration is called static assignment. A suffix handler associates a handler with a file suffix, and a context handler matches a handler with the request URI.

The LiteSpeed Module Administrator's Guide has additional details on this type of configuration.

4.6.2. Dynamic Assignment

Additionally, a module can explicitly ask the server core to assign a request to itself via the API call **register_req_handler**(). The module can request this dynamic assignment by calling the API within the callback function registered to the `LSI_HKPT_HTTP_BEGIN`, `LSI_HKPT_RECV_REQ_HEADER`, or `LSI_HKPT_URI_MAP` hook points.

Only the first module to call **register_req_handler**() succeeds. Calls to **register_req_handler**() that find an existing module-type handler registered will fail. Once a handler has been assigned by the server core (the `URI_MAP` hook being the final recommended opportunity), calling **register_req_handler**() has no effect.

The distribution example code in `hellohandler.c` was designed to require static registration via the server configuration.

The example in `hellohandler2.c` was developed to register itself as the handler with the API **register_req_handler**(), but can also be registered in the configuration.

4.7. A “Hello World” Module

In this section we’ll step through creating a “Hello World” handler module. The module name will be “mod_hello”, and its function would be so send the string “Hello World!.” to the client as HTML text.

The code shown is very similar to the examples provided in the distribution. The `lsi_module_t` module definition object is named ‘mod_hello’ using the recommended C preprocessor macro, and contains the address of an `lsi_module_t` request handler object ‘myhandler’, that includes a pointer to the request handling callback function `begin_process`. As the module relies on `LSIAPI`, we will need to configure the file system location of the `LSIAPI` directory in order to build our module.

1. Create a directory for the module code. We used `mod_hello`.

2. In the mod_hello directory, we create a file `mod_hello.c` with the following code:

```
#include "ls.h"

#define MNAME mod_hello
static int begin_process(const lsi_session_t *session); // forward declaration

/**
 * Define a handler, need to provide a lsi_reqhdlr_t object, in which
 * the first function pointer should not be NULL
 */
static lsi_reqhdlr_t myhandler = { begin_process, NULL, NULL, NULL, NULL, NULL, NULL };
LSMODULE_EXPORT lsi_module_t MNAME = { LSI_MODULE_SIGNATURE, NULL, &myhandler,
NULL, NULL, NULL, {0} };

static char resp_buf[] = "Hello World!.\r\n";

static int begin_process(const lsi_session_t *session)
{
g_api->set_status_code(session, 200);
g_api->set_resp_header(session, LSI_RSPHDR_CONTENT_TYPE, NULL, 0,
"text/html", 9, LSI_HEADEROP_SET);
```

```
    g_api->append_resp_body(session, resp_buf, sizeof(resp_buf) - 1);
    g_api->end_resp(session);
            return 0;
    }
```

3. To build the module, we use one of the following methods:
    1. Directly compile the module with the following commands:

        gcc -g -Wall -c -D_REENTRANT -I/LSIAPIDIR/include -fPIC mod_hello.c
        gcc -g -Wall -o mod_hello.so mod_hello.o -shared

Where `LSIAPIDIR` is the installation / source directory containing the `LSIAPI include` folder.

    2. The `addon/example` includes a shell script to help developers build modules named `ccc.sh`. We can copy `ccc.sh` to our working directory, then use the commands

        EXPORT LSIAPIDIR=/directory/containing/LSIAPI_include/
        ./ccc.sh mod_hello.c

to build the module (note we set the environment variable to point to our `LSIAPI` directory first).

    3. The `addon/example` also provides a Makefile developers may use, after editing the value of `LSIAPIDIR` in the Makefile.

make

Following any of the above methods, `mod_hello.so` has now been created.

In order for the module to be loaded during server startup, it must be registered in the server configuration and added to the server modules directory. While the details are available in the LiteSpeed Module Administrator's Guide, the following quick steps will set up the module in OpenLiteSpeed:

- In the Web Admin server configuration, under modules, add the module with the name 'mod_hello'
- In the VirtualHost settings select the Example VHost, then add a Context with a path of /mod_hello and a handler of type module, select mod_hello as the module handler.
- Copy (or link) the mod_hello.so file to the server modules directory (<*ServerRoot*>/modules/).
- Save the configuration and perform a graceful restart of the server.

Browsing to the virtual host url with a path of /mod_hello should display “Hello World!” in the client.

4.8. Multi-Threaded Modules

LiteSpeed Web Server provides a multi-threading model for request handler modules, which allows for greater efficiency and higher performance. Module developers who wish to take advantage of this feature must be aware of the underlying architecture.

4.8.1. Built In Multi-Threading

The LiteSpeed built-in multi-threading feature is enabled by setting the `ts_hdlr_ctx` field, in the `lsi_reqhdlr_t` object, to the predefined value `LSI_HDLR_DEFAULT_POOL`. All other `LSIAPI` calls and objects remain the same and the multi-threading interface is handled transparently by `LSIAPI`.

When configuring modules for multi-threading, developers must not make assumptions regarding the thread executing their code. For example, the request handling callback function may be run in a different thread than the main server or other callbacks (hooks) registered for the module.

For multi-threaded modules, the `lsi_reqhdlr_t` object is treated in a different manner than for single threaded modules, in the sense that the callbacks for `on_read_req_body` and `on_write_resp` are not invoked by the server. The expectation is that the thread handling reading and writing in the module handler callback is using the `LSIAPI` calls `read_req_body` and `append_resp_body`, which internally use synchronization (event notifiers) between threads to facilitate the read / write operations.

While details may change in the future, the current implementation runs the module’s `begin_process` callback for each session in a separate thread, and typically request body reads and response body writes are routed to notifiers that awake any threads awaiting these events, while all other module callbacks are invoked from the server main thread.

This approach allows simplifying the module’s request handling code by allowing it to block without affecting other sessions.

One optimization applicable to multi-threaded modules is that module code is not required to call `end_resp()` - the proper response is sent by the main server thread when the `begin_process` callback returns, indicating the handling of the request is complete. This optimization saves a thread context switch, providing a slight performance advantage.

Given the multi-threaded execution, developers must be sure to use thread-safe function calls and to protect critical regions (shared data, etc.) with synchronization mechanisms to ensure no race conditions, data corruption or deadlocks occur. For example, if a module uses the Data Storage API, the pointers are shared between the module threads, and access to them should either be limited to a single thread or otherwise properly guarded.

Validation tools such as ThreadSanitizer (part of gcc since 4.8 and clang since 3.2) are highly recommended to ensure correct code behavior. There should be no data races reported when running under ThreadSanitizer.

To aid in diagnosing module code, the `LSM_*` logging functions include the thread id in the log messages where appropriate.

For examples of modules using multi-threading, check the `addon/example` directory for files name “*mt*.c”, such as mtaltreadwrite.c, mteventseq.c, testredirectmt.c, etc.

4.8.2. Custom Multi-Threading

TBD.

5. MODULE DEVELOPMENT REFERENCE

This section of the guide focuses on the usage of the LSIAPI functionality, detailing the how-to aspects of using the interface. The interface specification, including function names, return value and argument types, struct definitions, etc., is detailed in the LiteSpeed LSAPI Reference Manual.

5.1. Modules

A module is defined by an instance of the `lsi_module_t` data structure and related implementation functions packed into a dynamic library. Each module MUST have a unique name. The dynamic library must use the naming scheme `<ModuleName>.so`.

All modules need to be placed in the directory **<***ServerRoot***>**`/modules/`. For example, in a typical production environment where the server has been installed in `/usr/local/lsws`, all modules need to be placed in the `/usr/local/lsws/modules/` directory.

5.1.1. Components of a Module

A module MUST have:

- An `lsi_module_t` instance declared as a global variable whose name MUST match the module name (recommended practice: use the C preprocessor to define `MNAME` as the actual desired module name, then use `MNAME` throughout the code).

A module MAY have one or more of the following components:

- An initialization routine to be called when a module is initially loaded by the system.
- Configuration parameter parsing functions, if the module configuration parameters are integrated with the LiteSpeed WebAdmin Console.
- Callback functions for each hook point that the module wants to plug into.
- User-defined global, module, or environment data.
- User-defined environment callback functions.

A **handler module** MUST have:

- An `lsi_reqhdlr_t` instance declared. Though not required, declaring the variable *static* helps avoid possible name collision with other modules.

- A callback function assigned to the `lsi_reqhdlr_t::begin_process` function pointer. Similarly, it is recommended that this be declared as a static function.

A **handler module** MAY have:

- A callback function for events related to reading the request body (single-threaded modules only).
- A callback function for events related to continued writing of the response body (single-threaded modules only).
- A callback function for server core to call to perform clean up.
- A non-NULL `ts_hdlr_ctx` pointer (multi-threaded modules only).
- A non-NULL `ts_enqueue_req` pointer (multi-threaded modules only).
- A non-NULL `ts_cancel_req` pointer (multi-threaded modules only).

5.1.2. Module Definition Structure

The `lsi_module_t` structure defines the module interface to the LSIAPI. The name of this variable, the module name, and the basename of the dynamic loadable library MUST all be the same. For example, if the module name is “mytest”, the library name MUST be `mytest.so`.

For consistency, the recommended practice is to define (C preprocessor) `MNAME` as the desired module name, and reference `MNAME` throughout the code.. For example:

```
#define MNAME mymodule
…
static lsi_serverhook_t serverHooks[] = { ... };
static lsi_reqhdlr_t reqhdlr = { ... } ;
static int init_module(lsi_module_t *pModule) { ... }
LSMODULE_EXPORT lsi_module_t MNAME = {
        LSI_MODULE_SIGNATURE,
        init_module,    // Optional module initialization function
        &reqhdlr,               // Request handler function
        cfg_parser,             // Optional config parser
        "Module about",         // Module "about" string
        serverHooks,            // Optional server hook routines
        {0}                     // Initializer for reserved element
};
```

The source file that holds this variable can have any name, but generally it is named using the module name, as it usually holds the main body of the module. The example Makefile provided

in the distribution treats each C code file in the compilation directory as if it contains a single, complete module, and compiles it to a shared library file with the same base name.

A module MUST declare one and only one instance of the data structure `lsi_module_t` as a statically initialized global variable. `lsi_module_t` is defined in the `include/ls.h` file located in the OpenLiteSpeed source directory or LiteSpeed Web Server Enterprise's decompressed download directory.

Description:

- **`LSI_MODULE_SIGNATURE`** is the structure signature and is required.
- **Init_module** If a module requires any initialization code, `init_pf` should point to an optional module initialization routine, otherwise `init_pf` should be NULL. Many modules use module initialization routines to also initialize any libraries that are used at the module level (all sessions).
- **reqhdlr** If the module performs a handler role, `reqhandler` defines special callback routines, otherwise it should be set to NULL. While static initialization is recommended, `reqhandler` can also be initialized in a module initialization routine. See 5.2. Handler Modules for further details.
- **cfg_parser** If a module needs its own configuration, and configuration parameters are integrated with the server WebAdmin Console (as opposed to using a custom configuration file, for example), `config_parser` specifies an interface for parsing the module configuration parameters. Otherwise `config_parser` must be NULL. If used, `config_parser` MUST be statically initialized. See 5.5.2. Configuration Parameters for details.
- **"Module about"** The optional `about` specifies module information for logging purposes. It can be initialized statically or during module initialization. The server will log this string after the module has been successfully initialized.
- **server_hooks** Global server level callback functions are defined in `serverhook`.

5.1.3. Module Initialization

A module is loaded and initialized after server startup, at the server configuration stage. Initialization takes place in the following steps:

1. The Server starts up and loads the main configuration.
2. The Server processes the module configuration section and gets a list of modules that need to be loaded.

3. For each module, the server calls the `LSIAPI` Module Manager to load a module with that module name.
4. The Module Manager searches the `<ServerRoot>/modules/` directory to locate the dynamic library with filename `<ModuleName>.so`, and loads that library into memory.
5. The Module Manager locates the global variable of the `lsi_module_t` instance with the module name. If not found, this module initialization fails.
6. If `config_parser` is not `NULL`, and the WebAdmin configuration contains space separated key-value configuration parameters for the module, the Module Manager calls the configuration parsing routine to parse the module configuration parameters stored in the server configuration.
7. If the `init_pf` function pointer is not `NULL`, the routine is called to perform module initialization tasks. Global-level module configurations parsed in step 6 can be accessed inside the initialization routine. Optionally, the`about` string can be updated to include information about this module.
8. If the module initialization succeeds, indicated by a return value of 0 from `init_pf`, the Module Manager writes a log message saying the module has been initialized and prints out the `about` string.
9. A return value of `-1` from `init_pf` indicates that module initialization has failed, so the module manager disables the module and writes a log message saying that it has failed to load.
10. Steps 3-9 are repeated until all modules have been processed.

5.1.4. Initialization Routine LSIAPI Access

In the module initialization routine, a number of tasks can be performed, such as loading additional dynamic libraries, initializing script language interpreters, creating temporary directories/files, allocating shared memory blocks, etc.

The initialization routine may use only module- and global-access `LSIAPI` functions, since session-level data is not yet available. Below are API functions that can be used during initialization.

Function group	LSIAPI Functions	Description
Module Data Management	init_module_data() set_module_data() get_module_data() free_module_data() init_file_type_mdata()	Initialize module data or module data files.

Environment Handling	register_env_handler()	Register environment handlers.
Global Utility Functions	get_server_root()	Get the server's root directory.
Logging	log() vlog() lograw()	Write to the server error logs.
Thread Handling	register_thread_cleanup()	Register a function to be called when a thread terminates, if threading is used which is managed by the server.
Module Configuration Parameters	get_config()	Obtain and complete parsing of configuration parameter information which was loaded at the time the module was loaded.

Table 1: API functions in Initialization

5.1.5. Module Configuration

In order to be loaded and utilized by the server, a module must be registered in the module configuration section of the server configuration. The module configuration section can be found in the WebAdmin Console under the Server > Modules tab.

Once registered, the callback functions of a module can be turned on and off at different levels:

- Global level
- Listener level
- Virtual Host level
- Context level

An option is available in the WebAdmin Console to disable the module. Disabling the module effectively turns off all registered callback functions within the module.

A module may create its own standalone custom configuration file, or have its configuration integrated into the server configuration. If integrated, configuration parameters can be managed through the WebAdmin Console. Note that only space separated key-value pairs are recognized as configuration parameters, and the module configuration callback function will only be called if

there are valid configuration parameters. Refer to the Configuration Parameters section of this document for details.

The Server module configuration globally defines the module configuration data. Once defined, the Listeners and Virtual Hosts have access to the modules and module configurations. The Server configuration defines the default values for module parameter data and filter control. These values can be inherited or overridden by the Listener and Virtual Host configuration data. Module priority is only defined at Server level and is inherited by the Listener and Virtual Host configurations.

5.2. Handler Modules

In LiteSpeed Web Server, an HTTP request must be assigned to a request handler, which is responsible for generating the response headers and body.

The response headers can be modified at later hook points before being sent out. If necessary, the response body can only be modified through the **filter** hook points.

A module can act as a customized HTTP request handler by defining a set of callback functions and making them available through the main module data structure. A handler module is the most common type of module.

5.2.1. Handler Structure and Callback Functions

A module that acts as a handler must provide a set of callback functions and statically initialize the member object`reqhandler` pointing to an instance of `lsi_reqhdlr_t`. The routines are called at appropriate times by the server core.

The `begin_process()` function starts processing a request for the current HTTP session specified by *pSession*. This is the only required callback function for a request handler. It is called after the `URI_MAP` hook point, when a module handler has been assigned to a request. In `begin_process()`, a module can access request headers and the request body received, set the response status code, add response headers, and write the response body. The `end_resp()` API should be called whenever a module handler finishes sending the response body.

On success, `begin_process()` should return 0, otherwise it should return an HTTP status code. The server core will send the corresponding HTTP error page back to the client.

The optional `on_read_req_body()` function may be called to process the request body. However it may not be called, even if provided, if the entire request body was available when the request was begun. This function is made available, and may optionally be called by the server. It is recommended when it is possible to process the request as quickly as as it comes

in as it allows control to be returned quickly to the server for optimal performance. Note that to process the body, it must be obtained by a call to `read_req_body()` or one of the other functions described below as the data is not directly passed to the functions.

The `set_req_wait_full_body()` API determines if the function is called when the server receives a portion of the request body, or after the complete request body has been received.

The return value of `begin_process()` should normally be 0, but is ignored by the server core.

The optional `on_write_resp()` function is called when the server flushes the buffered response body to the client. A module handler can continue to add more contents to the response body in this callback function. The `set_handler_write_state()` API can turn this callback function on and off.

`on_write_resp()` should return:

- `LSI_WRITE_RESP_ERROR` if an error occurred,
- `LSI_WRITE_RESP_FINISHED` if the response is completed, or
- `LSI_WRITE_RESP_CONTINUE` if the response is not completed and on_write_resp() needs to be called again.

The optional `on_clean_up()` is called to perform cleanup tasks after the handler is done in the following situations:

- The handler finished processing the request normally.
- The server need to cancel the current handler, in follow situations: server need to send back a HTTP error page (default or generated by another handler), an internal redirect has been triggered, the client cancelled the session by killing the connection while the request was still being processed.

In other words, if a handler has been assigned by server, `on_clean_up()` will be called when the handler is finished or cancelled.

Example:

In our mod_hello example, `end_resp()` is called at the end of `begin_process()`. Since we are neither processing the request body nor writing more response body, `on_read_req_body` and `on_write_resp` are set to `NULL`.

```
#include "ls.h"

#define MNAME mod_hello
```

```
static int begin_process(const lsi_session_t *session)
{
...
g_api->end_resp(session);
        return 0;
}
 /**
  * Define a handler, need to provide a struct _handler_st object, in which
  * the first function pointer should not be NULL
  */
static lsi_reqhdlr_t myhandler = { begin_process, NULL, NULL, NULL };
LSMODULE_EXPORT lsi_module_t MNAME = { LSI_MODULE_SIGNATURE, NULL, &myhandler,
NULL, "", serverHooks, {0} };
```

5.2.2. Accessing Request Data

When the server starts to process a request, the callback **begin_process**() is called. In this function, the user has access to the HTTP request data of the session (request header and request body), as well as a number of related environment variables. **On_read_req_body**() is another important handler callback with access to request data.

5.2.2.1. Request Header

API functions for accessing request header related data:

```
get_req_header()
get_req_header_by_id()
get_req_raw_headers()
get_req_raw_headers_length()
get_req_uri()
get_req_org_uri()
get_req_query_string()
get_req_cookies()
get_req_cookie_count()
```

These functions retrieve request header values for the session specified by their *pSession* argument. Values may be returned through the caller's specified buffer at *buf*, whose maximum size is *maxlen* or *buf_size*. The size of returned data is returned through the `int` pointer *valLen*, *uri_len*, or *len*.

Request Body

API functions for accessing the request body:

```
get_req_content_length()
read_req_body()
```

```
is_req_body_finished()
get_req_body_buf()
set_req_wait_full_body()
```

These functions manage the retrieval of the request body for the session specified by *pSession*.

The `read_req_body()` function is the primary routine to retrieve request body data. If data is available, the server core makes a deep copy of the request body to the caller's supplied buffer at *buf*, whose maximum size is *bufLen*. The routine returns the size of the data returned or zero if no data is currently available.

For any module handler that processes the request body, the handler callback functions `begin_process()` and `on_read_req_body()` should be implemented including calls to `read_req_body()` to retrieve the data, along with calls to the function `is_req_body_finished()` to ensure all the data has been accounted for.

The request body could be very large. The default method to read a large request body is block by block. In this case, the server core first calls the handler's `begin_process()` callback to begin processing, and then makes subsequent possibly multiple calls to the handler's `on_read_req_body()` callback as new data *chunks* become available. These callback routines should call `read_req_body()`, possibly multiple times, while data is available, and process it appropriately. The server core keeps track of how much data has been received from the client, and how much it has passed to the handler.

Alternatively, if a module handler wants to wait until the full request body is available, it can call the `set_req_wait_full_body()` API function inside the `begin_process()` callback function. In this case, the server core calls the handler's `on_read_req_body()` callback function only after the entire request body has been received.

It is also possible to call `set_req_wait_full_body()` ***before*** `begin_process(,)` from a callback function registered to the `HTTP_BEGIN`, `RECV_REQ_HEADER`, or `URI_MAP` hook point. In this case, the server core calls `begin_process()` only after the full body has been received, and does *not* call `on_read_req_body()` at all.

For example, if a module wants to scan the full request body before handler processing, it can call `set_req_wait_full_body()` in a callback function registered to `HTTP_BEGIN`.

IMPORTANT: if any module calls `set_req_wait_full_body()`, the action cannot be undone.

Finally, for GET requests and requests with a small body received entirely by the server with the request headers, *only* the handler's `begin_process()` is called; `on_read_req_body()` is *not* triggered.

The example module "waitfullreqbody.c" is an example of waiting for the full body before processing.

Server and Environment Variables

The following API functions access request variables created by the server and environment variables:

```
get_client_ip()
get_req_env()
get_req_var_by_id()
```

These functions retrieve values for the session specified by *pSession*. Values may be returned through the caller's specified buffer at *val*, whose maximum size is *maxValLen*.

5.2.3. Generating a Response

A module handler can call API functions to set response headers and/or append to the response body from the three handler callback functions, **begin_process**(), **on_read_req_body**(), and **on_write_resp**().

5.2.3.1. Response Header

API functions for manipulating the response header:

```
set_status_code()
get_status_code()
set_resp_content_length()
set_resp_header()
set_resp_header2()
get_resp_header()
get_resp_headers_count()
get_resp_headers()
remove_resp_header()
```

These functions manage the response headers for the session specified by *pSession*. *Header_id* specifies a response header `enum`; if set to `LSI_RESP_HEADER_UNKNOWN`, *name* and *nameLen* are used instead to specify the header id name string.

The response status code is part of the response header; the default is usually "200". The status code can be changed with the function `set_status_code()`.

The “Content-Type” header should be set for a response with a response body; otherwise, it is not necessary to set it.

A module does not have to set the response body size, but the `set_resp_content_length()` function can be used to specify this. If set, the module must send the response body with this *exact* size; otherwise, it may break the HTTP protocol. If a module tries to send a larger response body, the server will ignore the extra bytes; if a module does not provide enough response body data, the server will wait until `end_resp()` is called. Then, the underlying stream will close if the server receives insufficient response body data.

The response headers are sent out by the server core before the response body upon the first call to `flush()`. After this, modifications to the response headers have no effect.

5.2.3.2. Response Body

API functions for generating response body:

```
is_resp_buffer_available()
append_resp_body()
append_resp_bodyv()
send_file()
flush()
end_resp()
set_handler_write_state()
```

These functions manage the response body for the session specified by *pSession*.

The response body can be dynamic content generated on the fly, static content stored in files, or a mix of both.

Dynamic content is added with the `append_resp_body()` and `append_resp_bodyv()` functions, and is stored in the response body buffer managed by the server core. It is recommended that the `is_resp_buffer_available()` function be first called to determine if the server core is willing to buffer more response body data. The server core stores dynamic content in a memory mapped temporary file if the body size is larger than 1MB.

Static content can be added to the response body by calling `send_file()`. `send_file()` is an asynchronous call which tells the server to queue the operation and returns immediately. The server core can only queue one file. The module handler must return from its callback function immediately after calling `send_file()`; any instructions after the call will cause undefined behavior. The handler should wait for the next on_write_resp() callback to add more content to the response body.

Any pending response body can be flushed by calling the `flush()` API call. The server core first flushes pending dynamic content, then sends out the static content, then calls the `on_write_resp()` callback function to let the module handler add additional content to the response body.

If the response body is complete within `begin_process()`, the `end_resp()` function should be called to initiate the response send operation. If not, the `on_write_resp()` handler callback function should be defined to finish the response body.

`on_write_resp()` should return:

- `LSI_WRITE_RESP_FINISHED` when the response body is finished and should be sent (in which case `end_resp()` is *not* called),
- `LSI_WRITE_RESP_CONTINUE` to continue processing and request another callback,

or

- `LSI_WRITE_RESP_RET_ERROR` on error.

The `set_handler_write_state()` function can be used to suspend/resume the `on_write_resp()` callback. If a handler temporarily has nothing to add to the response body, it should suspend the `on_write_resp()` callback and resume it when more content needs to be added.
The example module “testtimer.c” demonstrates how to use `set_handler_write_state()`.

If a connection to a client is lost in the middle of a transmission, the server core will trigger the `HTTP_END` hook point. It is recommended that a handler add a callback function to this hook point so the module will be notified if the connection is interrupted, providing an entry point for any necessary cleanup.

5.3. Data Filter Modules

A filter is a callback function that is registered to a hook point for processing data from other sources. There are five filter hook points:

- L4_RECVING
- L4_SENDING
- RECV_REQ_BODY
- RECV_RESP_BODY
- SEND_RESP_BODY

Filters are intended for adding additional processing steps and are very versatile. While the server core processes a request or response, filters can be used to inspect and transform HTTP requests and response body data, implement socket connection level data encryption, etc. Filters can work together with any HTTP request handler.

If a filter callback function does not modify the input data, it is an “observer” filter; otherwise, if a filter modifies the input data, it is a “transformer” filter.

5.3.1. How Filters Work

At the base level, a filter is simply a callback function which produces output based on the given input. Although a module can register only one callback to a specific hook point, multiple modules can register to the same filter hook point. Multiple filters can then be chained together to form a filter chain. Each filter in the chain takes input, processes it, and passes its output as the input to the next filter in the chain. The order of the filter chain is determined by callback function priority. For callback functions with the same priority, the order of registration determines priority; earlier registration means higher priority. The data is processed by filters with higher priority first.

A filter may be classified as either an:

- Input stream filter, or
- Output stream filter

An input stream filter works in a way similar to the `read()` system call: the server core passes to the filter chain, a buffer and the maximum number of bytes. Each filter takes input from the previous, higher priority filter (or the server core), processes it, then passes the processed output to the next, lower priority filter until the end.

An output stream filter works in a way similar to the `write()` system call: the server core passes to the filter chain, a buffer and the size of data to be sent. Filters in the chain are called from high priority to low, and after each filter finishes processing its input data, it sends the result to the next filter. The last (lowest priority) filter is provided by the server core to either write to the underlying stream or save in a buffer.

5.3.2. Data Buffering

A filter processes an input data stream and generates an output data stream, which becomes the input to the next filter in the chain.

During data processing, it is not guaranteed that a filter will receive all necessary input data in one block, and the next filter is not guaranteed to finish consuming the whole block of output data. A filter may also need additional data to finish processing the current data block; *e.g.*, if it

needs to wait for more data in order to process as a whole piece. For these reasons, a filter may need to buffer input data and/or output data in its own buffers at the session level.

Since the input buffer is only valid for the current function call, a filter MUST NOT simply save the pointer to the input data for later use, but must make a deep copy of the portion of input data needed.

An “observer” filter does not change the content of the input stream, so the output stream is the same as the input stream. However, the filter may still need to buffer output data that the next filter in the chain could not yet accept. The buffered data needs to be passed to that filter before the current filter is invoked again.

A “transformer” filter, a filter which modifies the input data, may need to manage two buffers: one for the input data stream, one for the output data stream.

However, to optimize performance, a minimal usage of buffers and deep copies is recommended.

For an observer, it may be possible to pass the input data directly to the next filter without making deep copies.

For a transformer, the filter can avoid an input buffer by processing all input data immediately, then saving the result in the output buffer. Of course, this is not always possible; *e.g.*, a block of data that can only be decrypted after a complete block has been received.

A filter should follow the following principles to minimize the usage of buffers:

- A filter should flush pending data in the output buffer to the next filter first.
- A filter should return 0 if the output stream buffer is not empty.
- A filter should process input data in units such as 4K or 8K, then call the next filter to process the result immediately. If the next filter cannot take it all, buffer the remaining result data, and stop processing input data.

5.3.3. Creating a Filter

To add a filter hook, a module must provide a callback function, and register it to one of the filtering hook points. The filter callback function is passed a pointer to an `lsi_param_t` structure which defines callback parameters and specifies buffers for input or output. The function can call `LSIAPI` routines such as `stream_read_next()`, `stream_write_next()`, and `stream_writev_next()`. After filter processing, it must send its result as input to the next filter.

The `stream_read_next()` function is called from filter callback functions registered to the `LSI_HKPT_L4_RECVING` and `LSI_HKPT_RECV_REQ_BODY` hook points to get data from the higher level filter in the chain. The callback parameters are specified by *pParam*. *pBuf* specifies the buffer to hold the read data, and *size* defines the buffer size.

The `stream_write_next()` function is called from filter callback functions registered to the `LSI_HKPT_RECV_RESP_BODY` and `LSI_HKPT_SEND_RESP_BODY` hook points to send data to the next filter in the chain. The callback parameters are specified by *pParam*. *Buf* specifies the buffer to write, and *len* defines the buffer size.

`stream_write_next()` is called from filter callback functions registered to the `LSI_HKPT_L4_SENDING` hook point after it finishes the action and needs to call the next step. *Iov* specifies the vector of data to be written, and *count* specifies its size.

Example:

```
#include "ls.h"

static int httpreqread( lsi_param_t *pParam )
{
   filter processing, g_api->stream_read_next(), ...
}

static int handlerBeginProcess( lsi_session_t *pSession )
{
   ...
}

static int init_module( lsi_module_t *pModule )
{
   pModule->about = VERSION;
   return 0;
}

static lsi_serverhook_t serverHooks[] =
{
   ...
  { LSI_HKPT_RECV_REQ_BODY, httpreqread, LSI_HOOK_EARLY, LSI_HOOK_FLAG_TRANSFORM },
   lsi_serverhook_t_END   // Must put this at the end
```

```
};


static lsi_reqhdlr_t reqHandler = { handlerBeginProcess, NULL, NULL, NULL, NULL, NULL, NULL
};
LSMODULE_EXPORT lsi_module_t MNAME =
  { LSI_MODULE_SIGNATURE, init_module, &reqHandler, NULL, "v1.0", serverHooks, {0} };
```

5.3.4. HTTP Request Body Filter

The HTTP Request Body Filter must register with the hook point `RECV_REQ_BODY`.

All request body data received by the server core passes through this filter, and is then saved into the request body buffer managed by the server core.

In this filter, certain members of the `lsi_param_t` input parameter have specific usages (previously discussed in 5.4.6. Callback Function Parameters):

```
ptr1: the result data buffer as input for the next filter.
len1: size of the result buffer.
flag_out: set to LSI_CBFO_BUFFERED when current filter has buffered data.
flag_in: not used.
```

This filter first calls `stream_read_next()` to get request body data from a higher priority filter. The highest level filter is provided by the server core which reads the original request body data from the underlying stream.

If this is an observer filter, the result buffer pointer parameters **ptr1** and **len1** may be passed directly to `stream_read_next()`, to hold the data. This is an optimization, saving extra buffers and a data copy. The return value of `stream_read_next()` should then be used as the return value of this filter.

```
static int httpreqread( lsi_param_t *pParam )
{
   ...
   len = g_api->stream_read_next( pParam, (char *)pParam->ptr1, pParam->len1 );
   ...

   return len;
```

```
}
```

In a transformer filter, the function must supply its own buffer to `stream_read_next()`. Depending on how the data is processed, the buffer can be allocated from the stack with a local variable if it can be processed all at once, or dynamically allocated as session-level module data if data must be processed in blocks. Module data is discussed in 5.5.3. Module Data.

```
static int httpreqread( lsi_param_t *pParam )
{
   char tmpBuf[MAX_BLOCK_BUFSIZE];
   ...
   len = g_api->stream_read_next( pParam, tmpBuf, sizeof(tmpBuf) );
   if process data all at once, and result_len <= pParam->len1,
      put result in pParam->ptr1
   ...

   return result_len;
}
```

`stream_read_next()` returns the size of data read. If the filter's input buffer is not completely filled, `stream_read_next()` can be called multiple times as long as its return value is > 0.

If `stream_read_next()` returns 0, there is nothing to read from a higher priority filter, so the filter should return the current size of the result data.

If `stream_read_next()` returns < 0, something is wrong with the input stream. The filter should return `LSI_RET_ERROR`.

The processed results should be saved to the result buffer. If the result buffer is not big enough to hold the full result, the remaining portion needs to be buffered in a dynamically allocated session level buffer (see 5.5.3. Module Data) and `flag_out` needs to be set to `LSI_CBFO_BUFFERED`. When this filter is next called, it needs to continue to flush the result to the result buffer supplied before it can process more input.

Filters should try to minimize buffered data by only read enough input to create result data that exactly fills up the result buffer.

On success, the return value of this callback function should be the size of the processed result data being returned at this time.

If a filter callback function returns `LSI_RET_ERROR`, the current request is blocked and no more processing is performed. The callback function can set the response status code for the server core; if not set explicitly by the filter, status code 403 is used. If the error happens after the response header has been sent out, the server core terminates the request by suddenly closing the underlying stream.

Unlike a handler, `set_req_wait_full_body()` has no effect on this type of filter. The server core keeps calling the registered filter when more request body data becomes available.

5.3.5. Incoming Response Body Filter

The Incoming Response Body Filter must register with the hook point `RECV_RESP_BODY`.

Dynamic response body data generated by a handler passes through this filter, and is then saved into the server core managed response body buffer. Response body generated via `send_file()` will *not* go through this filter.

The handler generating the response body may not be able to buffer partial data blocks. Unless running out of response body buffer, the server core always saves the result data block into the response body buffer.

The members of the `lsi_param_t` input parameter with specific usages to this filter are:

ptr1: read-only input data buffer, containing the response body data (possibly the result of from a higher priority filter).

len1: size of the input buffer.

flag_out: should be set to LSI_CBFO_BUFFERED when current filter has buffered data.

flag_in: may include the following bit values:

LSI_CBFI_FLUSH indicates the filter must force flush buffered data and pass the flag to the next filter in the chain.

LSI_CBFI_EOF indicates this is the end of the stream and no further data is available. The filter should ignore any further input. Implies LSI_CBFI_FLUSH. A filter should only set this flag after all buffered data has been sent.

LSI_CBFI_STATIC is only set if the input data is from a static file, and if a filter does not need to check static file data it can skip processing it.

LSI_CBFI_RESPSUCC is set if the request handler does not abort in processing the request (e.g. external handler crash, proxy connection reset, etc.). If a filter only needs to process successful responses, it can skip processing unless this bit is set.

The filter processes the input data to create result data, then calls `stream_write_next()` to write the result to a lower priority filter.

If this is an observer, the input buffer pointer parameters `ptr1` and `len1` may be passed directly to `stream_write_next()` to avoid a deep copy.

```
static int httprespwrite( lsi_param_t *pParam )
{
   ...
   len = g_api->stream_write_next(
     pParam, (const char *)pParam->ptr1, pParam->len1 );
   ...

   return len;
}
```

If this is a transformer, the callback function must store the result in its own buffer. If the input data can be fully processed and the result can be fully written, the buffer can be a local stack variable. If input or result data needs to be processed in blocks, the partial blocks need to be saved into buffers allocated as session level module data – see 5.5.3. Module Data.

`stream_write_next()` returns the size of data accepted by the next filter. If there is pending data to be sent, `stream_write_next()` can be called repeatedly as long as its return value is > 0.

If `stream_write_next()` returns 0, it indicates that the next filter is unable to accept more data at this time, so the current filter should buffer the remaining result data in an allocated session level buffer, `flag_out` needs to be set, and the callback returns. When this filter is next called, it needs to continue to flush the result before it can process more input.

If `stream_write_next()` returns < 0, something is wrong with the output stream. The filter should return `LSI_RET_ERROR`.

On success, the return value of this callback function should be the size of the input data consumed at this time. In the case where a filter does not or cannot process the whole input data, but still accepts the entire input by buffering it, the callback return value should be the entire input size. It is then the responsibility of the module to manage its internal state with respect to input and output processing.

```
static int httprespwrite( lsi_param_t *pParam )
{
   ...
   process all pParam->ptr1 input data, put result data in myResult
```

```
    ...
    while ( myResult.size > 0 )
    {
        blockSz = ( myResult.size < MAX_BLOCKSZ ) ? myResult.size : MAX_BLOCKSZ;
        sz = g_api->stream_write_next( pParam, (const char *)myResult.ptr, blockSz );
        if ( sz < 0 )
            return LSI_RET_ERROR;
        if ( sz == 0 )
            break;
        myResult.ptr += sz;
        myResult.size -= sz;
    }
    if ( myResult.size > 0 )
        *pParam->flag_out |=LSI_CBFO_BUFFERED;
    ...

    return pParam->len1;    // since entire input was consumed, even if result left
}
```

If a filter callback function returns `LSI_RET_ERROR`, the server core interrupts the response processing. If the response header and body have not yet been sent back to the client, the server completely discards the current response, and returns an HTTP error page to the client. The callback function can set the response status code for the server core; if not set explicitly by the filter, status code 403 is used. If the error happens after the response header has been sent out, the server core terminates the response by suddenly closing the underlying stream.

`set_resp_wait_full_body()` can be used to tell the server core to wait for the full response body before sending any response header and body back to the client.

5.3.6. Outgoing Response Body Filter

The Outgoing Response Body Filter must register with the hook point `SEND_RESP_BODY`.

The server core passes both dynamic response body and static file content through this filter, with the result being sent out through the underlying stream. This filter prevents the server core from utilizing the `sendfile(2)` system call.

The members of the `lsi_param_t` input parameter with specific usages to this filter are:

ptr1: read-only input data buffer.
len1: size of the input buffer.
flag_out: set to LSI_CBFO_BUFFERED when current filter has buffered data.

flag_in: not used.

The filter processes the input data to create result data, then calls **stream_write_next**() to write the result to a lower priority filter.

If this is an observer, the input buffer pointer parameters **ptr1** and **len1** may be passed directly to **stream_write_next**() to avoid a deep copy.

If this is a transformer, the callback function must store the result in its own buffer. If the input data can be fully processed and the result can be fully written, the buffer can be a local stack variable. If input or result data needs to be processed in blocks, the partial blocks need to be saved into buffers allocated as session level module data. Module data is discussed in section 9.3 of this document.

The filter is not required to process or buffer the whole input data, though buffering should be avoided when possible and minimal buffering is recommended.

Stream_write_next() returns the size of data accepted by the next filter. If there is pending data to be sent, **stream_write_next**() can be called repeatedly as long as its return value is > 0.

If **stream_write_next**() returns 0, it indicates that the next filter is unable to accept more data at this time, so the current filter should buffer the remaining result data in an allocated session level buffer, **flag_out** needs to be set, and the callback returns. When this filter is next called, it needs to continue to flush the result before it can process more input.

```
*pParam->flag_out |=LSI_CBFO_BUFFERED;
```

If **stream_write_next**() returns < 0, something is wrong with the output stream. The filter should return `LSI_RET_ERROR`, and the underlying stream is closed.

On success, the return value of this callback function should be the size of the input data consumed at this time; *i.e.*, the size of processed input data, plus buffered input data.
On error, the callback should return `LSI_RET_ERROR`.

The example module “updatehttpout.c” demonstrates this type of filter.

5.3.7. L4 Input Stream Filter

The L4 Input Stream Filter must register with the hook point `L4_RECVING`.

This filter works in a way similar to the HTTP Request Body Filter.

See the updatetcpin*.c files in the addon/example directory.

5.3.8. L4 Output Stream Filter

The L4 Output Stream Filter must register with the hook point `L4_SENDING`.

This filter works in a way similar to the Outgoing Response Body Filter.

See the updatetcpout*.c files in the addon/example directory.

5.4. Hook Points and Callback Function Summary

A hook point is a location where the server core calls registered callback functions during the course of processing a request. A module can register a callback function to a hook point to perform certain tasks. There are two types of hooks: event and filter. An event hook provides notification that the server processing is at a specific point, allowing the module to take some action, but the data cannot be modified. A filter utilizes a callback function at the appropriate time for examining, processing, or modifying data from other sources.

5.4.1. Hook Points Overview

Hook points are defined by `enum LSI_HKPT_LEVEL` in `ls.h` (while all enumeration value names start with `LSI_HKPT_`, the tables and discussion below strip that common prefix for brevity.)

Additionally, the tables list the type of each hookpoint, where the type may be `E` for event and `F` for filter. Filter hook points enable a module to modify the associated data stream before it is passed along for further processing (by the server or other modules).

As shown in 4.4. Server Processing Flow and 4.5. Request Processing Call Flow, the server core exposes a number of hook points at different stages of the server lifecycle and request processing.

Each Module is allowed to register at most one callback function at each hook point. Module Priority Levels are set by the module when registering a callback function, and can also be adjusted via module configuration.

If more than one callback function (from multiple modules) are registered to a given hook point, the callback functions are called in order of Module Priority level, from the lowest numeric value (first) to the highest (last).

The allowable range of priority values is defined in `ls.h`, from `LSI_HOOK_FIRST` to `LSI_HOOK_PRIORITY_MAX`. While any number in the defined range is valid, the recommended practice is to select one of the predefined levels defined by `enum LSI_HOOK_PRIORITY` in `ls.h`.

A callback function can direct the server core to bypass any remaining callback functions in the chain via its return value. These return values include:

- LSI_SUSPEND: Suspend the current hookpoint (the module should arrange for resumption of processing).
- LSI_DENY: Deny access - stop processing and return an error code.
- LSI_ERROR: Indicate an error has occurred.
- LSI_OK: Processing should continue.

5.4.2. Server Lifecycle Hook Points

For modules that require some processing performed at specific points in the server lifecycle, the following hook points provide such access.

Name	Type	Triggered
MAIN_INITED	E	When the main (controller) process has completed its initialization and configuration, before servicing any requests. It occurs once upon startup.
MAIN_PREFORK	E	When the main (controller) process is about to start (fork) a worker process. This occurs for each worker, and may happen during system startup, or if a worker has been restarted.
MAIN_POSTFORK	E	After the main (controller) process has started (forked) a worker process. This occurs for each worker, and may happen during system startup, or if a worker has been restarted.
WORKER_INIT	E	In a worker process after it has been created by the main (controller) process. Note that a corresponding MAIN_POSTFORK hook may occur in the main process either before *or* after this hook.
WORKER_ATEXIT	E	In a worker process just before exiting.

		It is the last hook point of a worker process.
MAIN_ATEXIT	E	In the main (controller) process just before exiting. It is the last hook point of the server main process.

5.4.3. L4 Hook Points

Four hook points are available at the TCP Layer 4 connection level:

Name	Type	Triggered
L4_BEGINSESSION	E	When a client socket connection is established.
L4_ENDSESSION	E	When a client socket connection is closed.
L4_RECVING	F	When the server receives data from the client.
L4_SENDING	F	When the server sends data to the client.

The most common use of L4 hook points is for encryption or sub-channel multiplexing at the socket connection level. In other web servers, these hook points are used to implement features like SSL or SPDY. (LiteSpeed Web Server has built-in SSL and SPDY support.)

5.4.4. HTTP Session Hook Points

The following hook points are available at the HTTP session level:

Name	Type	Triggered
HTTP_BEGIN	E	When the server core starts to process a request, after the complete request header has been received and the request has been assigned to a virtual host. This event is intended for internal session level resource allocation. Execution of this hook point will not be interrupted by a callback function which can be used to do module preparation before any other tasks are performed..
RECV_REQ_HEADER	E	For initial request header processing. If a module handler is assigned in this hook point, the server built-in URL rewrite and URI_MAP hook point will be skipped.
URI_MAP	E	For mapping URI with request handlers. This is after the

		server built-in URL rewrite and context matching has been performed and before the URI to file system mapping.
HTTP_AUTH	E	For authentication checking.
RECV_REQ_BODY	F	For filtering the request body data while the server reads request body data block by block.
RCVD_REQ_BODY	E	After the full request body has been received. If there is no request body, this event will still trigger.
RECV_RESP_HEADER	E	When the response header is completed by a handler. It is intended for response header inspection, addition, or modification.
RECV_RESP_BODY	F	For filtering response body data generated by an HTTP request handler.
RCVD_RESP_BODY	E	After the full response body has been received. If there is no response body, this event will still trigger. The module should
HANDLER_RESTART	E	When a handler or a callback function performs an internal redirect. After executing this hook point, the server core will reinject the request with a new URL in the processing engine starting with the URI_MAP hook point. The module should release any module data.
SEND_RESP_HEADER	E	Right before the response header is sent to the client, this is intended for last minute response header modification.
SEND_RESP_BODY	F	For filtering the response body data being sent to the client.
HTTP_END	E	When the HTTP session ends. This is intended for resource deallocation and cleaning up. Execution of this hook point will not be interrupted by a callback function.

5.4.4.1. HTTP Request / Response Data Access

The table below shows what request/response data are available for HTTP session level hook points, as well as for the callback functions for a module handler. The table also gives a rough execution sequence of the hook and handler callback functions during an HTTP session. The

full body access option only invokes the module callback after the full request body has been received. The data stream access in called block-by-block as data is available, triggered by socket reads.

R = Read Only, W = Write Only, RW = Read Write, No = No Access

P = Read, it the request body is available (if the request body is small and has been read in, use the read_req_body() api call to access it.

FB = Full Body Access, DS = Data Stream Access

Hook Point	Request Header	Request Body	Response Header	Response Body
HTTP_BEGIN	R	No	No	No
RECV_REQ_HEADER	R	No	No	No
URI_MAP	R	P, R	No	No
HTTP_AUTH	R	P, R	No	No
lsi_reqhdlr_t:: begin_process()	R	P, R	RW	W
RECV_REQ_BODY	R	RW, DS	No	No
lsi_reqhdlr_t:: on_read_req_body()	R	RW, FB + DS	RW	W
RCVD_REQ_BODY	R	R, FB	No	No
RECV_RESP_HEADER	R	R, FB	RW	No
RECV_RESP_BODY	R	R, FB	RW	RW, DS
RCVD_RESP_BODY	R	R, FB	RW	RW, FB
SEND_RESP_HEADER	R	R, FB	RW	P, RW
lsi_reqhdlr_t:: on_write_resp()	R	R, FB	R	W
SEND_RESP_BODY	R	R, FB	R	RW, DS
HTTP_END	R	R, FB	R	R

The request headers and related server internal variables are available for all hook points. Request headers are read-only. Request URL (including URI and Query String) can be modified via API functions.

POST requests contain a request body. The request body length is from the "Content-Length" request header, so the size of the request body is available to all hook points.

After the `RECV_REQ_HEADER` hook point, the server core may try to read a certain amount of the request body before the `URI_MAP` hook point. If the request body is small enough, the hook point sequence may be switched: the `RECV_REQ_BODY` and `RCVD_REQ_BODY` hook points could be triggered before the `URI_MAP` hook point and the point that the server core calls `lsi_reqhdlr_t::begin_process()`.

The content of the request body may be available at the `URI_MAP` hook point. The API function **is_req_body_finished**() can be used to determine if the request body has been completely read or not.

The request body can only be modified at the `RECV_REQ_BODY` filter hook point.

The `RCVD_REQ_BODY` hook is always triggered, no matter if a request has a request body or not.

A module which has defined a `lsi_reqhdlr_t` can make modifications to the response header as early as in the `lsi_reqhdlr_t::begin_process()` callback function. Any modification to response headers prior to this callback will be discarded.

A module without a `lsi_reqhdlr_t` modify the response header in the `RECV_RESP_HEADER` and `SEND_RESP_HEADER` hook points.

The server will flush out response headers before sending the first byte of the response body to the client. Modifications made after sending the response header has no effect. Response status code must be set before sending the response header, the default status code is "`200`".

Response body processing has two main phases that a module can hook into:

- Receiving response body from a handler and storing it in the internal response body buffer managed by the server core. The `RECV_RESP_BODY` filter hook can be used to check/modify response body data before saving it into the body buffer. `RCVD_RESP_BODY` indicates the end of receiving the response body.

- Sending buffered response body to client. The `SEND_RESP_BODY` filter hook can be used to check/modify response body data before sending it out. `HTTP_END` indicates the end of sending the response body.

A module with the handler role is also responsible for generating the response body content prior to processing the `RECV_RESP_BODY` filter.

Module developers must not assume that `RCVD_RESP_BODY` will happen before the `SEND_RESP_BODY` hook. Usually, receiving and sending response body happens in parallel.

The server core can buffer the whole response body. However, if the response body is too large, the server creates a memory mapped temporary file to hold the content.

The server core buffers small chunks of response body data to avoid excessive fragments. Once the total buffered data is larger than a certain size, usually around 1.3K, the server core tries to flush buffered data in a relatively larger chunk. This is to avoid sending too many small chunks while minimizing latency introduced.

Usually, the response is sent over WAN, while the response from the handler backend is received through local IPC or over LAN. Receiving a response may be faster than sending them out. The server may suspend receiving if there is excessive buffered data that could not be sent out to the client.

Note that when debugging a module, that sometimes the response body may be in regular memory or it may be in memory mapped memory.

5.4.5. Defining a Callback Function

All callback functions should follow the prototype defined in `ls.h`:

```
typedef int ( *lsi_callback_pf )( lsi_param_t * );
```

The callback function should return either

- LSI_RET_ERROR (== -1)

or

- LSI_RET_OK (== 0)

If the return value is `LSI_RET_OK`, then the next callback function in the chain is called.

If the return value is `LSI_RET_ERROR`, then the hook chain is stopped.

5.4.6. Callback Function Parameters

When a callback function is called back by the server core, the `lsi_param_t` structure is passed as the input parameter to the callback.

The usages of **ptr1**, and **len1** vary depending on the specific hook point. Environment handler callback functions use the same prototype as callback functions, and are also included in the table below.

Hook Point Level	ptr1, len1
L4_RECVING RECV_REQ_BODY RECV_RESP_BODY	ptr1: input data block buffer, len1: size of the buffer
L4_SENDING	ptr1: `struct iovec` vector, len1: vector length
SEND_RESP_BODY	ptr1: data buffer to be sent, len1: length of the buffer
Environment callback	ptr1: environment variable value, len1: length

Table 2: Callback Parameter Definition

5.4.7. Enabling/Disabling a Callback Function

Callback functions must be registered in order to be called by the server core. Both module level (hooks specific to the module as a whole) and session level (hooks specific to a given session's conversation) are managed in the same way.

5.4.7.1. Managing Hooks

The hooks are registered statically by the module definition. They can be enabled or disabled via server configuration at the Listener, Virtual Host, and Context levels in various combinations.

For example, suppose a module level callback function was configured to be disabled at the Virtual Host level, then enabled for a context under that virtual host. The result would be that the module level callback functions would not be activated for that virtual host except for that specific context.

Note that all callback functions of a module are enabled or disabled at a given level at the same time; an individual callback function cannot be independently enabled or disabled.

Module level hooks are registered with an array of `lsi_serverhook_t` structures, terminated with the special `lsi_serverhook_t_END` define, referenced in the `lsi_module_t` module definition.

index specifies the hook point. The hook point level definitions `LSI_HKPT_*` are in `ls.h`.

cb is a pointer to the callback callback function.

priority defines the priority of this callback function within a function chain (see the `LSI_HOOK_PRIORITY` enum in `ls.h`).

flag lets the data in the stream be further controlled (see the `LSI_HOOK_FLAG` enum in `ls.h`).

`LSI_FLAG_TRANSFORM`: If set indicates that the module will modify the data; otherwise the module passively examines the data.

`LSI_FLAG_DECOMPRESS_REQUIRED`: For `LSI_HKPT_RECV_RESP_BODY` and `LSI_HKPT_SEND_RESP` body filters, the server core will guarantee that uncompressed data is passed to the filter.

`LSI_FLAG_PROCESS_STATIC`: For `LSI_HKPT_SEND_RESP_BODY` filters, if no additional filters are needed to process the static file, sendfile() will be used by the server.

`LSI_FLAG_ENABLE`: Either set this flag to enable a disabled hook or call the enable_hook function.

Example:

If in the “Hello World” module previously discussed we want to, given a specific condition, dynamically register the mod_hello handler via a callback function registered to `LSI_HKPT_RECV_REQ_HEADER` at the module level, the following code can be added:

```
//…(see mod_hello code above)...
static int recv_req_header_cbf( lsi_param_t *param ); // used here, defined below
static lsi_serverhook_t serverHooks[] =
{
  { LSI_HKPT_RECV_REQ_HEADER, recv_req_header_cbf, LSI_HOOK_NORMAL, LSI_FLAG_ENABLE
},
    lsi_serverhook_t_END   // Must put this at the end
};
// Add to MNAME the serverHooks definition:

LSMODULE_EXPORT lsi_module_t MNAME =
  { LSI_MODULE_SIGNATURE, NULL, myhandler, NULL, "v0.0", serverHooks };
```

```
static int recv_req_header_cbf( lsi_param_t *param )
{
    if ( strcmp( "/hello", g_api->get_req_uri( param->session, NULL ) ) == 0 )
    {
        g_api->register_req_handler( param->session, &MNAME, 0 );
    }
    return LSI_RET_OK;
}
... (the begin process in mod_hello above)...
```

Once mod_hello is loaded by the server, it handles all requests to URL “/hello”, echoing back “Hello World!”.

5.4.8. Callback Function Timeline

Data Available at Hook Point	LSIAPI Data Access Functions	Callback Function Purpose
LSI_HKPT_L4_BEGINSESSION	get_gdata_container empty_gdata_container purge_gdata_container get_gdata delete_gdata set_gdata set_timer remove_timer	Initialize user TCP session data.
LSI_HKPT_L4_RECVING	stream_read_next	Get stream data real-time. Continue to read TCP stream data until the return value is 0.
LSI_HKPT_HTTP_BEGIN	get_org_req_uri get_req_uri get_req_ip get_status_code get_module_param add_session_hook	Allocate resources for module data, user-defined data and configuration parameter data. Add hooks that will be used only in the current HTTP session. Register request handlers.

	init_module_data init_file_type_mdata set_module_data get_module_data register_req_handler get_multiplexer	
LSI_HKPT_URI_MAP	get_uri_file_path get_mapped_context_uri redirect	Get the URI to the mapped resource.
LSI_HKPT_HTTP_AUTH		Check authentication.
LSI_HKPT_RECV_REQ_HEADER	get_req_env get_req_env_by_type get_total_req_header_length get_total_req_headers get_cookies get_cookie_count get_cookie_value get_req_header get_req_query_string	Get request header data.
LSI_HKPT_RECV_REQ_BODY	get_req_content_length get_req_body is_req_body_finished set_req_wait_full_body set_handler_write_state	Get request content data real-time. Continue to call this function until the return value is 0 or the length is reached.
LSI_HKPT_RCVD_REQ_BODY	get_req_body_file_fd	Get entire body data from file after it has been received.
LSI_HKPT_RECV_RESP_HEADER	get_resp_headers_count get_resp_header set_resp_header set_resp_header2 set_req_env get_all_resp_headers	Get response header data.

	remove_resp_header	
LSI_HKPT_RECV_RESP_BODY	is_resp_buffer_available append_resp_body append_resp_bodyv	Get a pointer to the response body as callback parameter.
LSI_HKPT_RCVD_RESP_BODY		Registered hook callback will be called subsequently for large files. Buffer data over calls to get the entire body.
LSI_HKPT_SEND_RESP_HEADER		Send header.
LSI_HKPT_SEND_RESP_BODY	stream_write_next set_resp_content_length set_status_code end_resp flush	Send remaining response body. Set response info. Response body is finished.
LSI_HKPT_SENDFILE_RESP_BODY	send_file	Send response body that is in the form of a file.
LSI_HKPT_HTTP_END	free_module_data	Free allocated resources for module data, user-define data and configuration parameter data.
LSI_HKPT_L4_SENDING	stream_writev_next	Write the TCP stream data real-time. Set the flag_out flag of the callback parameters until the write is complete.
LSI_HKPT_L4_ENDSESSION		

Table 4: Callback Function Timeline

5.5. User Data

While processing a task, a module may have a need for user-defined data. For example, a module may want to store the state of processing or keep temporary variables for use by another step. User-defined data can be accessed within a module, among different modules, or in different processes. There are two types of user data in LSIAPI: configuration parameters and module data.

5.5.1. Resource Management

The user is responsible for allocating user data resources, and must define a deallocation callback function for each type. The user may also create environment variables. An environment variable's release is handled exclusively by the system.

The recommended method for allocating and releasing TCP and HTTP session level resources is to define callback functions at the begin and end session levels, so that the functions are always called at the appropriate time. Resources can be allocated and initialized at the beginning, and the release function is called at the end.

For TCP level global callback functions, use `LSI_HKPT_L4_BEGINSESSION` and `LSI_HKPT_L4_ENDSESSION`.

For HTTP session level callback functions, use `LSI_HKPT_HTTP_BEGIN` and `LSI_HKPT_HTTP_END`.

The details are below in 5.5.3. Module Data.

The server implementation automatically calls resource freeing functions for the global and configuration parameter data when the server application terminates.

5.5.2. Configuration Parameters

Parametric data is entered in the WebAdmin configuration when registering the module. It can then be accessed at any point within the executing module. Parameters can be set for each level of configuration (see the LSI_CFG_LEVEL enum): server, listener, virtual host and context level, including all branches of context.

When a module is loaded, the module configuration parameters are parsed by the server before any configuration callback is called. Only if there are one or more space separated title/value pairs for the module a configuration callback be called. If no configuration parameters are found, or if they are invalid, the callback will not be called.

At the time when the `lsi_module_t` instance is specified, the config_parser parameter, which is of type `lsi_confparser_t` must be specified as the fourth parameter.

parse_config() is a required callback function for the server to call to parse the user defined parameters and return a pointer to the user defined configuration data structure (which may be dynamically allocated or be static). params is an input parameter which is an array which hold the module_param_info_t and param_count holds the number of elements in the array. initial_config is an optional input pointer to the default configuration inherited from the parent level if any. level is an input parameter which is an integer derived from enum LSI_CFG_LEVEL. name is an input parameters of the name of the Server/Listener/VHost or URI of the Context, not used at the server level.

free_config() is an optional callback function for the server to call to release a pointer to the user defined data if not static. The only parameter is the config which would have been returned from the parse_config function.

config_keys is a statically defined array of lsi_config_key_t. Note that since it is an array, the last element of the array must consist of an element of { NULL, 0, 0 } to indicate the end.

As a simple example, if for a parse function which used a single parameter named "LSIPARAM1"

```
#define MOD_ABOUT "Module About"
#define MNAME     "modname"
lsi_config_key_t config_key[] =
{
        {"LSIPARAM1", 0, 0}, // Neither id or level used
        {NULL, 0, 0}   //The last position must have a NULL to indicate end of the array
};

static void *parse_config(module_param_info_t *param, int param_count,
                void *_initial_config, int level, const char *name);

lsi_confparser_t confparser = { parse_config, NULL, config_key }; // no free function
LSMODULE_EXPORT lsi_module_t MNAME = { LSI_MODULE_SIGNATURE,// Required value
                mod_init,       // Module initialization called at load time.
                &request_handler,               // Request handler data pointer.
                &confparser,                    // Config parser
                MOD_ABOUT,              // About
 NULL,                          // Server hooks
 {0} };
```

Again, only if configuration parameters are found at a given level and a **parse_config** callback specified in the lsi_module_t structure for the module will the callback be called. The callback is called even before the first function of that level. Thus for server level configuration parameters, this callback is called before the module initialization **init_pf** callback function is called.

The **config_parser**() callback is called for each level, with the parameter set that is defined in that level. The module processes this data to create a user-defined data structure for storage. During execution, the handler can then access the corresponding parameter data structure by calling the **get_config**() function or the **get_module_data()** function in that level.

If the user wants to "inherit" the parameter data from the previous level, the parse function must be written in a way that retains the data from the previous call. The calling order follows the system inheritance diagram in the section Web Server Architecture and Hierarchy.

To further continue the simple example, parse a single, globally defined parameter:

```
typedef struct {
   int m_param;
} mystuff_t;

mystuff_t mystuff = { 0 }; // Initialize it

// Parse each parameter
static int parse_list(module_param_info_t *param, mod_lsphp_config_t *config)
{
   int *pParam;
   ls_confparser_t confparser;

   ls_confparser(&confparser);
   ls_objarray_t *pList = ls_confparser_line(&confparser, param->val,
                                  param->val + param->val_len);
   int count = ls_objarray_getsize(pList);
   if (count == 0)
      return 0;

   unsigned long maxParamNum = param->key_index + 1;
   if (maxParamNum > 1) // Just 1 parameter
         maxParamNum = 1;

   ls_str_t *p;
```

```
    long val;
    int i;

    for (i = 0; i < count && i < maxParamNum; ++i)
    {
        p = (ls_str_t *)ls_objarray_getobj(pList, i);
        val = strtol(ls_str_cstr(p), NULL, 10);
        switch(param->key_index)
        {
            case 0:
                pParam = &config->m_slow_ms;
                break;
        }
        *pParam = val;
    }
    ls_confparser_d(&confparser);
    return 0;
}

static void *parse_config(module_param_info_t *param, int param_count,
                          void *_initial_config, int level, const char *name) {
    int i;
    mystuff_t *pInitConfig = (mystuff_t *)_initial_config;
    mystuff_t *pConfig = &mystuff; // global
    // Address inheritance
    if (pInitConfig)
        memcpy(pConfig, pInitConfig, sizeof(mod_lsphp_config_t));
    else
        memset(pConfig, 0, sizeof(mod_lsphp_config_t));

    if (!param)
        return (void *)pConfig;

    for (i=0; i<param_count; ++i)
        parse_list(&param[i], pConfig);
    return (void *)pConfig;
}
```

For an example module which does a more complex configuration, see the addon/example/testmoduledata.c example.

5.5.3. Module Data

Module Data is incredibly useful for associating data with an action at a specific level. In particular, most request handlers will use session data with the LSI_DATA_HTTP level. It will be initialized in the module handler and allocated by begin_module and freed when the session is freed (which may also be in begin_module).

Module data is user-defined data associated with a module at different levels:

- HTTP session level, valid within the HTTP session of the module.
- L4 session level, valid within the TCP L4 session of the module.
- IP level, shared amongst all connections with the same IP address.
- Virtual Host level, associated with a virtual host.
- File level, associated with a file in the file system.

The following functions are used to manage user data:

```
init_module_data()
set_module_data()
get_module_data()
free_module_data()
init_file_type_mdata()
```

The **init_module_data**() function initializes module data for the level (scope) specified by *level*, and must be called before **set_module_data**() or **get_module_data**() which set and get module data for a given *pSession*. The callback function specified in the init by *cb* is called by the server when the module data is destroyed, and should release allocated data.

The **get_cb_module_data()** function returns the module data specific to the callback at the specified level.

The **free_module_data**() function is used to free the module data immediately using callback function *cb*; it is not used by the server .

init_file_type_mdata() provides initialization before using FILE type module data for the file specified by *path* and *pathLen*. On success, the routine returns the file descriptor for accessing the file.

5.6. Environment Variable Handler

The user-defined environment variables and handlers provide a method of communication and control between modules. The variables are registered as environment variables with a

callback handler. When the variables are created/set or changed, the callback function is called. The handler can also be called directly by functions in other modules. The environment data is only available to the current HTTP session.

The **register_env_handler**() function registers the callback *cb* with the environment variable specified by *env_name*, so that calling **set_req_env**() for that environment variable causes the callback to be invoked.

The **set_req_env**() function sets or creates the environment variable specified by *name* with the value specified by *val* for the given *pSession*. In addition, if a callback is registered to this environment variable, it is called.

The **get_req_env**() function gets the pre-defined server or environment variable specified by *name*, and returns the variable value to the buffer at *val* whose size is specified by *maxValLen*. The function returns the length of the value string.

The **get_req_var_by_id**() function gets the pre-defined server or environment variable specified by *id*, and returns the variable value to the buffer at *val* whose size is specified by *maxValLen*. The function returns the length of the value string. The values for *id* are defined by `enum lsi_req_variable` in `ls.h`.

The example modules “testtimer.c” and “testmoduledata.c”, and the cache code, demonstrate the use of environment variable handling.

5.7. Shared Memory

The Shared Memory system available through `LSIAPI` provides a highly efficient method to access and manage data within and amongst LiteSpeed modules. The data is persistent across system reboot, being backed by *mmap* files. The sizes of the shared memory segments grow as needed so the user need not be concerned about wastefully over-allocating resources. Note, however, that as with any shared memory system, the developer must maintain the concept of *offsets* rather than *pointers* to data. An offset to data in shared memory will always remain constant, whereas a user pointer to the data may change if the shared memory segment is remapped by the system. `LSIAPI` provides routines to convert offsets to user space pointers, and these should be used.

There are two primary interfaces to LiteSpeed’s Shared Memory system:

- Shared Memory Pools
- Shared Memory Hash Tables

5.7.1. Shared Memory Pools

The Shared Memory Pool interface provides access to shared memory in blocks whose sizes are defined by the user. The system uses an efficient slab allocation memory management scheme to return undefined/uninitialized memory to the user who can define and manage it accordingly. The user identifies these blocks of memory to the system by offset and size.

The LSIAPI functions managing shared memory pools are:

```
shm_pool_init()
shm_pool_alloc()
shm_pool_free()
shm_pool_off2ptr()
```

The **shm_pool_init**() function initializes a shared memory pool with the name specified by *pName* and initial memory size in bytes *initSize*. If the pool does not exist, a new one is created. There may be situations when it is desirable to remove/delete the mmap file backing the shared memory pool to start *clean* (but this must be done knowing the consequences).

If *pName* is NULL, the system default name and size are used (*e.g.*, “LsShm”, and 1 page, 8K bytes). If *initSize* is zero, the system default size is used. The shared memory segments grow in size as needed.

Shm_pool_init() is generally the first routine called when using the shared memory pool system. On success, the routine returns a handle, *pShmpool*, to be used with all other shared memory pool functions. On error NULL is returned.

The **shm_pool_alloc**() function allocates a shared memory block of *size* bytes from the shared memory pool *pShmpool*, and returns the offset of the allocated memory. On error, 0 is returned.

The **shm_pool_free**() function frees/releases the shared memory block at offset *offset* of *size* bytes back to the shared memory pool *pShmpool*.

WARNING: When using **shm_pool_free**(), it is the responsibility of the user to ensure that *offset* and *size* are valid, that *offset* was returned from a previous call to **shm_pool_alloc**(), and that *size* is the same block size used in the allocation. Furthermore, **shm_pool_free**() must NOT be called again for an already freed block. Bad things will happen with invalid parameters.

Shm_pool_off2ptr() converts the shared memory pool offset *offset* in memory pool *pShmpool* to a user space pointer. On error, NULL is returned.

Example:

```
#include "ls.h"

        lsi_shmpool_t *pShmpool;
        lsi_shm_off_t dataOffset;

        {
           char *ptr;

           ...
           pShmpool = g_api->shm_pool_init( "SHMPool", 0 );
           if ( pShmpool == NULL )
              error;
           ...

           dataOffset = g_api->shm_pool_alloc( pShmpool, 16 );
           if ( dataOffset == 0 )
              error;
           ptr = (char *)g_api->shm_pool_off2ptr( pShmpool, dataOffset );
           if ( ptr == NULL )
              error;
           strcpy( ptr, "Hello World" );
           ...
        }

        {
           ...
           printf( "%s\n", (char *)g_api->shm_pool_off2ptr( pShmpool, dataOffset ) );
           g_api->shm_pool_free( pShmpool, dataOffset, 16 );
           ...
        }
```

5.7.2. Shared Memory Hash Tables

The Shared Memory Hash Table interface built on the Shared Memory Pool system provides an efficient method to implement an associative array in shared memory. At the base level, a key simply maps to a value. A number of user-defined parameters permit the developer to optimize the hashing for its specific purposes.

Once a hash table entry is set up in shared memory, the most efficient way to access the entry is through its offset (NOT pointer) which should be saved for subsequent use. Note, however, that it is then the responsibility of the user to ensure the allocated space for the value is not exceeded. If necessary, the entry should be deleted and a new larger one be created.

The LSIAPI functions managing shared memory hash tables are:

shm_htable_init()
shm_htable_add()
shm_htable_update()
shm_htable_set()
shm_htable_find()
shm_htable_get()
shm_htable_delete()
shm_htable_clear()
shm_htable_off2ptr()

The **shm_htable_init**() function initializes a shared memory hash table in memory pool *pShmpool*, with the hash name specified by *pName* and initial hash table index size (in entries/buckets) *initSize*. *Hash_pf* optionally specifies a function to be used for hash key generation. *Comp_pf* optionally specifies a function to be used for key comparison. If the hash table does not exist, a new one is created.

If *pShmpool* is NULL, a shared memory pool object with the name specified by *pName* is used. If *pName* is NULL, the system default name is used (*e.g.*, "LsHash"). If *initSize* is zero, the system default size is used. If *comp_pf* is NULL, the default compare function is used (strcmp(3)).

The functions **shm_htable_add**(), **shm_htable_update**(), and **shm_htable_set**() all attempt to define the hash table entry specified by *pKey* and *keyLen* with *pValue* and *valLen*. Each routine returns an offset to the entry value in the hash table on success and 0 on error. The routines differ in the following way:

- **shm_htable_add** - add a new entry; the key must NOT currently exist in the table.
- **shm_htable_update** - update/modify the entry value; the key MUST currently exist.
- **shm_htable_set** - set the entry value whether or not the key currently exists *i.e.*, either add a new entry or update an existing one.

The **shm_htable_find**() function finds the hash table entry specified by *pKey* and *keyLen* in hash table *pShmhash*, returning the offset to the entry value, and the length of the value through *pvalLen*.

Shm_htable_get() is similar to **shm_htable_find**() if the entry exists; but in addition, if the hash table entry does not exist, a new entry is created with size specified by the caller at *pvalLen*. If the flags specified by the caller at *pFlags* include LSI_SHM_INIT, the new entry value is initialized (cleared). The flags returned through *pFlags* specify that a new entry was created by setting LSI_SHM_CREATED.

The **shm_htable_delete**() function deletes/removes the hash table entry specified by *pKey* and *keyLen* in hash table *pShmhash*. The **shm_htable_clear**() function deletes all hash table entries for the hash table specified by *pShmhash*.

Shm_htable_off2ptr() converts the shared memory hash table offset *offset* in hash table *pShmhash* to a user space pointer. On error, NULL is returned.

Example:

```
#include "ls.h"

typedef struct {
   int m_type;
   int m_size;
   uint8_t m_ubuf[ UBUFSIZE ];
} mystuff_t;

lsi_shmpool_t *pShmpool;
lsi_shmhash_t *pShmhash;

int init_func()
{
   pShmpool = g_api->shm_pool_init( "SHMPool", 0x2000 );
   if ( pShmpool == NULL )
      error;
   pShmhash = g_api->shm_htable_init( pShmpool, "SHMHash", 0, NULL, NULL );
   if ( pShmhash == NULL )
      error;
   ...
}

int set_func( const char *key, int type, const void *pValue, int size,
  lsi_shm_off_t *pSavedOffset )
{
   mystuff_t valStuff;
   lsi_shm_off_t valOffset;

   valStuff.m_type = type;
   valStuff.m_size = size;
   memcpy( valStuff.m_ubuf, pValue, size );
   valOffset = g_api->shm_htable_set( pShmhash, (const uint8_t *)key, strlen(key),
```

```
    (const uint8_t *)&valStuff, sizeof(valStuff) );
  if ( valOffset == 0 )
     error;
  *pSavedOffset = valOffset;
  ...
}

mystuff_t * find_func( const char *key, lsi_shm_off_t *pSavedOffset )
{
  int valLen;
  lsi_shm_off_t valOffset;

  valOffset = g_api->shm_htable_find( pShmhash, (const uint8_t *)key, strlen(key),
    &valLen );
  if ( valOffset == 0 )
     error;
  *pSavedOffset = valOffset;
  ...
  return (mystuff_t *)g_api->shm_htable_off2ptr( pShmhash, valOffset );
}
```

Once an initial *set* or *find* has been successful for an entry, it would be most efficient to access the entry through its offset (NOT pointer) rather than searching again by key. The routines would be simpler and more efficient as follows.

```
int setagain_func( lsi_shm_off_t savedOffset, int type, const void *pValue, int size )
{
  mystuff_t *pStuff;

  pStuff = (mystuff_t *)g_api->shm_htable_off2ptr( pShmhash, savedOffset );
  pStuff->m_type = type;
  pStuff->m_size = size;
  memcpy( pStuff->m_ubuf, pValue, size );
  ...
}

mystuff_t * findagain_func( lsi_shm_off_t savedOffset )
{
  return (mystuff_t *)g_api->shm_htable_off2ptr( pShmhash, savedOffset );
}
```

5.7.2.1. Hash and Compare Functions

Options to **shm_htable_init**() allow the developer to specify functions to be used for hash key generation and key comparison. Note that *both* of these routines are significant in determining whether or not an entry key matches. Thus, for example, if a system was to be created being insensitive to the case of ascii characters in keys, *both* the hash and compare functions would have to reflect this, as below.

```
lsi_hash_key_t hash_func( const void *__s, int len )
{
   lsi_hash_key_t __h = 0;
   const char *p = (const char *)__s;
   char ch = *(const char*)p++;
   for ( ; ch ; ch = *((const char*)p++))
   {
      if (ch >= 'A' && ch <= 'Z')
         ch += 'a' - 'A';
      __h = __h * 31 + ( ch );
   }
   return __h;
}

int comp_func( const void *pVal1, const void *pVal2, int len )
{
return strncasecmp( (const char *)pVal1, (const char *)pVal2, len );
}

pShmhash = g_api->shm_htable_init( pShmpool, "SHMHash", 0, hash_func, comp_func );
```

6. EXAMPLE MODULE CREATION

The following example creates a handler which handles all URLs containing "/mytest", and replies with the string "MyTest!" to the browser.

6.1. Create File mytest.c

Below is a generic module template.

```
#include "ls.h"
#define MNAME mytest

lsi_module_t MNAME;

static int init_module( lsi_module_t *pModule )
{
   return 0;
}
static int beginProcess( lsi_session_t *pSession )
{
   return 0;
}
static int handlerOnRead( lsi_session_t *pSession )
{
   return 0;
}
static int handlerOnWrite( lsi_session_t *pSession )
{
   return 0;
}

lsi_reqhdlr_t myHandler =
{ beginProcess, handlerOnRead, handlerOnWrite, NULL, NULL, NULL, NULL };
LSMODULE_EXPORT lsi_module_t MNAME =
{ LSI_MODULE_SIGNATURE, init_module, &myHandler, NULL, "v0.0", NULL, {0} };
```

This may be used as a good starting point for new modules.

6.2. Add Handlers and Functionality

We modify the template code to use the begin process event handling callback, and to use a server hook to dynamically register the module as a request handler.

The user-defined handler callback function is set to handle the begin process event in the `lsi_reqhdlr_t` structure. That structure is then set in the `lsi_module_t` structure.

The `reg_handler()` filter is configured with the `lsi_serverhook_t` parameter in the `lsi_module_t` structure, which defines server callback functions. Its purpose is to catch any URLs with the string “/mytest” in it. If found, the filter function registers the event handler callback `beginProcess()` associated with handler `myHandler`.

```
#include "ls.h"
#include <string.h>
#define MNAME mytest
lsi_module_t MNAME;

static int reg_handler( lsi_param_t *pParam )
{
   const char *uri;
   int len;

   uri = g_api->get_req_uri( pParam->_session, &len );
   if ( len >= 7 && strncasecmp( uri, "/mytest", 7 ) == 0 )
   {
      g_api->register_req_handler( pParam->_session, &MNAME, 7 );
   }
   return LSI_RET_OK;
}

static int init_module( lsi_module_t *pModule )
{
   return 0;
}

static int beginProcess( lsi_session_t *pSession )
{
   g_api->append_resp_body( pSession, "MyTest!", 7 );
   g_api->end_resp( pSession );
   return 0;
}

static lsi_serverhook_t serverHooks[] =
```

```
{
  { LSI_HKPT_RECV_REQ_HEADER, reg_handler, LSI_HOOK_FIRST, LSI_FLAG_ENABLED },
    lsi_serverhook_t_END   // Must put this at the end
};

lsi_reqhdlr_t myHandler = { beginProcess, NULL, NULL, NULL, NULL, NULL, NULL };
LSMODULE_EXPORT lsi_module_t MNAME =
  { LSI_MODULE_SIGNATURE, init_module, &myHandler, NULL, "v1.0", serverHooks, {0} };
```

6.3. Build the Library and Test the Module

1. Use the `ccc.sh` script to create the `mytest.so` file, then the module file is ready.

2. Copy `mytest.so` to the `<ServerRoot>/modules/` directory (folder).
3. Add and enable the module name under the WebAdmin → Configurations → Server → Modules tab.
4. Restart the server.
5. Use `curl` to create a URL request as follows:

   ```
   curl -i http://localhost:8088/mytest
   ```

The result should be in the form header followed by “MyTest!” as the body:

HTTP/1.1 200 OK

Transfer-Encoding: chunked

Date: Thu, 20 Feb 2014 22:07:18 GMT

Server: LiteSpeed

MyTest!

7. SPECIAL TOPICS

7.1. Generating an Error Page

If a module detects an error condition, it may want to return an error message to the client. One method of doing this is through the return value of the handler's **begin_process**() callback routine. On success, **begin_process**() should return 0, but if it returns an HTTP status code (*e.g.*, 403 Forbidden, 404 Not Found, 405 Method Not Allowed), the corresponding HTTP error page is sent back to the client.

```
static int handlerBeginProcess( lsi_session_t *pSession )
{
   ...
   if ( error )
   {
      return HTTP_code;
   }
   ...
   g_api->end_resp( pSession );
   return 0;
}

static lsi_reqhdlr_t myhandler =
{ handlerBeginProcess, NULL, NULL, NULL, NULL, NULL, NULL };
```

Alternatively, the module can generate a custom error page by defining the error response body in the standard response manner, but additionally specifying the error condition with the LSIAPI function **set_status_code**(). In this case, the handler's callback routine must return 0, so that the server core processes the response normally, while the error code is conveyed through the status field set by the module.

```
static int handlerBeginProcess( lsi_session_t *pSession )
{
   ...
   if ( error )
   {
      g_api->append_resp_body( pSession, "My ERROR message\r\n", 18 );
      ...
      g_api->set_status_code( pSession, HTTP_code );
   }
   ...
```

```
    g_api->end_resp( pSession );
    return 0;
}
```

7.2. URI/URL Rewrite and Redirection

URI/URL rewrite and redirection are techniques used to make a resource accessible through alternate methods for a variety of reasons.

`LSIAPI` provides mechanisms to do this, primarily through its API function **set_uri_qs**(). Modifications to the URI can be combined with changes to the query string in this routine.

There are a number of places during the processing of an HTTP request that rewrite/redirection can occur.

7.2.1. URI Rewrite

First, a URL can be rewritten *before* a handler is selected. This is often used for URL shortening, as with messaging technologies or to expose a more meaningful or user-friendly URL. The *action* `LSI_URI_REWRITE` is used for this purpose. The following code segment rewrites a URI and appends a query string to an existing one:

```
static int check_uri( lsi_param_t *pParam )
{
    const char *uri;
    uri = g_api->get_req_uri( pParam->_session, NULL );
    if ( strcmp( "/friendly", uri ) == 0 )
    {
        g_api->set_uri_qs( pParam->_session,
          LSI_URI_REWRITE|LSI_URL_QS_APPEND,
          "/this_is_an_unfriendly/uri_specification.345", 44, "XYZ", 3 );
    }
    return LSI_RET_OK;
}

static lsi_serverhook_t serverHooks[] =
{
  { LSI_HKPT_RECV_REQ_HEADER, check_uri, LSI_HOOK_NORMAL-1, LSI_FLAG_ENABLED },
    lsi_serverhook_t_END   // Must put this at the end
};
```

7.2.2. Internal Redirection

There are situations after a handler has been selected when redirection internally is desired. An example of this might be authentication checking by a handler, followed by redirection to a static file. In this case, `LSI_URL_REDIRECT_INTERNAL` is the *action* specified:

```
static int handlerBeginProcess( lsi_session_t *pSession )
{
   ...
   g_api->set_uri_qs( pSession,
     LSI_URL_REDIRECT_INTERNAL|LSI_URL_QS_DELETE, "/index.html", 11, NULL, 0 );
   ...
}
```

7.2.3. External Redirection

Lastly, external redirection is the case where the server should redirect the client to a different URL in the response. In this case, the response header includes an HTTP status code (*e.g.*, 301 Moved Permanently, 302 Found, 307 Temporary Redirect), and the Location header field specifies the new information. For example:

```
static int handlerBeginProcess( lsi_session_t *pSession )
{
   ...
   g_api->set_uri_qs( pSession,
     LSI_URL_REDIRECT_307|LSI_URL_QS_SET, "/new_location", 13, "ABC", 3 );
   ...
}
```

would result in a response header similar to:

HTTP/1.1 307 Temporary Redirect
...
Server: LiteSpeed
Location: http://localhost:8088/new_location?ABC
...

8. TROUBLESHOOTING

The WebAdmin Console has tools to turn on logging and debug information. These can be found under the Configuration → Server → Log tab. The log files, debug, and log levels can be set.

8.1. Common Problems

8.1.1. A registered module causes the server to crash.

When the server crashes, the WebAdmin Console will not launch so it cannot be used to remove the offending module. Instead, rename or remove the module file in the modules directory. Once the .so file is moved or renamed, the server will treat the module as missing ane will run without loading it.

8.1.2. A modules fails to function after upgrading the server.

When the server is upgraded, there may be changes to the API requiring a rebuild of the modules.

8.1.3. A module is not found when trying to register it.

Confirm the following items:

- The module name matches the MNAME value (*i.e.*, `lsi_module_t` variable)
- The module name matches the name of the `.so` loadable library.
- The module .so file is in the correct modules directory.
- If the module has an init function, it returns LS_OK.

8.1.4. Confirming a module has loaded.

View the error.log file when NOTICE logging is enabled, the module should be listed if it loaded correctly.

9. APPENDIX

9.1. Examples Included in Distribution and Output

Descriptions and output of the some of the enclosed examples. Please refer to the example source for implementation details and additional example modules.

9.1.1. hellohandler

Handler with NULL `init_pf` pointer, that needs to be registered with the configuration file through WebAdmin. In OpenLiteSpeed, add the module to the server, then add a script handler with a suffix of “123” in the Example vhost. The corresponding configuration file vhost.conf will contain the following:

```
scripthandler  {
  add                     module:hellohandler 123
}
```

Command line executions using curl for several example strings and the output:

> touch $LSWS_HOME/Example/html/1.123

```
> curl -i_http://localhost:8088/1.123
```

HTTP/1.1 200 OK
Transfer-Encoding: chunked
Date: Tue, 27 Aug 2013 15:15:21 GMT
Accept-Ranges: bytes
Server: LiteSpeed

Hello module handler.
>

```
> curl -i_http://localhost:8088/11.123
```

HTTP/1.1 404 Not Found
Date: Tue, 27 Aug 2013 15:20:38 GMT
Accept-Ranges: bytes
Server: LiteSpeed
ETag: "c3-5204fcb6-40e32f"
Last-Modified: Fri, 09 Aug 2013 14:29:10 GMT

Content-Type: text/html
Content-Length: 195
<HTML>
<HEAD>
<TITLE>Page Not Found</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<HR>
<H1>Request Page Not Found</H1>
This is a customized error page for missing pages.
<HR>
</BODY>
</HTML>
>

With script handlers, the file is checked before forwarding the request to the handler, so a 404 error response is returned if the file 1.123 does not exist.

9.1.2. hellohandler2

Functionally, hellohandler2 is similar like hellohandler, except the suffix specified is ".345", and the handler is registered dynamically when the suffix is detected. The handler is then called on the `begin_process` event.

Below is the execution of hellohandler2. Since the handler is registered dynamically, the existence of the file does not matter. The purpose of these two examples is to demonstrate the difference of these two registering methods.

```
> curl -i http://192.168.0.238:8088/hi.345
```

HTTP/1.1 200 OK
Transfer-Encoding: chunked
Date: Wed, 23 Oct 2013 21:02:30 GMT
Accept-Ranges: bytes
Server: LiteSpeed

Hello module handler2.

9.1.3. logreqhandler

A handler where the `init()` is again NULL, that has to be registered with the configuration file using WebAdmin using the suffix "234".

In this case, we send response strings to the browser, and write logs to errorlog.
This is the testing result.

```
> curl -i http://localhost:8088/1.234
HTTP/1.1 200 OK
SENDBODY: ......00000000....
Transfer-Encoding: chunked
Date: Tue, 27 Aug 2013 15:37:42 GMT
Accept-Ranges: bytes
Server: LiteSpeed

Hi, My 1st module
The 2nd line
123
123 523---1235
The last lineThe last line
>
```

9.1.4. setrespheader

This is an example of how to add a hook filter to append a string to a response header.

9.1.5. reqinfhandler

An example of querying all of the request environment and server environment variables. It also provides a handler to process user request body data differently when URIs are different. It dynamically registers the handlers and shows how to access the HTTP level user data.

In this case, if the first 8 letters of the URL is "/reqinfo", then the request will be set to be handled by this handler, and in the `begin_process` function, the module will add the information to the response body.

Then, if the next letters of the URL are "/echo", the request body will be send back as the response body. If the next letters of the URL are "/md5", the md5 of the request body will be sent back as the response body. If the next letters of the URL are "/upload", the request body will be saved to file /tmp/uploadfile on the server.

Use the file reqform.html for this testing.

9.1.6. testmoduledata

This test module demonstrates the use of module data sharing. It replies with the number of times that a particular IP accesses the server, the page being accessed, and how many times this file has been accessed. If the test URI is /testmoduledata/file1, and if /file1 exists in the testing vhost directory, then the URI will be handled by the module.

9.1.7. updatehttpout

This example demonstrates setting callback functions on `LSI_HKPT_HTTP_BEGIN`, `LSI_HKPT_RECV_RESP_BODY` and `LSI_HKPT_RECV_REQ_HEADER`.

The filter callback function `httprespwrite` adds a space to each response body letter and the callback function `httpreqHeaderRecved` prints the request information to the error log.

9.1.8. updatetcpin and updatetcpout

These are TCP level filter examples. The filter function l4recv parses the received data, and on finding "\r\n\r\n", it unzips the remaining data and sends it to the next step so that only the unzipped data is visible after the filter processing. The filter function l4send translates all HEX view type letters to binary view type. The filter are complements - using both filters in sequence will set the response back to the original data.

9.1.9. sendfilehandler

An example to use the api **send_file**() to send a file in a handler.

9.1.10. mthello

An example multi-thread module ala hellohandler

9.1.11. mtaltreadwrite

An example multi-thread module that performs alternate reads and writes per provided byte/character counts