HTTPS Secure Server User Guide

Version 1.50

For use with HTTPS Secure Server module versions 3.11 and above

Date: 20-Jun-2017 14:33

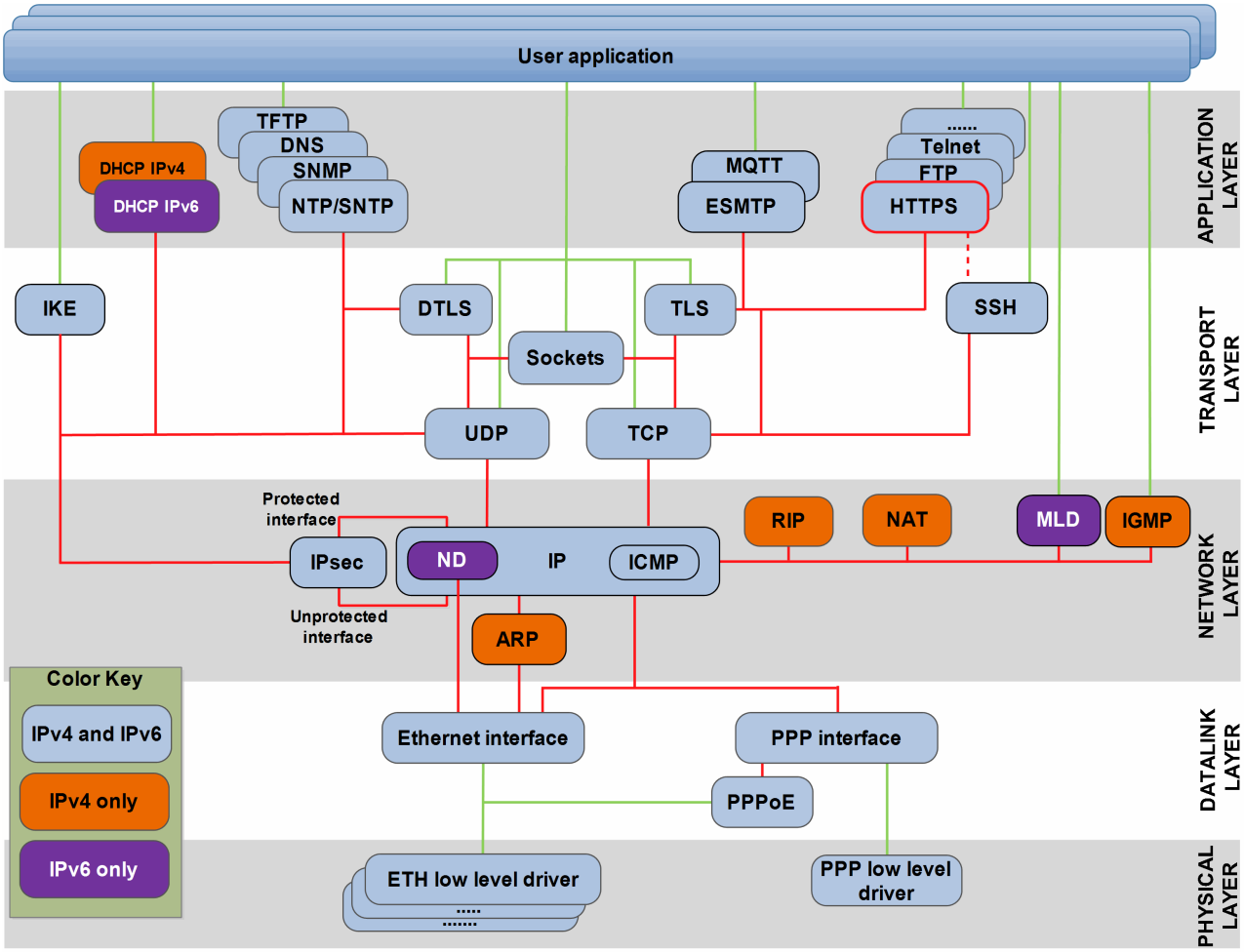
Table of Contents

1 System Overview

1.1 Introduction

This guide is for those who want to implement HTTPS Secure Server as part of HCC Embedded's MISRA-compliant TCP/IP stack.

The HTTPS Secure Server module works with both IPv4 and IPv6 networks. Its place in the HCC TCP/IP stack is shown below. (In this diagram green lines show interfaces available to users of the stack, red lines show interfaces internal to the TCP/IP system.)



The Hypertext Transfer Protocol Secure (HTTPS) provides secure communication over computer networks. HTTPS resources are identified and located on the network using Uniform Resource Identifiers (URIs). This module provides HTTP's secure version when you enable this, but to use HTTPS your system must include HCC's Transport Layer Security (TLS) module. It is TLS that provides the security by encrypting the whole HTTPS message, including the header and the request/response content. Where this manual refers to HTTP, the reference applies equally to HTTPS.

HTTPS operates as a request-response protocol in the client/server model. The secure client may be a web browser, for example, while an application running on a computer hosting a website may be the secure server.

The secure client sends an HTTPS request message to the secure server. The server, which may provide resources such as HTML files to the client, or perform other functions for it, sends a response message back to the client. This contains completion status information about the request. It may also contain requested content within the message body. The entire HTTPS message is encrypted, including the headers and the content.

1.2 Methods

The secure server module provides the following methods:

Method	Description
GET	Retrieves the information identified by the Request-URI.
HEAD	This is identical to GET except that the server returns only the metadata held in the HTTP headers. It does not return the message body itself. This is useful for obtaining metadata about an entity without transferring the body of the entity itself.
POST	Requests that the origin server accept the specified entity as a new subordinate of the resource identified by the Request-URI in the Request-Line. This method is used, for example, to post to a forum.
PUT	Requests that the enclosed entity be stored under the supplied Request-URI.
DELETE	Requests that the server delete the resource identified by the Request-URI.

These are the basis for the examples shown in Code Examples.

1.3 Feature Check

The main features of the system are the following:

- Conforms to the HCC Advanced Embedded Framework.
- Compliant with HCC's MISRA-compliant TCP/IP stack.
- Works with both IPv4 and IPv6 networks.
- Compliant with RFC 2818.
- Designed for integration with both RTOS and non-RTOS based systems.
- Can be configured to use BSD sockets.
- Supports all standard HTTP methods: GET, PUT, POST, and DELETE.
- Supports HTTP Secure (HTTPS) connections
- Handles a configurable number of simultaneous connections.
- Handles static ROMed pages.
- Can be connected to any file system and process pages received from it.
- Pages may contain dynamic content that can be created by user-specified functions.
- Supports dynamic variables from tags in HTML.
- Supports optional user authentication based on user name and IP address (as a sample).
- Demo package provides sample implementations to base your HTTP server on.
- It is easy to add features like the following: web on SD card, access filtering, multi-user web authentication, and multi-language web pages.

1.4 Packages and Documents

Packages

The table below lists the packages that you need in order to use this module:

Package	Description
hcc_base_doc	This contains the two guides that will help you get started.
ip_app_http	The HTTPS Secure Server package described in this manual.
ip_app_http_demo	The demo package of sample implementations to base your secure server on.
mip_base	The TCP/IP Dual Stack base package.
ip_base_v4, ip_base_v6	The TCP/IP stack base packages for IPv4 and IPv6, respectively.
ip_tls	The Transport Layer Security (TLS) and DTLS package, needed if a secure connection is used.

Documents

For an overview of the HCC TCP/IP stack software, see Product Information on the main HCC website.

Readers should note the points in the HCC Documentation Guidelines on the HCC documentation website.

HCC Firmware Quick Start Guide

This document describes how to install packages provided by HCC in the target development environment. Also follow the *Quick Start Guide* when HCC provides package updates.

HCC Source Tree Guide

This document describes the HCC source tree. It gives an overview of the system to make clear the logic behind its organization.

HCC TCP/IP Dual Stack System User Guide

This is the core document that describes the complete TCP/IP stack. It covers both IPv4 and IPv6 systems.

HCC HTTPS Secure Server User Guide

This is this document.

1.5 Change History

This section describes past changes to this manual.

- To download earlier manuals, see Archive: HTTPS Secure Server User Guide.
- For the history of changes made to the package code itself, see History: ip_app_http.

The current version of this manual is 1.50. The full list of versions is as follows

Manual version	Date	Software version	Reason for change
1.50	2017-06-20	3.11	New *Change History* format.
1.40	2017-03-28	3.11	Updated network diagram.
1.30	2017-01-16	3.10	Updated network diagram. Added function group tables.
1.20	2016-03-29	3.08	Reorganized *System Overview* section.
1.10	2016-03-11	3.07	First online version.

2 Source File List

The following sections describe all the source code files included in the system. These files follow the HCC Embedded standard source tree system, described in the *HCC Source Tree Guide*. All references to file pathnames refer to locations within this standard source tree, not within the package you initially receive.

> **Note:** Do not modify any files except the configuration files.

2.1 API Header File

The file **src/api/api_ip_app_http.h** is the only file that should be included by an application using this module. For details of these API functions, see Application Programming Interface.

2.2 Configuration Files

These files in **src/config** contain all the configurable HTTPS Secure Server parameters. Configure these as required. For details of these options, see Configuration Options.

File	Description
config_ip_app_http.c	Implements content type strings.
config_ip_app_http.h	Contains the standard and dynamic configuration options.

2.3 System File

The system file is **src/ip/apps/http/http.c. This file should only be modified by HCC**.

2.4 Version File

The file **src/version/ver_ip_app_http.h** contains the version number of this module. This version number is checked by all modules that use this module to ensure system consistency over upgrades.

3 Configuration Options

Set the standard and dynamic system configuration options in the file **src/config/config_ip_app_http.h**. The file **src/config/config_ip_app_http.c** holds the content type strings, which you can modify as required.

3.1 Standard Options

This section lists the non-dynamic configuration options and their default values.

HTTP_SERVER_TASK_STACK_SIZE

The HTTPS Secure Server task stack size. The default value is 512. This task accepts connections and then passes the process to the connection task. This is required to allow the connection to be accepted as fast as possible.

HTTP_CONN_TASK_STACK_SIZE

The HTTPS connection task stack size. The default value is 2048. All request open, write, close and response open, read, and close functions are called from this task, so set the stack size accordingly.

HTTP_SRV_PORT

The default server port. The default value is 80.

HTTP_MAX_CONNECTIONS

The maximum number of connections. The default value is 4.

HTTP_MAX_TRANSFER_UNIT

The maximum transfer size a connection can send at a time. The default value is 2048. Depending on the size of the assigned buffers, this can be slightly more.

HTTP_IDLE_TIMEOUT

The idle timeout. This is the time in seconds after which the connection is closed if there is no communication on it. The default value is 10.

For secure connections we recommend increasing this idle timeout value because establishing TLS connections is a long process.

> **Note:** For both these timeouts the precision is 500ms, so timeout may occur 500ms earlier.

HTTP_KEEPALIVE_TIMEOUT

The keep-alive timeout in seconds, the length of time the connection remains open. Set this to 0 to disable keep-alive handling. The default value is 2.

HTTP_MAX_REQUEST_BUF_SIZE

The maximum file request buffer size. The default value is 128. The request buffer holds one of the following items:

- For the GET method, the request URI, the optional URL-encoded INPUT string.
- In the case of a multipart/form-data request content type, the boundary string.

HTTP_MAIN_PAGE_FILE_NAME

The main page access path and file name. The default value is "/index.html".

HTTP_HEADER_VER

The default header HTML version. The default value is "HTTP/1.1". This option defines one of the default parameters returned in the HEAD.

HTTP_HEADER_SERVER

The default header server name. The default value is "HCCWeb". This option defines the other default parameter returned in the HEAD.

HTTP_DYNAMIC_LENGTH

Keep this at the default value of 1 if dynamic length is allowed. This is useful if the user file needs to generate dynamic content, or when dynamic variables are enabled.

HTTP_SECURE_ENABLE

Set this to enable HTTP Secure connections. The default value is 0.

> **Note:** The **HTTP_SECURE_ENABLE** option must be enabled for dynamic variables.

HTTP_SRV_PORT_SECURE

The default HTTPS server port. The default value is 443.

HTTP_SEC_TRANSFER_UNIT

The HTTP secure connection transfer unit for socket connection. The default value is 1200.

> **Note:** This must be less than the maximum amount of data that can be transferred in a single secure PDU by TLS.

3.2 Dynamic Options

This section lists the dynamic configuration options and their default values.

HTTP_DYNAMIC_VARIABLES

Set this to enable use of dynamic variables. The default value is 0.

> **Note:** The HTTP_DYNAMIC_LENGTH and HTTP_SECURE_ENABLE options described above must be enabled if dynamic variables are to be used.

HTTP_DYN_VAR_HDLR_COUNT

The maximum number of concurrent dynamic variable handlers. The default value is 2.

HTTP_DYN_VAR_TAG_NAME

The dynamic variable tag name. The default value is "<var:".

For example, if tag name is "var:" the format will be: <var:name>. The length of this tag must always be less than HTTP_DYN_VAR_READ_BUF_SIZE.

HTTP_DYN_VAR_ERROR_MSG

The message copied if dynamic variable resolution fails. The default value is "ERROR".

HTTP_DYN_VAR_MAX_LENGTH

The maximum length of a resolved variable. The default value is 16.

HTTP_DYN_VAR_READ_BUF_SIZE

The read ahead buffer size. The default value is 256.

HTTP_USE_SOCKET

Keep the default of 0 to use the native HCC implementation. Set it to 1 to use the BSD socket implementation.

HTTP_SOCKET_IP_V6_ENABLE

Keep the default of 0 to use only IPv4 connections. Set it to 1 to enable IPv6 for the socket.

HTTP_USE_STD_SOCKET

Keep the default of 0 to use HCC's Socket implementation. Set it to 1 to use the standard Socket implementation.

HTTP_SOCKET_NO_LINGER

Keep the default of 0 to enable a linger time to be set. **This is not supported by LwIP.**

HTTP_SOCKET_SO_SNDTIMEO

Set this to 1 to enable a send timeout to be set.

> **Note:** Disable this when using LwIP, because it handles the SO_SNDTIMEO option incorrectly for infinite timeout.

3.3 config_ip_app_http.c

The file **src/config/config_ip_app_http.c** contains an array of content type strings, part of which is shown below. You may modify these if required.

```
const t_http_media_type  g_http_media_type[] =
{
    { "text/html", "htm"
#if ( HTTP_DYNAMIC_VARIABLES != 0 )
    , FALSE
#endif
    }
  , { "text/html", "html"
#if ( HTTP_DYNAMIC_VARIABLES != 0 )
      , FALSE
#endif
    }
  , { "text/css", "css"
#if ( HTTP_DYNAMIC_VARIABLES != 0 )
      , FALSE
#endif
    }
  ..
  ..
  ..
}
```

4 Application Programming Interface

This section documents the Application Programming Interface (API). It includes all the functions that are available to an application program.

4.1 Module Management

The functions are the following:

Function	Description
http_init()	Initializes the module and allocates the required resources.
http_start()	Starts the module.
http_stop()	Stops the module.
http_delete()	Deletes the module and releases the resources it used.

http_init

Use this function to initialize the HTTPS Secure Server module and allocate the required resources.

> **Note:** Call this before any other function.

Format

```
t_http_ret http_init ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
HTTP_SUCCESS	Successful execution.
HTTP_ERROR	See Error Codes.

http_start

Use this function to start the HTTPS Secure Server module.

> **Note:** Call **http_init()** before this function.

Format

```
t_http_ret http_start ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
HTTP_SUCCESS	Successful execution.
HTTP_ERROR	See Error Codes.

http_stop

Use this function to stop the HTTPS Secure Server module. This closes the server TCP port.

Format

```
t_http_ret http_stop ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
HTTP_SUCCESS	Successful execution.
HTTP_ERROR	See Error Codes.

http_delete

Use this function to delete the HTTPS Secure Server module and release the associated resources.

Format

```
t_http_ret http_delete ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
HTTP_SUCCESS	Successful execution.
HTTP_ERROR	See Error Codes.

4.2 Server Management

The functions are the following.

Function	Description
http_get_content_type()	Searches for the extension of a given content name in the media type extension table.
http_get_urlenc_value()	Searches for the value of a variable name in a URL-encoded buffer.
http_register_cb()	Registers callback functions.

http_get_content_type

Use this function to search for the extension of a given content name in the media type extension table.

Format

```
t_http_ret http_get_content_type (
    const char_t                        name[],
    const t_http_media_type * * const   pp_ctype )
```

Arguments

Name	Description	Type
name[]	The content name.	char_t
pp_ctype	Where to write the pointer to the media type.	t_http_media_type * *

Return Values

Code	Description
HTTP_SUCCESS	Successful execution.
HTTP_ERROR	The media type was not found.

http_get_urlenc_value

Use this function to search for the value of a variable name in a URL-encoded buffer.

Format

```
t_http_ret http_get_urlenc_value (
    const char_t * const   p_buf,
    const uint16_t         buf_len,
    const char_t * const   p_name,
    char_t                 value[],
    const uint16_t         value_len )
```

Arguments

Name	Description	Type
p_buf	The buffer containing the URL encoded string.	char_t *
buf_len	The length of the buffer.	uint16_t
p_name	A pointer to the name of the variable to search for.	char_t *
value[]	Where to write the value of the requested name.	char_t
value_len	The length of the value buffer.	uint16_t

Return Values

Code	Description
HTTP_SUCCESS	Successful execution.
HTTP_ERR_NOT_FOUND	The value was not found.
HTTP_ERR_LENGTH	The value was found but the *value_len* is short.
HTTP_ERROR	Error in the URL encoded string.

http_register_cb

Use this function to register callback functions.

Format

```
t_http_ret http_register_cb ( const t_http_cb_dsc * const p_cb_dsc )
```

Arguments

Name	Description	Type
p_cb_dsc	A pointer to the callback functions descriptor.	t_http_cb_dsc *

Return Values

Code	Description
HTTP_SUCCESS	Successful execution.
HTTP_ERROR	Operation failed.

4.3 Callback Functions

These functions are the interface between the HTTPS Secure Server and your file system. You can implement these as required.

> **Note:** All callback functions are called from the same HTTPS Secure Server task context and are protected against concurrent calls. That is, no callback function can interrupt another callback function.

The callbacks are the following.

Function	Description
t_http_req_open_cb()	Specifies the format of the callback function called when a request is received.
t_http_req_write_cb()	Specifies the format of the callback function called to pass data received in the request (in the PUT, POST, or DELETE cases).
t_http_req_close_cb()	Specifies the format of the callback function called to close a file or to free any other resource.
t_http_resp_open_cb()	Specifies the format of the callback function called to initialize the response phase.
t_http_resp_read_cb()	Specifies the format of the callback function called to get the response data.
t_http_resp_close_cb()	Specifies the format of the callback function called when all response data has been sent. This can be used to close a file or to release any other resource.
t_http_dynvar_cb()	Specifies the format of the callback function that may be called to handle dynamic variables.

t_http_req_open_cb

The **t_http_req_open_cb** definition specifies the format of the callback function that is called when a request is received.

The user implementation (based on the request method) can decide whether a resource needs to be allocated (for example, a file in the PUT, POST or DELETE cases). If the resource it tries to allocate is not currently available (for example, because there is no free file handle), it can signal that a later retry is required. (Retry is always executed after a response close callback.)

Format

```
typedef t_http_ret ( * t_http_req_open_cb ) (
    const t_http_req_info * const   p_req_info,
    uint32_t * const                p_hdl )
```

Arguments

Parameter	Description	Type
p_req_info	A pointer to the HTTP request information.	t_http_req_info *
p_hdl	Where to write the internal handle.	uint32_t *

Return Codes

Code	Description
HTTP_SUCCESS	Successful execution.
HTTP_RETRY	The server must retry later.
HTTP_ERROR	Operation failed.

t_http_req_write_cb

The **t_http_req_write_cb** definition specifies the format of the callback function that is called to pass data received in the request (in the PUT, POST, or DELETE cases).

The implementation can decide how to process the data (for example, to store it in a structure or a file).

> **Note:** This function is always called from the same task context as **t_http_req_open_cb()**.

Format

```
typedef t_http_ret ( * t_http_req_write_cb ) (
    const t_http_req_info * const   p_req_info,
    const uint32_t                  hdl,
    const uint8_t * const           p_buf,
    const uint16_t                  buf_len )
```

Arguments

Parameter	Description	Type
p_req_info	A pointer to the HTTP request information.	t_http_req_info *
hdl	The internal handle. This is returned by the callback **t_http_req_open_cb()**.	uint32_t
p_buf	Where to write the data received.	uint8_t *
buf_len	The length of the data received.	uint16_t

Return Codes

Code	Description
HTTP_SUCCESS	Successful execution.
HTTP_ERROR	Operation failed.

t_http_req_close_cb

The **t_http_req_close_cb** definition specifies the format of the callback function that is called in two situations:

- when all data has been received (in the PUT, POST or DELETE cases).
- immediately after the **t_http_req_open_cb()** callback (in the GET case).

This callback can be used to close a file or to free any other resource.

The response HEAD is sent after this function returns, so it must output the response information (content type and length). The content length can be set to HTTP_LENGTH_UNKNOWN if the content length is not known (for example, when the response is generated dynamically) and chunked transfer needs to be used. (You must set HTTP_DYNAMIC_LENGTH in **config_ip_app_http.h** for this to work).

> **Note:** This function is always called from the same task context as **t_http_req_open_cb()**.

Format

```
typedef t_http_ret ( * t_http_req_close_cb ) (
    const t_http_req_info * const   p_req_info,
    const uint32_t                  hdl,
    t_http_resp_info * const        p_resp_info )
```

Arguments

Parameter	Description	Type
p_req_info	A pointer to the HTTP request information.	t_http_req_info *
p_hdl	The internal handle. This is returned by the callback **t_http_req_open_cb()**.	uint32_t *
p_resp_info	Where to write the response information.	t_http_resp_info *

Return Codes

Code	Description
HTTP_SUCCESS	Successful execution.
HTTP_ERR_NOT_FOUND	No response entity was found.
HTTP_ERROR	Operation failed.

t_http_resp_open_cb

The **t_http_resp_open_cb** definition specifies the format of the callback function that is called to initialize the response phase.

The user implementation can decide whether a resource needs to be allocated. If it tries to do this and the resource is not currently available (for example, because there is no free file handle), it can signal that a later retry is needed. (The retry is always executed after a **t_http_resp_close_cb()** callback.) Alternatively, you can assign a new internal handle for use by the **t_http_resp_read_cb()** and **t_http_resp_close_cb()** callbacks.

Note: This function is always called from the same task context as **t_http_req_open_cb()**.

Format

```
typedef t_http_ret ( * t_http_resp_open_cb ) (
    const t_http_req_info * const   p_req_info,
    const uint32_t                  hdl,
    uint32_t * const                p_new_hdl )
```

Arguments

Parameter	Description	Type
p_req_info	A pointer to the HTTP request information.	t_http_req_info *
hdl	The internal handle. This is returned by the callback **t_http_req_open_cb()**.	uint32_t
p_new_hdl	Where to write the new handle.	uint32_t *

Return Codes

Code	Description
HTTP_SUCCESS	Successful execution.
HTTP_RETRY	The server must retry later.
HTTP_ERROR	Operation failed.

t_http_resp_read_cb

The **t_http_resp_read_cb** definition specifies the format of the callback function that is called to get the response data.

The implementation can decide how to process the data (for example, read it from a file or generate content dynamically).

> **Note:** This function is always called from the same task context as **t_http_req_open_cb()**.

Format

```
t_http_ret http_resp_read_cb (
    const t_http_req_info * const   p_req_info,
    const uint32_t                  hdl,
    uint8_t * const                 p_buf,
    const uint16_t                  buf_len,
    uint16_t * const                p_rd_len )
```

Arguments

Parameter	Description	Type
p_req_info	A pointer to the HTTP request information.	t_http_req_info *
hdl	The internal handle. This is returned by the callback **t_http_resp_open_cb()**.	uint32_t
p_buf	Where to write the response data.	uint8_t *
buf_len	The length of the output buffer.	uint16_t
p_rd_len	Where to write the length of the response data.	uint16_t *

Return Codes

Code	Description
HTTP_SUCCESS	Successful execution.
HTTP_ERROR	Operation failed.

t_http_resp_close_cb

The **t_http_resp_close_cb** definition specifies the format of the callback function that is called when all response data has been sent. This can be used to close a file or to release any other resource.

> **Note:** This function is always called from the same task context as **t_http_req_open_cb()**.

Format

```
typedef t_http_ret ( * t_http_resp_close_cb ) (
    const t_http_req_info * const   p_req_info,
    const uint32_t                  hdl )
```

Arguments

Parameter	Description	Type
p_req_info	A pointer to the HTTP request information.	t_http_req_info *
hdl	The internal handle. This is returned by the callback **t_http_resp_open_cb()**.	uint32_t

Return Codes

Code	Description
HTTP_SUCCESS	Successful execution.
HTTP_ERROR	Operation failed.

t_http_dynvar_cb

The **t_http_dynvar_cb** definition specifies the format of the callback function that may be called to handle dynamic variables.

Format

```
typedef t_http_ret ( * t_http_dynvar_cb ) (
    const t_http_req_info * const   p_req_info,
    const char_t * const            p_var_name,
    char_t * const                  p_var_value )
```

Arguments

Parameter	Description	Type
p_req_info	A pointer to the HTTP request information.	t_http_req_info *
p_var_name	A pointer to the name of the dynamic variable.	char_t *
p_var_value	On return, where to write the value of the resolved dynamic variable. The maximum length of the output string is HTTP_DYN_VAR_MAX_LENGTH (excluding the 0 terminator).	char_t *

Return Codes

Code	Description
HTTP_SUCCESS	Successful execution.
HTTP_ERROR	Operation failed.

4.4 Error Codes

If a function executes successfully, it returns with HTTP_SUCCESS. The following table shows the meaning of the HTTPS Secure Server error codes.

> **Note:** Check other error code values in the base system by using the *HCC TCP/IP v4 Stack System User Guide*.

Return Value	Value	Description
HTTP_SUCCESS	0U	Successful execution.
HTTP_RETRY	1U	Retry later.
HTTP_ERR_NOT_FOUND	2U	Not found.
HTTP_ERR_LENGTH	3U	Length error.
HTTP_ERROR	4U	General error.
HTTP_LENGTH_UNKNOWN	0xFFFFFFFFU	Length not known.

4.5 Types and Definitions

t_http_cb_dsc

The *t_http_cb_dsc* structure is the callback functions descriptor.

Element	Type	Description
t_http_req_open_cb	hcd_req_open_cb	The request open callback.
t_http_req_write_cb	hcd_req_write_cb	The request write callback.
t_http_req_close_cb	hcd_req_close_cb	The request close callback.
t_http_resp_open_cb	hcd_resp_open_cb	The response open callback.
t_http_resp_read_cb	hcd_resp_read_cb	The response read callback.
t_http_resp_close_cb	hcd_resp_close_cb	The response close callback.
t_http_dynvar_cb	hcd_dynvar_cb	The dynamic variable callback. This only applies if HTTP_DYNAMIC_VARIABLES is set (non-zero).

t_http_media_type

The *t_http_media_type* structure holds the media type features. It takes this form:

Element	Type	Description
p_hme_type	char_t *	The content type.
p_hme_ext	char_t *	The file name extension.
b_hme_dyn_var	uint8_t	This only applies if HTTP_DYNAMIC_VARIABLES is set (non-zero). Set this TRUE if you need to resolve dynamic variables.

t_http_req_info

The *t_http_req_info* structure holds the HTTPS Secure Server request information. It takes this form:

Element	Type	Description
hreq_ip_port	t_ip_port	The remote IP port.
hreq_method	t_http_method	The request method (HTTP_METHOD_xxx).
p_hreq_uri	char_t *	A pointer to the URI.
p_hreq_uri_input	char_t *	A pointer to the URI input.
hreq_uri_input_len	uint16_t	The URI input length.
hreq_ctype	t_http_req_ctype	The request content type (only used with the PUT, POST, and DELETE methods).
hreq_clen	uint32_t	The request content length (only used with the PUT, POST, and DELETE methods). This can be set to HTTP_LENGTH_UNKNOWN if the length is not known (for example, when the response is generated dynamically) and chunked transfer needs to be used; see t_http_req_close_cb() for details.
p_hreq_boundary	char_t *	A pointer to the boundary (in the case of multipart or form data).

t_http_method

The *t_http_method* typedef holds the following methods.

Method	Description
HTTP_METHOD_INVALID	Invalid method.
HTTP_METHOD_GET	Retrieves the data that is identified by the URI.
HTTP_METHOD_HEAD	This method is similar to GET but returns just HTTP headers, not the document body.
HTTP_METHOD_PUT	Specifies that the data in the body section must be stored under the supplied URL, which must already exist.
HTTP_METHOD_POST	Creates a new object linked to the specified object.
HTTP_METHOD_DELETE	Asks the server to delete the information corresponding to the given URL.

5 Integration

This section describes all aspects of the module that require integration with your target project. This includes porting and configuration of external resources.

5.1 OS Abstraction Layer

All HCC modules use the OS Abstraction Layer (OAL) that allows the module to run seamlessly with a wide variety of RTOSes, or without an RTOS.

This module uses the following OAL components:

OAL Resource	Number Required
Tasks	2
Mutexes	1
Events	2

5.2 Utilities

The HTTP code creates and uses a single timer in the **hcc_timer** module. The **hcc_timer** module is included in your system when you install the base TCP/IP modules.

5.3 PSP Porting

The Platform Support Package (PSP) is designed to hold all platform-specific functionality, either because it relies on specific features of a target system, or because this provides the most efficient or flexible solution for the developer. For full details, see the *HCC Base Platform Support Package User Guide*.

The module makes use of the following standard PSP functions:

Function	Package	Element	Description
psp_memcpy()	psp_base	psp_string	Copies a block of memory. The result is a binary copy of the data.
psp_strncat()	psp_base	psp_string	Appends a string.
psp_strncmp()	psp_base	psp_string	Compares two strings of defined length.
psp_strncpy()	psp_base	psp_string	Copies one string of defined length to another.
psp_strnlen()	psp_base	psp_string	Gets the length of a string.

The module does not make use of any of the standard PSP macros.

6 Code Examples

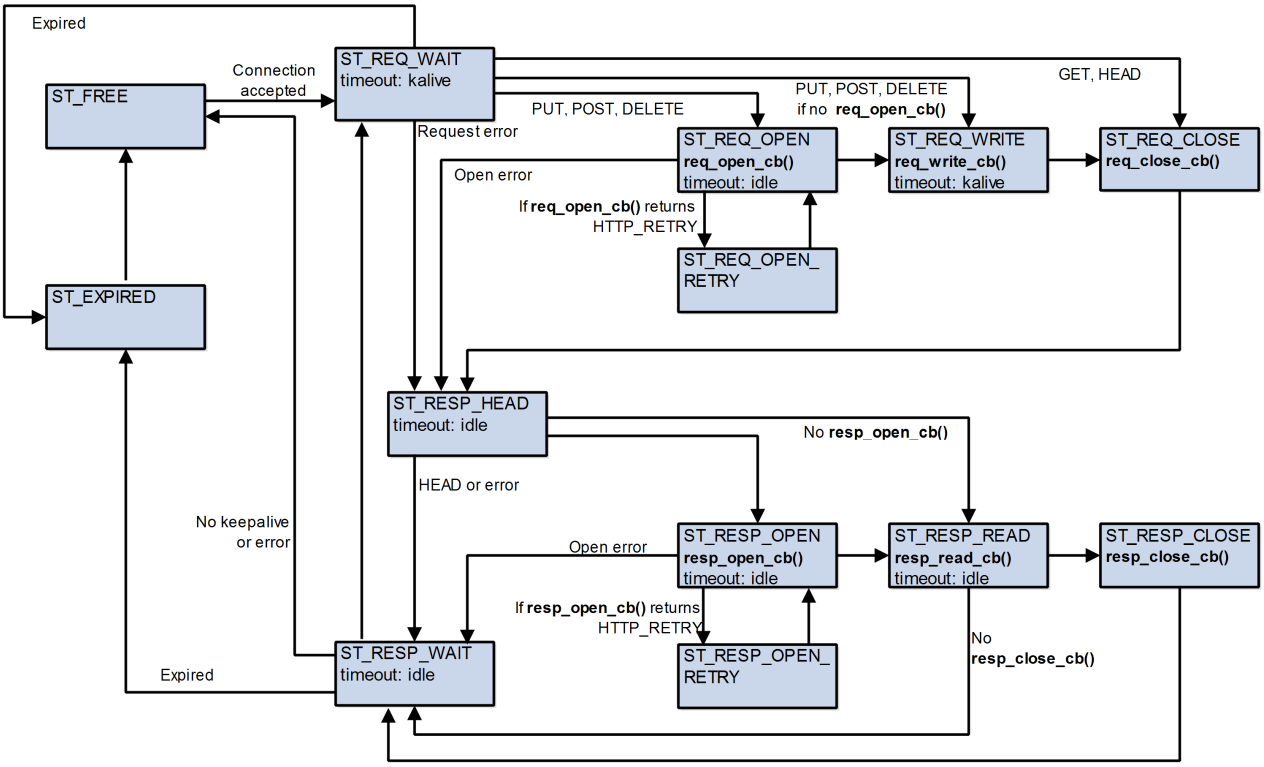
The first section below describes the HTTP flow in the form of a state machine. The sections that follow it show how to code each of four cases. These match the files in the **ip_app_http_demo** package. The implementations covered are as follows:

- User GET implementation.
- User POST implementation.
- User Get Input.
- User GET using the FAT file system.

Note: To increase the clarity of the state machine diagrams, the initial "**t_http_**" has been omitted from each callback name.

6.1 The HTTP Process

This diagram shows a complete picture of the HTTP flow in the form of a state machine. This shows how a connection is established and how the received HTTP request is processed. The callbacks in the state machine show where you should add user code to create the web pages (for clarity the initial "**t_http**_" has been omitted from each callback name).



The server waits in request wait state (ST_REQ_WAIT) until it receives a request from a client.

As an example of how the system operates, here is the basic process when the server receives a POST request:

1. If a **t_http_req_open_cb()** callback is available, it moves to response open state ST_REQ_OPEN.
2. The **t_http_req_open_cb()** callback allocates a resource if this is necessary. If none is available it retries later.
3. It moves to state ST_REQ_WRITE. The **t_http_req_write_cb()** callback is used to pass data received in the request. The implementation can decide how to process the data (for example, to store it in a structure or a file).
4. It moves to state ST_REQ_CLOSE. The **t_http_req_close_cb()** callback is used to close a file or to free any other resource. The response HEAD is sent after this function returns, so it must output the response information (content type and length).
5. It moves to ST_RESP_HEAD and from here starts the response.
6. If a **t_http_resp_open_cb()** callback is available, it moves to response open state ST_RESP_OPEN. The user implementation can decide whether a resource needs to be allocated and the callback tries to do this, retrying if necessary.

7. It moves to state ST_RESP_READ and the **t_http_resp_read_cb()** callback function is called to get the response data. The implementation can decide how to process the data (for example, read it from a file or generate content dynamically).
8. When all response data has been sent, the system moves to state ST_RESP_CLOSE and calls **t_http_resp_read_cb()**. This can be used to close a file or to release any other resource.

6.2 User GET Implementation

This example demonstrates the GET method with URL-encoded input and web pages generated without a file system.

When any HTML page is requested from the server, this implementation checks whether first and last names are available for the requester IP address. It then proceeds as follows:

- If the names are available it displays a welcome page including the names.
- If the names are not available it returns a web page that asks for the names.
- If the internal name table is full it displays a "Table full" web page.

Demo File

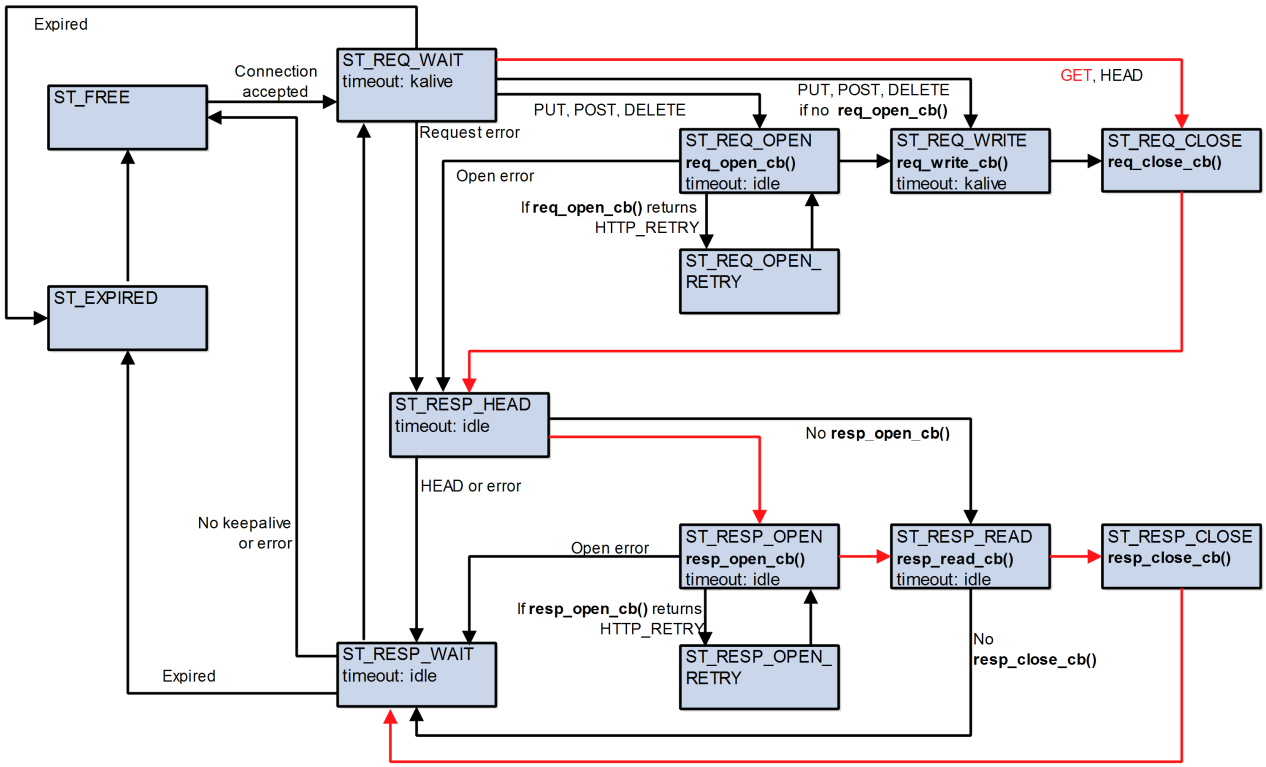
This implementation is covered by the file **http_user_get.c**.

Unused Callbacks

The **http_req_open_cb()** and **http_req_write_cb()** callbacks are not used in this implementation.

State Diagram

The red lines on this diagram show the GET process. In this example, both **resp_open_cb()** and **resp_close_cb()** callbacks are available.



Defines

```
#define INDEX_HTML       "/index.html"
#define DVAR_HTML        "/dvar.html"
#define FID_INDEX_HTML   0
#define FID_DVAR_HTML    1

/* index.html features */
#define LINE_COUNT       50000
#define LINE_SIZE        80
#define POS_STR_SIZE     4
#define FILE_SIZE        ( LINE_COUNT * LINE_SIZE )
#define MAX_USER_FILES   3

#define LINE_POS         "0000000"
#define LINE_POS_SIZE    ( sizeof( LINE_POS ) - 1u )
#define LINE_SEP         " - "
#define LINE_BR          "<br>"
#define LINE_DYN_VAR     "<var:my_str_0>"

/* dvar.html features */
#define DVAR_FILE_LENGTH ( sizeof( g_dvar_file ) - 1 )
#define DVAR_LINE_LENGTH 40
```

Typedefs

```
typedef struct
{
  uint8_t   b_used;
  uint8_t   fid;
  char_t    buf[LINE_SIZE + 1];
  uint32_t  pos;
  uint8_t   mcnt;
}
t_ufile;
```

Local Variables

```
/* callback descriptor */
static const t_http_cb_dsc  g_http_cb_dsc =
{
  http_req_open_cb
  , http_req_write_cb
  , http_req_close_cb
  , http_resp_open_cb
  , http_resp_read_cb
  , http_resp_close_cb
#if ( HTTP_DYNAMIC_VARIABLES != 0 )
  , http_dynvar_cb
#endif
};

static t_ufile  g_ufile[MAX_USER_FILES];
static char_t   g_pos[POS_STR_SIZE + 1];
#if ( HTTP_DYNAMIC_VARIABLES != 0 )
static uint8_t  g_call_cnt;
#endif

static char_t  g_dvar_file[] =
{
  "12345678901234567890123456789012345678901234567890"
  "test<va:ior><var:call_cnt><br>iiiiieee<b"
  "r>eaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa<var:"
  "my_str_0>bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb<"
  "my_str_0>ccccccccccccccccccccccccc<var:m"
  "y_str_0>ddddddd<var:my_str0>ddddddddddddw"
};
```

Functions for User and File Handling

inc_snum_val

This function increments the value of a string number.

```
static void inc_snum_val ( char_t snum[], uint16_t snum_size )
{
  uint16_t  cnt;

  cnt = snum_size;
  --snum_size;
  while ( cnt > 0u )
  {
    ++( snum[snum_size] );
    if ( snum[snum_size] > '9' )
    {
      snum[snum_size] = '0';
    }
    else
    {
      break;
    }

    --snum_size;
    --cnt;
  }
}
```

f_get_ufile

This function gets the pointer to the user file from the handle.

```
static t_ufile * f_get_ufile ( const uint32_t hdl )
{
  uint16_t  pos;
  uint8_t   mcnt;
  t_ufile * p_ufile;

  p_ufile = NULL;

  pos = hdl;
  mcnt = ( pos >> 8 );
  pos &= 0xffu;
  if ( pos < MAX_USER_FILES )
  {
    if ( g_ufile[pos].mcnt == mcnt )
    {
      p_ufile = &( g_ufile[pos] );
    }
  }

  return p_ufile;
}
```

http_user_init

In this implementation this function tries to allocate a user file.

```
t_http_ret http_user_init ( void )
{
  t_http_ret  ret_val;
  uint16_t    pos;

  for ( pos = 0u ; pos < MAX_NAMES ; pos++ )
  {
    g_name_list[pos].ip_addr = 0u;
    g_name_list[pos].firstn[0] = '\0';
    g_name_list[pos].lastn[0] = '\0';
  }

  for ( pos = 0u ; pos < HTTP_MAX_CONNECTIONS ; pos++ )
  {
    g_user_file[pos].hdl = pos;
    g_user_file[pos].b_used = FALSE;
  }
  ret_val = http_register_cb( &g_http_cb_dsc );

  return ret_val;
}
```

Callbacks

Unused Callbacks

The **http_req_open_cb()** and **http_req_write_cb()** callbacks are not used in this implementation.

http_req_close_cb

In this implementation this function checks whether the request method is GET, gets the content type based on the URI, and gets the length of the file (identified by URI) from the file system.

```
 static t_http_ret http_req_close_cb ( const t_http_req_info * const p_req_info
                                     , const uint32_t hdl
                                     , t_http_resp_info * const p_resp_info )
{
  t_http_ret  ret_val;
  HCC_UNUSED_ARG( hdl );
  ret_val = HTTP_ERROR;

  if ( ( ( p_req_info->hreq_method ) == HTTP_METHOD_HEAD )
      || ( ( p_req_info->hreq_method ) == HTTP_METHOD_GET ) )
  {
    /* get response content type */
    ret_val = http_get_content_type( p_req_info->p_hreq_uri
                                   , &( p_resp_info->p_hresp_ctype ) );
    if ( ret_val == HTTP_SUCCESS )
    {
      if ( strcmp( p_req_info->p_hreq_uri, INDEX_HTML ) == NULL )
      {
#if ( HTTP_DYNAMIC_VARIABLES != 0 )
        p_resp_info->hresp_clen = HTTP_LENGTH_UNKNOWN;
#else
        p_resp_info->hresp_clen = FILE_SIZE;
#endif
      }
      else if ( strcmp( p_req_info->p_hreq_uri, DVAR_HTML ) == NULL )
      {
#if ( HTTP_DYNAMIC_VARIABLES != 0 )
        p_resp_info->hresp_clen = HTTP_LENGTH_UNKNOWN;
#else
        p_resp_info->hresp_clen = DVAR_FILE_LENGTH;
#endif
      }
      else
      {
        ret_val = HTTP_ERR_NOT_FOUND;
      }
    }
  }
  return ret_val;
}
```

http_resp_open_cb

In this implementation this function searches for a free file handle. If it finds one it tries to open the file (identified by URI), otherwise it tells the server to retry later.

```
 static t_http_ret http_resp_open_cb ( const t_http_req_info * const p_req_info
                                     , const uint32_t hdl
                                     , uint32_t * const p_new_hdl )
{
  t_http_ret  ret_val;
  t_ufile   * p_ufile;
  uint32_t    pos;
  uint16_t    lpos;

  HCC_UNUSED_ARG( hdl );
  ret_val = HTTP_SUCCESS;

  for ( pos = 0u ; pos < MAX_USER_FILES ; pos++ )
  {
    p_ufile = &( g_ufile[pos] );
    if ( p_ufile->b_used == FALSE )
    {
      break;
    }
  }

  if ( pos < MAX_USER_FILES )
  {
    *p_new_hdl =  pos + ( ( p_ufile->mcnt ) << 8u );
    if ( strcmp( p_req_info->p_hreq_uri, INDEX_HTML ) == NULL )
    {
      p_ufile->b_used = TRUE;
      p_ufile->fid    = FID_INDEX_HTML;
      p_ufile->pos    = 0u;

      inc_snum_val( g_pos, POS_STR_SIZE );
      strcpy( p_ufile->buf, LINE_POS );
      strcat( p_ufile->buf, LINE_SEP );
      strcat( p_ufile->buf, g_pos );
      strcat( p_ufile->buf, LINE_SEP );
#if ( HTTP_DYNAMIC_VARIABLES != 0 )
      strcat( p_ufile->buf, LINE_DYN_VAR );
      strcat( p_ufile->buf, LINE_SEP );
#endif
      for ( lpos = strlen( p_ufile->buf )
           ; lpos < LINE_SIZE - 4u
           ; lpos++ )
      {
        p_ufile->buf[lpos] = pos + '0';
      }

      p_ufile->buf[lpos] = '\0';
      strcat( p_ufile->buf, LINE_BR );
    }
    else if ( strcmp( p_req_info->p_hreq_uri, DVAR_HTML ) == NULL )
    {
```

```
      p_ufile->b_used = TRUE;
      p_ufile->fid    = FID_DVAR_HTML;
      p_ufile->pos    = 0u;
    }
    else
    {
      ret_val = HTTP_ERROR;
    }
  }
  else
  {
    ret_val = HTTP_RETRY;
  }

  return ret_val;
}
```

http_resp_read_cb

In this implementation this function simply fills the buffer with the data read from a file (opened by the **http_resp_open_cb()** callback).

```
 static t_http_ret http_resp_read_cb ( const t_http_req_info * const p_req_info
                                     , const uint32_t hdl
                                     , uint8_t * const p_buf
                                     , const uint16_t buf_len
                                     , uint16_t * const p_rd_len )
{
  t_http_ret  ret_val;
  t_ufile   * p_ufile;
  uint16_t    rlen;
  uint16_t    tot_len;

  HCC_UNUSED_ARG( p_req_info );

  ret_val = HTTP_SUCCESS;
  rlen = 0u;
  tot_len = buf_len;

  p_ufile = f_get_ufile( hdl );
  if ( p_ufile != NULL )
  {
    if ( ( p_ufile->fid ) == FID_INDEX_HTML )
    {
      while ( ( tot_len >= LINE_SIZE )
             && ( p_ufile->pos < FILE_SIZE ) )
      {
        inc_snum_val( p_ufile->buf, LINE_POS_SIZE );

        memcpy( &( p_buf[rlen] ), p_ufile->buf, LINE_SIZE );
        p_ufile->pos += LINE_SIZE;
        rlen += LINE_SIZE;
        tot_len -= LINE_SIZE;
      }
    }
    else if ( ( p_ufile->fid == FID_DVAR_HTML ) == FID_DVAR_HTML )
    {
      if ( ( p_ufile->pos ) < DVAR_FILE_LENGTH )
      {
        memcpy( &( p_buf[rlen] ), &( g_dvar_file[p_ufile->pos] ), DVAR_LINE_LENGTH );
        p_ufile->pos += DVAR_LINE_LENGTH;
        rlen = DVAR_LINE_LENGTH;
      }
    }
  }
  else
  {
    ret_val = HTTP_ERROR;
  }
  if ( ret_val == HTTP_SUCCESS )
  {
    *p_rd_len = rlen;
  }

  return ret_val;
}
```

http_resp_close_cb

In this implementation this function closes the file opened by **http_resp_open_cb()**.

```
 static t_http_ret http_resp_close_cb ( const t_http_req_info * const p_req_info
                                      , const uint32_t hdl )
{
  HCC_UNUSED_ARG( p_req_info );

  t_ufile * p_ufile;

  p_ufile = f_get_ufile( hdl );
  if ( p_ufile != NULL )
  {
    p_ufile->b_used = FALSE;
  }

  return HTTP_SUCCESS;
}
```

http_dynvar_cb

This function is the dynamic variable handler.

```
#if ( HTTP_DYNAMIC_VARIABLES != 0 )
static t_http_ret http_dynvar_cb ( const t_http_req_info * const p_req_info
                                  , const char_t * const p_var_name
                                  , char_t * const p_var_value )
{
  t_http_ret  ret_val;

  HCC_UNUSED_ARG( p_req_info );

  ret_val = HTTP_SUCCESS;

  if ( strcmp( p_var_name, "my_str_0" ) == NULL )
  {
    strcpy( p_var_value, "--MyString0--" );
  }
  else if ( strcmp( p_var_name, "call_cnt" ) == NULL )
  {
    sprintf( p_var_value, "--%d--", g_call_cnt );
    ++g_call_cnt;
  }
  else
  {
    ret_val = HTTP_ERROR;
  }

  return ret_val;
} /* http_dynvar_cb */
#endif /* if ( HTTP_DYNAMIC_VARIABLES != 0 ) */
```

6.3 User POST Implementation

This example demonstrates the user POST method with URL-encoded input and web pages generated without a file system.

When any HTML page is requested from the server, this implementation checks whether first and last names are available for the requester IP address. It then proceeds as follows:

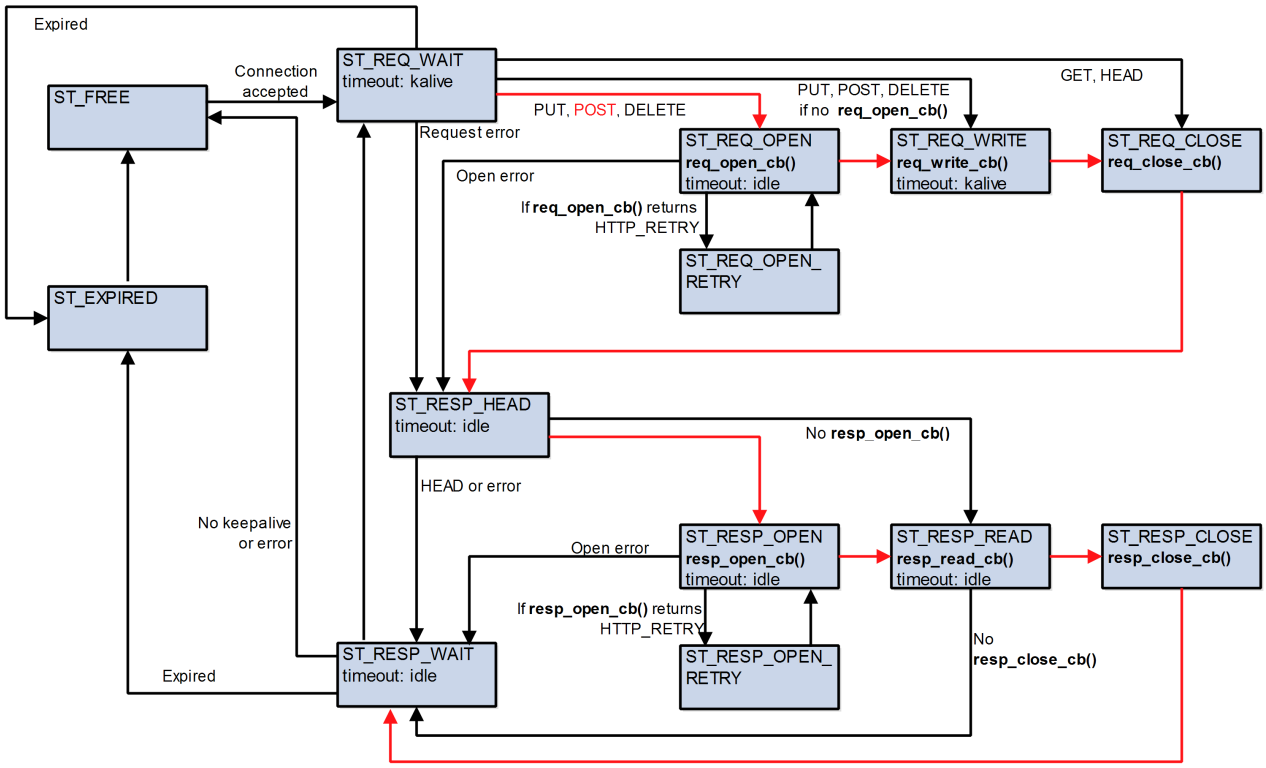
- If the names are available it displays a welcome page including the names.
- If the names are not available it returns a web page that asks for the names.
- If the internal name table is full it displays a "Table full" web page.

Demo File

This implementation is covered by the file **http_user_post.c**.

State Diagram

The red lines on this diagram show the POST process. In this example, both **resp_open_cb()** and **resp_close_cb()** callbacks are available.



Defines and Typedefs

```
#define SUCCESS        0u
#define ERR_TABLE_FULL 1u
#define ERROR          2u
#define MAX_NAMES      4     /* max. entries in the name table */
#define MAX_NAME_SIZE  16    /* max. first and last name size */
#define FIRST_NAME     "first_name"    /* first name identifier */
#define LAST_NAME      "last_name"     /* last name identifier */

/* name entry */
typedef struct
{
    uint32_t  ip_addr;               /* IP address */
    char_t    firstn[MAX_NAME_SIZE]; /* first name */
    char_t    lastn[MAX_NAME_SIZE];  /* last name */
} t_name_entry;

/* user file */
typedef struct
{
    uint16_t  hdl;            /* handle */
    uint8_t   b_used;         /* TRUE if entry is used */
    int       save_name_ret;  /* return code of save_name() in case of POST */
    uint32_t  fremain;        /* remaining bytes from the file */
    char_t  * p_buf;          /* pointer to the current position in the file */
} t_user_file;
```

Local Variables

```
static t_name_entry  g_name_list[MAX_NAMES];  /* name table */
/* user files - for simplicity the no. files is equal to the max. no. HTTP connections */
static t_user_file  g_user_file[HTTP_MAX_CONNECTIONS];

/* callback descriptor */
static const t_http_cb_dsc  g_http_cb_dsc =
{
    http_req_open_cb
    , http_req_write_cb
    , http_req_close_cb
    , http_resp_open_cb
    , http_resp_read_cb
    , http_resp_close_cb
#if ( HTTP_DYNAMIC_VARIABLES != 0 )
    , http_dynvar_cb
#endif
};

/* Enter name request form */
static char_t  g_entername_html[] =
{
    "<html>"
    "<body>"
    "  <b>Please enter your name:</b><br>"
    "  <form action=\"index.html\" method=\"get\">"
    "    First name: <input type=\"text\" name=\""FIRST_NAME "\"><br>"
    "    Last name: <input type=\"text\" name=\""LAST_NAME "\"><br>"
    "    <input type=\"submit\" value=\"Send\">"
    "  </form>"
    "</html>"
    "</body>"
};

/* Welcome page */
static char_t  g_welcome_html[] =
{
    "<html>"
    "<body>"
    " <b><var:"FIRST_NAME "> <var:"LAST_NAME "></b> welcome on HCC demo page."
    "</html>"
    "</body>"
};

/* Table full message */
static char_t  g_tablefull_html[] =
{
    "<html>"
    "<body>"
    "  <b>Sorry table is full. Please increase MAX_NAMES.</b>"
    "</html>"
    "</body>"
};
```

Functions for User and File Handling

get_free_user_file

This function searches for a free user file. On return *pp_uf* shows where to write the pointer to the free user file.

```
 static int get_free_user_file ( t_user_file * * const pp_uf )
{
    int       ret_val;
    uint16_t  pos;
    ret_val = ERROR;

    /* Search for an available user file */
    for ( pos = 0u
        ; ( pos < HTTP_MAX_CONNECTIONS ) && ( g_user_file[pos].b_used == TRUE )
        ; pos++ )
    {
    }

    if ( pos < HTTP_MAX_CONNECTIONS )  /* User file present */
    {
        *pp_uf = &( g_user_file[pos] );
        ret_val = SUCCESS;
    }

  return ret_val;
}
```

get_name

This function searches for a free user file. On return *pp_uf* shows where to write the pointer to the free user file.

```
 static int get_name ( const t_http_req_info * const p_req_info
                     , const t_name_entry * * const pp_name_entry )
{
    int            ret_val;       /* return value */
    t_name_entry * p_name_entry;  /* pointer to the name entry */
    uint8_t        pos;           /* position */
    ret_val = ERROR;

    /* Step through the name table */
    for ( pos = 0u
        ; ( pos < MAX_NAMES ) && ( ret_val == ERROR )
        ; pos++ )
    {
        p_name_entry = &( g_name_list[pos] );
        /* Matching IP address, entry found. */
        if ( ( p_name_entry->ip_addr ) == ( p_req_info->hreq_ip_port.ipp_ip_addr ) )
        {
            *pp_name_entry = p_name_entry;  /* Set output parameter */
            ret_val = SUCCESS;
        }
    }
  return ret_val;
}
```

save_name

This function tries to place (or update) the name received with POST in the name table.

```
static int save_name ( const t_http_req_info * const p_req_info
                     , const char_t * const p_buf
                     , const uint16_t buf_len )
{
    int            ret_val;       /* return value */
    t_name_entry * p_name_entry;  /* pointer to the name entry */
    uint8_t        pos;           /* position */
    uint8_t        fpos;          /* free position */

    uint8_t        b_found;       /* TRUE if entry was found or added */
    ret_val = SUCCESS;
    b_found = FALSE;
    fpos = MAX_NAMES;

    /* Step through the name table */
    for ( pos = 0u
        ; ( pos < MAX_NAMES ) && ( b_found == FALSE )
        ; pos++ )
    {
        p_name_entry = &( g_name_list[pos] );
        /* Store entry position if the entry is free */
        if ( ( fpos == MAX_NAMES ) && ( ( p_name_entry->ip_addr ) == 0u ) )
        {
            fpos = pos;
        }
        /* Matching IP address, entry found. */
        if ( ( p_name_entry->ip_addr ) == ( p_req_info->hreq_ip_port.ipp_ip_addr ) )
        {
            b_found = TRUE;
        }
    }
    if ( b_found == FALSE )                         /* entry was not found */
    {
        if ( fpos < MAX_NAMES )                     /* and there is free space in the table */
        {
            p_name_entry = &( g_name_list[fpos] );  /* set name entry to the free position */
        }
        else                                        /* no free position */
        {
            ret_val = ERR_TABLE_FULL;               /* set table full error code */
        }
    }
    /* Name entry found or free position available */
    if ( ret_val == SUCCESS )
    {
    /* Try to get first name from URL encoded GET input */
        if ( http_get_urlenc_value( p_buf
                                  , buf_len
                                  , FIRST_NAME
                                  , &( p_name_entry->firstn[0] )
                                  , MAX_NAME_SIZE ) == HTTP_SUCCESS )
        {
```

```
            /* Try to get last name from URL encoded GET input */
            if ( http_get_urlenc_value( p_buf
                                       , buf_len
                                       , LAST_NAME
                                       , &( p_name_entry->lastn[0] )
                                       , MAX_NAME_SIZE ) == HTTP_SUCCESS )
            {
          /* First and last name were obtained: store IP address and set b_found to TRUE */
                p_name_entry->ip_addr = p_req_info->hreq_ip_port.ipp_ip_addr;
                b_found = TRUE;
            }
        }
    }
    if ( b_found == FALSE )  /* entry not found */
    {
        ret_val = ERROR;       /* Set error code */
    }
  return ret_val;
}
```

Callbacks

http_req_open_cb

In this implementation this function tries to allocate a user file.

```
static t_http_ret http_req_open_cb ( const t_http_req_info * const p_req_info
                                   , uint32_t * const p_hdl )
{
    int           ret_val;
    t_user_file * p_uf;
    HCC_UNUSED_ARG( p_req_info );
    ret_val = HTTP_ERROR;
    if ( get_free_user_file( &p_uf ) == SUCCESS )
    {
        *p_hdl = p_uf->hdl;
        p_uf->b_used = TRUE;  /* set user file used flag */
        ret_val = HTTP_SUCCESS;
    }
  return ret_val;
}
```

http_req_write_cb

In this implementation this function tries to obtain the name from the POST request.

```
static t_http_ret http_req_write_cb ( const t_http_req_info * const p_req_info
                                    , const uint32_t hdl
                                    , const uint8_t * const p_buf
                                    , const uint16_t buf_len )
{
    g_user_file[hdl].save_name_ret = save_name( p_req_info, (char_t *)p_buf, buf_len );
    return HTTP_SUCCESS;
}
```

http_req_close_cb

In this implementation this function checks if the request method was GET, gets the content type based on the URI, and allocates a user file which is set to a specific (welcome, table full or name request) page, depending on the availability of the name based on the IP address.

```
static t_http_ret http_req_close_cb ( const t_http_req_info * const p_req_info
                                    , const uint32_t hdl
                                    , t_http_resp_info * const p_resp_info )
{
    t_http_ret           ret_val;
    int                  name_ret;
    t_user_file        * p_uf;
    const t_name_entry * p_name_entry;

    HCC_UNUSED_ARG( hdl );
    p_uf = &( g_user_file[hdl] );

    ret_val = HTTP_ERROR;
    if ( ( ( p_req_info->hreq_method ) == HTTP_METHOD_HEAD )
      || ( ( p_req_info->hreq_method ) == HTTP_METHOD_GET )
      || ( ( p_req_info->hreq_method ) == HTTP_METHOD_POST ) )
    {
    /* Get response content type */
        ret_val = http_get_content_type( p_req_info->p_hreq_uri
                                       , &( p_resp_info->p_hresp_ctype ) );
        if ( ret_val == HTTP_SUCCESS )
        {
            if ( ( p_req_info->hreq_method ) == HTTP_METHOD_GET )  /* GET request */
            {
                /* Try to get the name based on the request information (IP address) */
                name_ret = get_name( p_req_info, &p_name_entry );
            }
            else  /* POST request */
            {
                /* In case of POST request set the return code saved at request write callback
*/
                /* from save_name() call */
                name_ret = p_uf->save_name_ret;
            }

            if ( name_ret == SUCCESS ) /* Name is present - return the welcome page */
            {
                p_uf->p_buf = &( g_welcome_html[0] ); /* Set user file's buffer pointer */
                p_uf->fremain = sizeof( g_welcome_html ) - 1u;  /* Set remaining size */
            }
            else if ( name_ret == ERR_TABLE_FULL ) /* Table is full - return table full page */
            {
                p_uf->p_buf = &( g_tablefull_html[0] ); /* Set user file's buffer pointer */
                p_uf->fremain = sizeof( g_tablefull_html ) - 1u; /* Set remaining size */
            }
            else    /* Name is not available, return name request page */
            {
                p_uf->p_buf = &( g_entername_html[0] ); /* Set user file's buffer pointer */
                p_uf->fremain = sizeof( g_entername_html ) - 1u; /* Set remaining size */
```

```
            }
            p_resp_info->hresp_clen = p_uf->fremain; /* Set response content length */
        }
    }

    if ( ( ( p_req_info->hreq_method ) == HTTP_METHOD_HEAD )
            || ( ret_val != HTTP_SUCCESS ) )
    {
        p_uf->b_used = FALSE;  /* Free user file */
    }
  return ret_val;
}
```

http_resp_open_cb

In this implementation this function just sets the new handle to the handle used for the request callback functions, as the user file is already allocated by **http_req_close_cb()**.

```
 static t_http_ret http_resp_open_cb ( const t_http_req_info * const p_req_info
                                     , const uint32_t hdl
                                     , uint32_t * const p_new_hdl )
{
    HCC_UNUSED_ARG( p_req_info );

    *p_new_hdl = hdl;

    return HTTP_SUCCESS;
}
```

http_resp_read_cb

In this implementation this function simply copies data from the user file (pointing to a static web page) to the output buffer.

```
static t_http_ret http_resp_read_cb ( const t_http_req_info * const p_req_info
                                    , const uint32_t hdl
                                    , uint8_t * const p_buf
                                    , const uint16_t buf_len
                                    , uint16_t * const p_rd_len )
{
    t_user_file * p_uf;
    uint16_t      rlen;

    HCC_UNUSED_ARG( p_req_info );

    p_uf = &( g_user_file[hdl] );       /* Get pointer to the user file */
    if ( buf_len > ( p_uf->fremain ) )  /* Buffer length is more than the unsent data */
    {
        rlen = p_uf->fremain;           /* Set read length to remaining data */
    }
    else                                /* Buffer size is smaller than remaining data */
    {
        rlen = buf_len;                 /* Set read length to buffer length */
    }
    if ( rlen > 0u )                    /* If there is anything to read */
    {
        memcpy( p_buf, p_uf->p_buf, rlen );  /* Copy data to destination buffer */
        p_uf->fremain -= rlen;               /* Decrease remaining length */
        p_uf->p_buf += rlen;                 /* Set new position in the buffer */
    }
    *p_rd_len = rlen;                   /* Set read length output parameter */

    return HTTP_SUCCESS;
}
```

http_resp_close_cb

In this implementation this function marks the user file free.

```
 static t_http_ret http_resp_close_cb ( const t_http_req_info * const p_req_info
                                      , const uint32_t hdl )
{
    HCC_UNUSED_ARG( p_req_info );
    g_user_file[hdl].b_used = FALSE;
    return HTTP_SUCCESS;
}
```

http_dynvar_cb

This function is the dynamic variable handler.

```
#if ( HTTP_DYNAMIC_VARIABLES != 0 )
static t_http_ret http_dynvar_cb ( const t_http_req_info * const p_req_info
                                 , const char_t * const p_var_name
                                 , char_t * const p_var_value )
{
    t_http_ret           ret_val;
    int                  name_ret;
    const t_name_entry * p_name_entry;
    HCC_UNUSED_ARG( p_req_info );
    ret_val = HTTP_SUCCESS;
    name_ret = get_name( p_req_info, &p_name_entry );
    if ( name_ret == SUCCESS )
    {
        if ( strcmp( p_var_name, FIRST_NAME ) == NULL )
        {
            strcpy( p_var_value, &( p_name_entry->firstn[0] ) );
        }
        else if ( strcmp( p_var_name, LAST_NAME ) == NULL )
        {
            strcpy( p_var_value, &( p_name_entry->lastn[0] ) );
        }
        else
        {
            ret_val = HTTP_ERROR;
        }
    }
    else
    {
        ret_val = HTTP_ERROR;
    }

    return ret_val;
} /* http_dynvar_cb */
#endif
```

6.4 User Get Input

This example demonstrates the GET method with URL-encoded input and web pages generated without a file system.

When any HTML page is requested from the server, this implementation checks whether first and last names are available for the requester IP address. It then proceeds as follows:

- If the names are available it displays a welcome page including the two names.
- If the names are not available it returns a web page that asks for the names.
- If the internal name table is full it displays a "Table full" web page.

Demo File

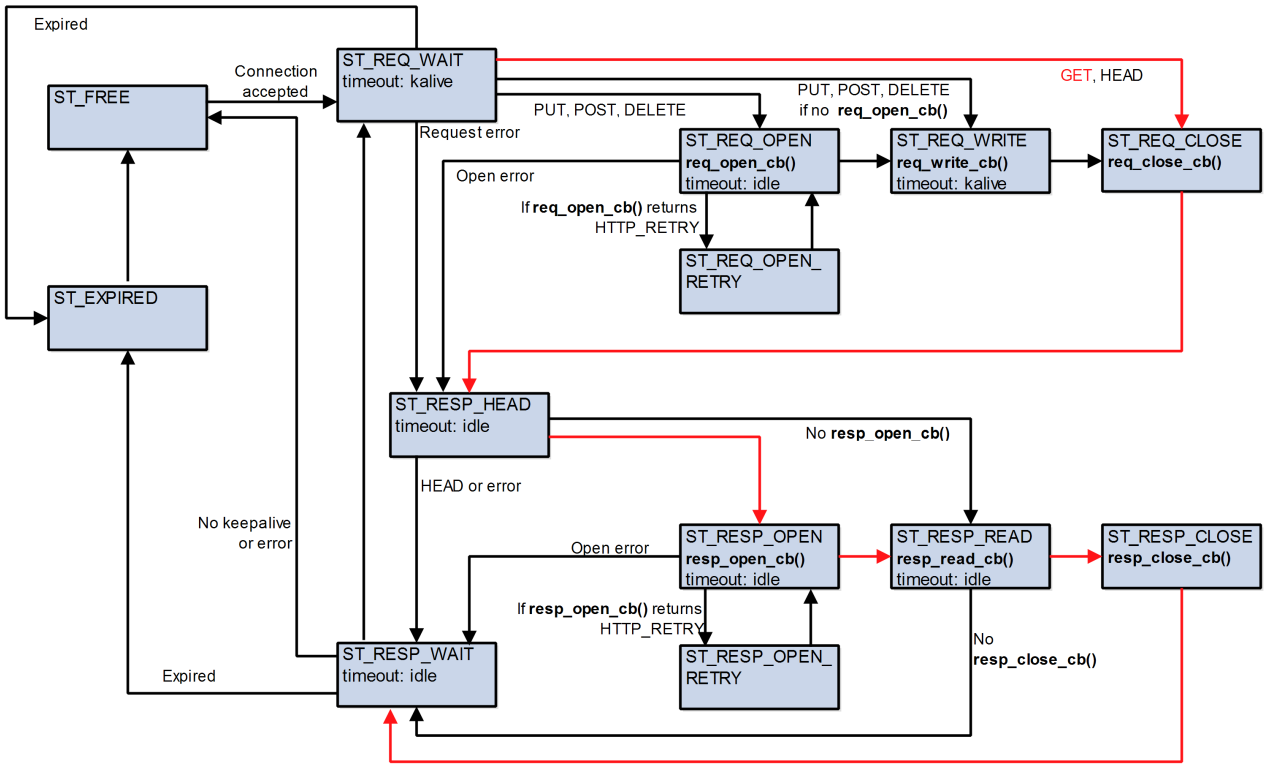
This implementation is covered by the file **http_user_get_input.c**.

Unused Callbacks

The **http_req_write_cb()** callback is not used in this implementation.

State Diagram

The red lines on this diagram show the GET process. In this example, both **resp_open_cb()** and **resp_close_cb()** callbacks are available.



Defines

```
/* internal error codes */
#define SUCCESS        0u
#define ERR_TABLE_FULL 1u
#define ERROR          2u

#define MAX_NAMES      4     /* Maximum number of entries in the name table */
#define MAX_NAME_SIZE  16    /* Maximum first and last name size */

#define FIRST_NAME     "first_name"    /* First name identifier */
#define LAST_NAME      "last_name"     /* Last name identifier */
```

Typedefs

```
/* name entry */
typedef struct
{
  uint32_t  ip_addr;                /* IP address */
  char_t    firstn[MAX_NAME_SIZE];  /* first name */
  char_t    lastn[MAX_NAME_SIZE];   /* last name */
} t_name_entry;

/* user file */
typedef struct
{
  uint16_t  hdl;       /* handle */
  uint8_t   b_used;    /* TRUE if entry is used */
  uint32_t  fremain;   /* remaining bytes from the file */
  char_t  * p_buf;     /* pointer to the current position in the file */
} t_user_file;
```

Local Variables

```
static t_name_entry  g_name_list[MAX_NAMES];  /* name table */
/* user files -, for simplicity no. of files is equal to  max. no. of HTTP connections */
static t_user_file  g_user_file[HTTP_MAX_CONNECTIONS];

/* callback descriptor */
static const t_http_cb_dsc  g_http_cb_dsc =
{
  http_req_open_cb
  , http_req_write_cb
  , http_req_close_cb
  , http_resp_open_cb
  , http_resp_read_cb
  , http_resp_close_cb
#if ( HTTP_DYNAMIC_VARIABLES != 0 )
  , http_dynvar_cb
#endif
};

/* Enter name request form */
static char_t  g_entername_html[] =
{
  "<html>"
  "<body>"
  "  <b>Please enter your name:</b><br>"
  "  <form action=\"index.html\" method=\"get\">"
  "    First name: <input type=\"text\" name=\""FIRST_NAME "\"><br>"
  "    Last name: <input type=\"text\" name=\""LAST_NAME "\"><br>"
  "    <input type=\"submit\" value=\"Send\">"
  "  </form>"
  "</html>"
  "</body>"
};

/* Welcome page */
static char_t  g_welcome_html[] =
{
  "<html>"
  "<body>"
  " <b><var:"FIRST_NAME "> <var:"LAST_NAME "></b> welcome on HCC demo page."
  "</html>"
  "</body>"
};
/* Table full message */
static char_t  g_tablefull_html[] =
{
  "<html>"
  "<body>"
  "  <b>Sorry table is full. Please increase MAX_NAMES.</b>"
  "</html>"
  "</body>"
};
```

Functions for User and File Handling

get_free_user_file

This function searches for a free user file. On return *pp_uf* shows where to write the pointer to the free user file.

```
 static int get_free_user_file ( t_user_file * * const pp_uf )
{
  int       ret_val;
  uint16_t  pos;
  ret_val = ERROR;

  /* search for an available user file */
  for ( pos = 0u
       ; ( pos < HTTP_MAX_CONNECTIONS ) && ( g_user_file[pos].b_used == TRUE )
       ; pos++ )
  {
    if ( pos < HTTP_MAX_CONNECTIONS )  /* user file present */
    {
      *pp_uf = &( g_user_file[pos] );
      ret_val = SUCCESS;
    }
  }
  return ret_val;
}
```

get_name

This function searches for the name based on the IP address.

On return *pp_name_entry* shows where to write the pointer to the name entry.

```
static int get_name ( const t_http_req_info * const p_req_info
                    , const t_name_entry * * const pp_name_entry )
{
  int            ret_val;       /* return value */
  t_name_entry * p_name_entry;  /* pointer to the name entry */
  uint8_t        pos;           /* position */
  uint8_t        fpos;          /* free position */
  uint8_t        b_found;       /* TRUE if entry was found or added */

  ret_val = SUCCESS;
  b_found = FALSE;
  fpos = MAX_NAMES;

  /* step through the name table */
  for ( pos = 0u
       ; ( pos < MAX_NAMES ) && ( b_found == FALSE )
       ; pos++ )
  {
    p_name_entry = &( g_name_list[pos] );

    /* store entry position if the entry is free */
    if ( ( fpos == MAX_NAMES )
        && ( ( p_name_entry->ip_addr ) == 0u ) )
    {
      fpos = pos;
    }

    /* matching IP address, entry found. */
    if ( ( p_name_entry->ip_addr ) == ( p_req_info->hreq_ip_port.ipp_ip_addr ) )
    {
      b_found = TRUE;
    }
  }

  if ( b_found == FALSE )                   /* entry was not found */
  {
    if ( fpos < MAX_NAMES )                 /* and there is free space in the table */
    {
      p_name_entry = &( g_name_list[fpos] );  /* set name entry to the free position */
    }
    else                                    /* no free position */
    {
      ret_val = ERR_TABLE_FULL;             /* set table full error code */
    }
  }

  /* name entry found or free position available */
  if ( ret_val == SUCCESS )
  {
    /* try to get first name from URL encoded GET input */
```

```
    if ( http_get_urlenc_value( p_req_info->p_hreq_uri_input
                              , p_req_info->hreq_uri_input_len
                              , FIRST_NAME
                              , &( p_name_entry->firstn[0] )
                              , MAX_NAME_SIZE ) == HTTP_SUCCESS )
    {
      /* try to get last name from URL encoded GET input */
      if ( http_get_urlenc_value( p_req_info->p_hreq_uri_input
                                , p_req_info->hreq_uri_input_len
                                , LAST_NAME
                                , &( p_name_entry->lastn[0] )
                                , MAX_NAME_SIZE ) == HTTP_SUCCESS )
      {
        /* first and last name could be obtained, store IP address */
        /* and set b_found to TRUE */
        p_name_entry->ip_addr = p_req_info->hreq_ip_port.ipp_ip_addr;
        b_found = TRUE;
      }
    }
  }

  if ( b_found == TRUE )           /* entry found */
  {
    *pp_name_entry = p_name_entry; /* set output parameter */
  }
  else                             /* entry was not found */
  {
    ret_val = ERROR;               /* set error code */
  }

  return ret_val;
}
```

http_user_init

In this implementation this function tries to allocate a user file.

```
t_http_ret http_user_init ( void )
{
  t_http_ret  ret_val;
  uint16_t    pos;
  for ( pos = 0u ; pos < MAX_NAMES ; pos++ )
  {
    g_name_list[pos].ip_addr = 0u;
    g_name_list[pos].firstn[0] = '\0';
    g_name_list[pos].lastn[0] = '\0';
  }
  for ( pos = 0u ; pos < HTTP_MAX_CONNECTIONS ; pos++ )
  {
    g_user_file[pos].hdl = pos;
    g_user_file[pos].b_used = FALSE;
  }
  ret_val = http_register_cb( &g_http_cb_dsc );
  return ret_val;
}
```

Callbacks

Unused Callbacks

The **http_req_write_cb()** callback is not used in this implementation.

http_req_open_cb

In this implementation this function tries to allocate a user file.

```
static t_http_ret http_req_open_cb ( const t_http_req_info * const p_req_info
                                    , uint32_t * const p_hdl )
{
  int           ret_val;
  t_user_file * p_uf;
  HCC_UNUSED_ARG( p_req_info );

  ret_val = HTTP_ERROR;

  if ( get_free_user_file( &p_uf ) == SUCCESS )
  {
    *p_hdl = p_uf->hdl;
    p_uf->b_used = TRUE;  /* set user file used flag */
    ret_val = HTTP_SUCCESS;
  }

  return ret_val;
}
```

http_req_close_cb

In this implementation this function checks whether the request method is GET, gets the content type based on the URI, allocates a user file which is set to a specific (welcome, table full or name request) page depending on the availability of the name based on the IP address.

```
 static t_http_ret http_req_close_cb ( const t_http_req_info * const p_req_info
                                     , const uint32_t hdl
                                     , t_http_resp_info * const p_resp_info )
{
  t_http_ret           ret_val;
  int                  name_ret;
  t_user_file        * p_uf;
  const t_name_entry * p_name_entry;

  HCC_UNUSED_ARG( hdl );
  ret_val = HTTP_ERROR;

  if ( ( ( p_req_info->hreq_method ) == HTTP_METHOD_HEAD )
      || ( ( p_req_info->hreq_method ) == HTTP_METHOD_GET ) )
  {
    /* get response content type */
    ret_val = http_get_content_type( p_req_info->p_hreq_uri
                                   , &( p_resp_info->p_hresp_ctype ) );
    if ( ret_val == HTTP_SUCCESS )
    {
      p_uf = &( g_user_file[hdl] );
      /* try to get the name based on the request information (IP address) */
      name_ret = get_name( p_req_info, &p_name_entry );
      if ( name_ret == SUCCESS )                      /* name present, return the */
      {                                               /* welcome page */
        p_uf->p_buf = &( g_welcome_html[0] );         /* set user file's buffer pointer */
        p_uf->fremain = sizeof( g_welcome_html ) - 1u; /* set remaining size */
      }
      else if ( name_ret == ERR_TABLE_FULL )            /* table is full, return table full */
      {                                                 /* page */
        p_uf->p_buf = &( g_tablefull_html[0] );         /* set user file's buffer pointer */
        p_uf->fremain = sizeof( g_tablefull_html ) - 1u; /* set remaining size */
      }
      else                                              /* name is not available, return */
      {                                                 /* name request page */
        p_uf->p_buf = &( g_entername_html[0] );         /* set user file's buffer pointer */
        p_uf->fremain = sizeof( g_entername_html ) - 1u; /* set remaining size */
      }

      p_resp_info->hresp_clen = p_uf->fremain;  /* set response content length */
    }
  }

  if ( ( ( p_req_info->hreq_method ) == HTTP_METHOD_HEAD )
      || ( ret_val != HTTP_SUCCESS ) )
  {
    p_uf->b_used = FALSE;  /* free user file */
  }

  return ret_val;
} /
```

http_resp_open_cb

In this implementation this function just sets the new handle to the handle used for the request callback functions, as the user file is already allocated by **http_req_close_cb()**.

> **Note:** There is no need for a new handle; set the handle to the current handle.

```
 static t_http_ret http_resp_open_cb ( const t_http_req_info * const p_req_info
                                     , const uint32_t hdl
                                     , uint32_t * const p_new_hdl )
{
  HCC_UNUSED_ARG( p_req_info );

  *p_new_hdl = hdl;

  return HTTP_SUCCESS;
}
```

http_resp_read_cb

In this implementation this function simply copies data from the user file (pointing to a static web page) to the output buffer.

```
 static t_http_ret http_resp_read_cb ( const t_http_req_info * const p_req_info
                                     , const uint32_t hdl
                                     , uint8_t * const p_buf
                                     , const uint16_t buf_len
                                     , uint16_t * const p_rd_len )
{
  t_user_file * p_uf;
  uint16_t      rlen;

  HCC_UNUSED_ARG( p_req_info );

  p_uf = &( g_user_file[hdl] );       /* get pointer to the user file */
  if ( buf_len > ( p_uf->fremain ) )  /* buffer length is more than the unsent */
  {                                   /* data */
    rlen = p_uf->fremain;             /* set read length to remaining data */
  }
  else                                /* buffer size is smaller than remaining */
  {                                   /* data */
    rlen = buf_len;                   /* set read length to buffer length */
  }
  if ( rlen > 0u )                       /* if there is anything to read */
  {
    memcpy( p_buf, p_uf->p_buf, rlen );  /* copy data to destination buffer */
    p_uf->fremain -= rlen;               /* decrease remaining length */
    p_uf->p_buf += rlen;                 /* set new position in the buffer */
  }
  *p_rd_len = rlen;                   /* set read length output parameter */

  return HTTP_SUCCESS;
}
```

http_resp_close_cb

In this implementation this function marks the user file free.

```
 static t_http_ret http_resp_close_cb ( const t_http_req_info * const p_req_info
                                      , const uint32_t hdl )
{
  HCC_UNUSED_ARG( p_req_info );

  g_user_file[hdl].b_used = FALSE;

  return HTTP_SUCCESS;
}
```

http_dynvar_cb

This function is the dynamic variable handler.

```
#if ( HTTP_DYNAMIC_VARIABLES != 0 )
static t_http_ret http_dynvar_cb ( const t_http_req_info * const p_req_info
                                  , const char_t * const p_var_name
                                  , char_t * const p_var_value )
{
  t_http_ret  ret_val;
  HCC_UNUSED_ARG( p_req_info );
  ret_val = HTTP_SUCCESS;
  if ( strcmp( p_var_name, "my_str_0" ) == NULL )
  {
    strcpy( p_var_value, "--MyString0--" );
  }
  else if ( strcmp( p_var_name, "call_cnt" ) == NULL )
  {
    sprintf( p_var_value, "--%d--", g_call_cnt );
    ++g_call_cnt;
  }
  else
  {
    ret_val = HTTP_ERROR;
  }
  return ret_val;
} /* http_dynvar_cb */
#endif /* if ( HTTP_DYNAMIC_VARIABLES != 0 ) */
```

6.5 User GET Using the FAT File System

This example demonstrates the GET method interfaced to a FAT file system. Here all requested URIs are directly read from the file system.

Demo File

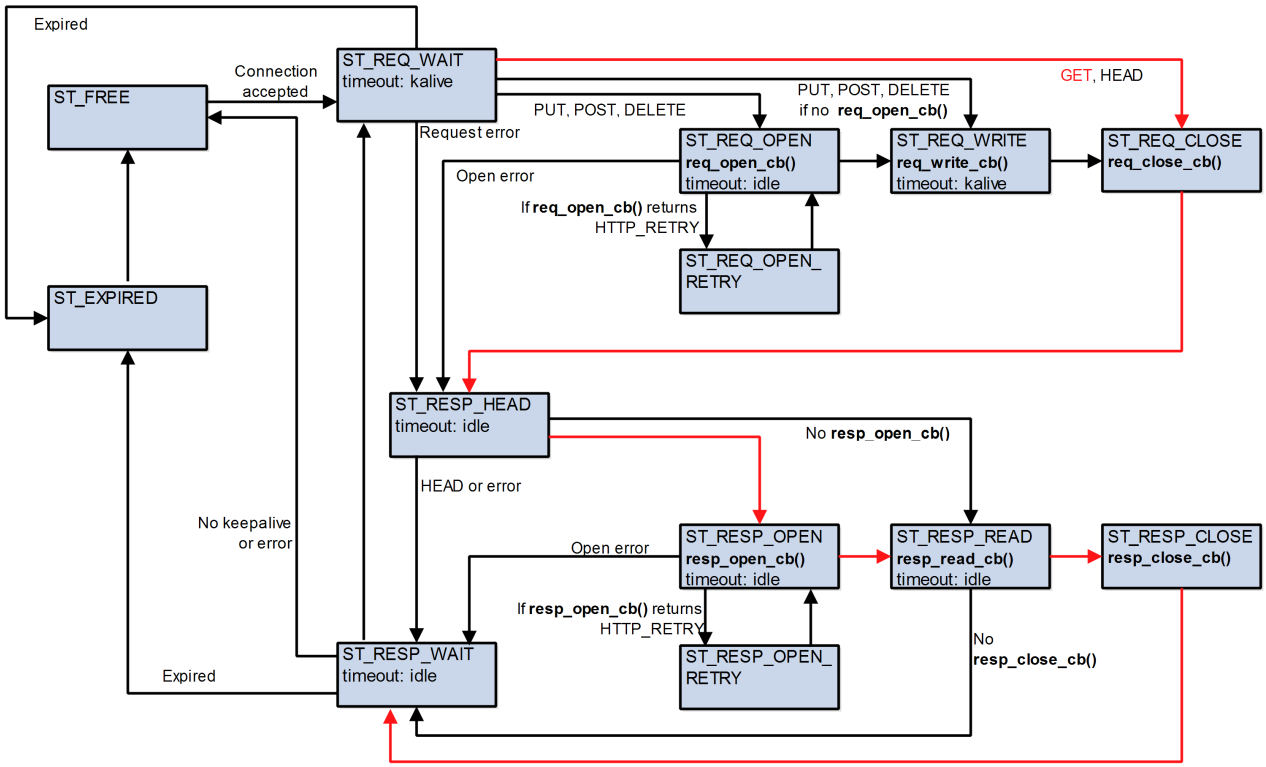
This implementation is covered by the file **http_user_get_fs.c**.

Unused Callbacks

The **http_req_open_cb()** and **http_req_write_cb()** callbacks are not used in this implementation; these can only handle GET requests.

State Diagram

The red lines on this diagram show the GET process. In this example both **resp_open_cb()** and **resp_close_cb()** callbacks are available.



Defines and Typedefs

```
/* define the max. no. files the module is allowed to use */
#define MAX_FILES        2
/* File cache size */
#define FILE_CACHE_SIZE  4096
/* HTTP root directory */
#define HTTP_ROOT_DIR  "/"

typedef struct
{
  F_FILE  * uf_file;
#if FILE_CACHE_SIZE
  uint8_t   uf_buf[FILE_CACHE_SIZE];
  uint32_t  uf_buf_len;
  uint32_t  uf_buf_pos;
#endif
} t_user_file;
```

Local Variables

```
/* callback descriptor */
static const t_http_cb_dsc  g_http_cb_dsc =
{
  http_req_open_cb
  , http_req_write_cb
  , http_req_close_cb
  , http_resp_open_cb
  , http_resp_read_cb
  , http_resp_close_cb
#if ( HTTP_DYNAMIC_VARIABLES != 0 )
  , (t_http_dynvar_cb)NULL
#endif
};

/* file handles */
static t_user_file g_user_file[MAX_FILES];
```

Functions for User and File Handling

http_user_init

In this implementation this function initializes the HTTPS user module.

```
t_http_ret http_user_init ( void )
{
  t_http_ret  ret_val;
  uint16_t    pos;
  for ( pos = 0u ; pos < MAX_FILES ; pos++ )
  {
    g_user_file[pos].uf_file = (F_FILE *)NULL;
#if FILE_CACHE_SIZE
    g_user_file[pos].uf_buf_len = 0u;
    g_user_file[pos].uf_buf_pos = 0u;
#endif
  }
  ret_val = http_register_cb( &g_http_cb_dsc );
  return ret_val;
}
```

Callbacks

Unused Callbacks

The **http_req_open_cb()** and **http_req_write_cb()** callbacks are not used in this implementation; these can only handle GET requests.

http_req_close_cb

In this implementation this function checks whether the request method is GET, gets the content type based on the URI, and gets the length of the file (identified by its URI) from the file system.

```
static t_http_ret http_req_close_cb ( const t_http_req_info * const p_req_info
                                    , const uint32_t hdl
                                    , t_http_resp_info * const p_resp_info )
{
  t_http_ret  ret_val;
  int         ret_fs;
  long        flen;
  char_t    * p_uri;
  HCC_UNUSED_ARG( hdl );
  ret_val = HTTP_ERROR;
  if ( ( ( p_req_info->hreq_method ) == HTTP_METHOD_HEAD )
      || ( ( p_req_info->hreq_method ) == HTTP_METHOD_GET ) )
  {
    /* get response content type */
    ret_val = http_get_content_type( p_req_info->p_hreq_uri
                                   , &( p_resp_info->p_hresp_ctype ) );
    if ( ret_val == HTTP_SUCCESS )
    {
      f_enterFS();
      ret_fs = f_chdir( HTTP_ROOT_DIR );
      if ( ret_fs == F_NO_ERROR )
      {
        p_uri = p_req_info->p_hreq_uri;
        if ( *p_uri == '/' )
        {
          p_uri++;       /* Skip '/' charachter */
        }

        flen = f_filelength( p_uri );  /* get file length */
        /* file available */
        if ( flen > 0 )
        {
          /* set response length */
          p_resp_info->hresp_clen = (uint32_t)flen;
        }
        else
        {
          ret_val = HTTP_ERR_NOT_FOUND;
        }
      }
      else
      {
        ret_val = HTTP_ERROR;
      }
    }
  }
  return ret_val;
}
```

http_resp_open_cb

In this implementation this function searches for a free file handle. If a handle is free, it tries to open the file (identified by its URI); otherwise, it tells the server to retry later.

```
static t_http_ret http_resp_open_cb ( const t_http_req_info * const p_req_info
                                    , const uint32_t hdl
                                    , uint32_t * const p_new_hdl )
{
  t_http_ret  ret_val;
  int         ret_fs;
  uint16_t    pos;
  char_t    * p_uri;
  HCC_UNUSED_ARG( hdl );
  ret_val = HTTP_RETRY;
  for ( pos = 0u
       ; ( pos < MAX_FILES ) && ( ( g_user_file[pos].uf_file ) != NULL )
       ; pos++ )
  {
  }
  if ( pos < MAX_FILES )
  {
    f_enterFS();

    ret_fs = f_chdir( HTTP_ROOT_DIR );
    if ( ret_fs == F_NO_ERROR )
    {
      p_uri = p_req_info->p_hreq_uri;
      if ( *p_uri == '/' )
      {
        p_uri++;       /* Skip '/' charachter */
      }

      g_user_file[pos].uf_file = f_open( p_uri, "r" );
      if ( ( g_user_file[pos].uf_file) != NULL )
      {
        *p_new_hdl = pos;
        ret_val = HTTP_SUCCESS;
      }
      else
      {
        ret_val = HTTP_ERROR;
      }
    }
    else
    {
      ret_val = HTTP_ERROR;
    }
  }
  return ret_val;
}
```

http_resp_read_cb

In this implementation this function simply fills the buffer with the data read from a file opened by **http_resp_open_cb()**.

```
static t_http_ret http_resp_read_cb ( const t_http_req_info * const p_req_info
                                    , const uint32_t hdl
                                    , uint8_t * const p_buf
                                    , const uint16_t buf_len
                                    , uint16_t * const p_rd_len )
{
  t_user_file * uf;
#if FILE_CACHE_SIZE
  uint32_t      blen;
  uint16_t      bpos;
#endif
  HCC_UNUSED_ARG( p_req_info );
  uf = &( g_user_file[hdl] );
#if FILE_CACHE_SIZE
  bpos = 0u;
  do
  {
    if ( ( uf->uf_buf_pos ) == ( uf->uf_buf_len ) )
    {
      uf->uf_buf_len = (uint32_t)f_read( &( uf->uf_buf[0] ), 1, FILE_CACHE_SIZE, uf->uf_file );
      uf->uf_buf_pos = 0u;
    }

    blen = ( uf->uf_buf_len ) - ( uf->uf_buf_pos );
    if ( blen > (uint32_t)( buf_len - bpos ) )
    {
      blen = buf_len - bpos;
    }

    if ( blen > 0u )
    {
      memcpy( p_buf + bpos, &( uf->uf_buf[uf->uf_buf_pos] ), blen );
      uf->uf_buf_pos += blen;
      bpos += blen;
    }
  }
  while ( ( bpos < buf_len )
         && ( blen > 0u ) );

  *p_rd_len = (uint16_t)bpos;
#else
  *p_rd_len = (uint16_t)f_read( p_buf, 1, buf_len, uf->uf_file );
#endif

  return HTTP_SUCCESS;
}
```

http_resp_close_cb

In this implementation this function closes the file opened by by **http_resp_open_cb()**.

```
static t_http_ret http_resp_close_cb ( const t_http_req_info * const p_req_info
                                     , const uint32_t hdl )
{
  HCC_UNUSED_ARG( p_req_info );
  f_close( g_user_file[hdl].uf_file );

  g_user_file[hdl].uf_file = (F_FILE *)NULL;
#if FILE_CACHE_SIZE
  g_user_file[hdl].uf_buf_len = 0u;
  g_user_file[hdl].uf_buf_pos = 0u;
#endif
  return HTTP_SUCCESS;
}
```