emWeb

CPU and TCP/IP stack independent web server

User Guide & Reference Manual

Document: UM07002
Software Version: 3.40
Revision: 0
Date: April 2, 2020

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions. SEGGER makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. SEGGER specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel.	+49 2173-99312-0
Fax.	+49 2173-99312-28
E-mail:	ticket_emnet@segger.com*
Internet:	www.segger.com

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please inform us and we will try to assist you as soon as possible. Contact us for further information on topics or functions that are not yet documented.

Print date: April 2, 2020

Software	Revision	Date	By	Description
3.40	0	200402	OO	Chapter "Web server (Add-on)" updated. • IP_WEBS_ConfigStaticEncodedFiletypes() added. • IP_WEBS_Send204NoContent() added. • Progress status WEBS_PROGRESS_STATUS_METHOD_URI_VER_PARSED added to structure WEBS_PROGRESS_INFO . • Added missing "paVFiles" member to WEBS_APPLICATION structure.
3.30b	0	181026	OO	Chapter "Web server (Add-on)" updated. • Progress status WEBS_PROGRESS_STATUS_METHOD_URI_VER_PARSED added to structure WEBS_PROGRESS_INFO . • Added missing "paVFiles" member to WEBS_APPLICATION structure. Chapter "WebSocket (Add-on)" updated. • Link to IP_WEBS_WEBSOCKET_AddHook() added for better explanation of samples.
3.30	0	180704	OO	Chapter "Web server (Add-on)" updated. • IP_WEBS_SetErrorPageCallback() added. • IP_WEBS_AddProgressHook() added. • IP_WEBS_HEADER_AddFieldHook() added. • IP_WEBS_HEADER_CopyData() added. • IP_WEBS_HEADER_GetFindToken() added. • IP_WEBS_HEADER_SetCustomFields() added.
3.20	0	170622	OO	Chapter "Web server (Add-on)" updated. • Minor changes.
3.16	0	170323	OO	Chapter "Web server (Add-on)" updated. • IP_WEBS_ConfigFindGZipFiles() added.
3.14	0	161223	OO	Chapter "WebSocket (Add-on)" added. Chapter "Web server (Add-on)" updated. • NumBytesFullUriBuf added to IP_WEBS_ConfigBufSizes() WEBS_BUFFER_SIZES sample. • Extended return values for IP_WEBS_Process[Last][Ex](). • IP_WEBS_WEBSOCKET_AddHook() added. • Structure IP_WEBS_WEBSOCKET_API added.
3.12	0	161010	OO	Chapter "Web server (Add-on)" updated. • Digest authentication support added. • IP_WEBS_GetProtectedPath() added. • IP_WEBS_UseAuthDigest() added. • IP_WEBS_AUTH_DIGEST_CalcHA1() added. • IP_WEBS_AUTH_DIGEST_GetURI() added.
3.10	0	160912	OO	Chapter "Web server (Add-on)" updated. • IP_WEBS_AddPreContentOutputHook() updated. • IP_WEBS_SendFormattedString() added.
3.08a	0	160712	OO	Chapter "Web server (Add-on)" updated. • IP_WEBS_CountRequiredMem() added. • IP_WEBS_SetUploadFileSystemAPI() added. • IP_WEBS_SetUploadMaxFileSize() added. • Updated structure IP_WEBS_FILE_INFO.
3.08	0	160630	OO	Chapter "Web server (Add-on)" updated. • IP_WEBS_AddPreContentOutputHook() added. • IP_WEBS_ConfigUploadRootPath() added. • IP_WEBS_Init() description in API table updated. • IP_WEBS_SendLocationHeader() added. • IP_WEBS_SetHeaderCacheControl() added.
3.04	0	160316	OO	Chapter "Web server (Add-on)" updated. • IP_WEBS_AddRequestNotifyHook() added.
3.02	0	151125	OO	Chapter "Web server (Add-on)" updated. • IP_WEBS_UseRawEncoding() added. • IP_WEBS_GetConnectInfo() added.
3.00	0	150813	OO	Chapter "Web server (Add-on)" updated. • IP_WEBS_AddUpload() added.

Software	Revision	Date	By	Description
				• IP_WEBS_ConfigBufSizes() added. • IP_WEBS_ConfigRootPath() added. • IP_WEBS_Flush() added. • IP_WEBS_Init() added. • IP_WEBS_ProcessEx() added. • IP_WEBS_ProcessLastEx() added. • IP_WEBS_SendHeaderEx() added.
2.12c	0	130515	OO	Chapter "Web server (Add-on)" updated. • IP_WEBS_GetURI() added. • IP_WEBS_Reset() added.
2.12	0	130312	OO	Chapter "Web server (Add-on)" updated. • IP_WEBS_AddVFileHook() updated. • IP_WEBS_Redirect() added. • IP_WEBS_StoreUserContext() added. • IP_WEBS_RetrieveUserContext() added. • IP_WEBS_GetDecodedStrLen() added. • IP_WEBS_METHOD_* API added.
2.10	0	120913	OO	Chapter "Web server (Add-on)" updated. • Information regarding file uploads added. • More detailed description about multiple connections added. • IP_WEBS_AddFileTypeHook() added. • IP_WEBS_AddVFileHook() added. • IP_WEBS_ConfigSendVFileHeader() added. • IP_WEBS_ConfigSendVFileHookHeader() added. • IP_WEBS_GetParaValuePtr() added. • IP_WEBS_SendHeader() added.
1.54b	0	090603	SK	Chapter "Web server (Add-on)" updated. • "IP_WEBS_Process()" updated. • "IP_WEBS_ProcessLast()" added. • "IP_WEBS_OnConnectionLimit()" updated.
1.54a	1	090520	SK	Chapter "Web server (Add-on)" updated. • Section "Changing the file system type" added. • Section "IP_WEBS_SetFileInfoCallback" updated.
1.54a	0	090508	SK	Chapter "Web server (Add-on)" updated. • IP_WEBS_GetNumParas() added. • IP_WEBS_GetParaValue() added. • IP_WEBS_DecodeAndCopyStr() added. • IP_WEBS_DecodeString() added. • IP_WEBS_SetFileInfoCallback() added. • IP_WEBS_CompareFilenameExt() added. • Section "Dynamic content" added • Section "Common Gateway interface" moved into section "Dynamic content".
1.42	0	080821	SK	Chapter "Web server (Add-on)": • List of valid values for CGI parameter and values added.
1.30	1	080610	SK	Chapter "Web server (Add-on)" section "Resource usage" added
1.30	0	080423	SK	Chapter "Web server (Add-on)" updated.
1.00	1	071002	SK	Product name changed to "emWeb":
1.00	0	070927	SK	Initial version.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler).
- The C programming language.
- The target processor.
- DOS command line.

If you feel that your knowledge of C is not sufficient, we recommend *The C Programming Language* by Kernighan and Richie (ISBN 0--13--1103628), which describes the standard in C programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

Table of contents

Chapter 1

Introduction

The emWeb Web server is an optional extension to emNet. The Web server can be used with emNet or with a different TCP/IP stack. All functions that are required to add a Web server task to your application are described in this chapter.

1.1 emNet Web server

The emNet Web server is an optional extension which adds the HTTP protocol to the stack. It combines a maximum of performance with a small memory footprint. The Web server allows an embedded system to present Web pages with dynamically generated content. It comes with all features typically required by embedded systems: multiple connections, authentication, forms and low RAM usage. RAM usage has been kept to a minimum by smart buffer handling.

The Web server implements the relevant parts of the following Request For Comments (RFC).

RFC#	Description
[RFC 1945]	HTTP - Hypertext Transfer Protocol -- HTTP/1.0 Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc1945.txt*
[RFC 2069]	An Extension to HTTP : Digest Access Authentication Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc2069.txt*
[RFC 2616]	HTTP - Hypertext Transfer Protocol -- HTTP/1.1 Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc2616.txt*

The following table shows the contents of the emNet root directory:

Directory	Content
`.\Application\`	Contains the example application to run the Web server with emWeb. The standard application consists of two files. `IP_WebserverSample.c` and `Webserver_DynContent.c`. `IP_WebserverSample.c` fits for the most applications without modifications. `Webserver_DynContent.c` includes the dynamic parts of our sample application, like virtual files, CGI functions, etc.
`.\Config\`	Contains the Web server configuration file. Refer to *Web server configuration* on page 66 for detailed information.
`.\Doc\`	Contains the emWeb manual.
`.\IP\`	Contains the Web server sources, `IP_Webserver.c`, `IP_Webserver.h` and `IP_UTIL_BASE64.c`, `IP_UTIL.h`.
`.\SEGGER\`	Contains SEGGER helper functions.
`.\Shared\IP\Application\`	Contains the file `Webserver_DynContent.c` to run the Web server with emWeb.
`.\Shared\IP\IP_FS\`	Contains the sources for the file system abstraction layer (emFile, Windows and Linux) and the read-only file system. Refer to *File system abstraction layer* on page for detailed information.
`.\Windows\IP\Webserver\`	Contains the source, the project files and an executable to run emNet Web server on a Microsoft Windows host. Refer to *Using the Web server sample* on page 18 for detailed information.
`.\Windows\IP\UDPDiscoverGUI\`	Contains UDP Discover tool to find the target in network.

1.2 Feature list

- Low memory footprint.
- Dynamic Web pages.
- Authentication supported (Basic and Digest).
- Forms: POST and GET support.
- Multiple connections supported.
- JavaScript supported.
- AJAX supported.
- SSE supported.
- REST supported.
- r/o file system included.
- HTML to C converter included.
- Independent of the file system: any file system can be used.
- Independent of the TCP/IP stack: any stack with sockets can be used.
- Demo with authentication, various forms, dynamic pages included.
- Project for executable on PC for Microsoft Visual Studio included.

1.3 Requirements

TCP/IP stack

The emWeb Web server requires a TCP/IP stack. It is optimized for emNet, but any RFC-compliant TCP/IP stack can be used. The shipment includes a Win32 simulation, which uses the standard Winsock API and an implementation which uses the socket API of emNet.

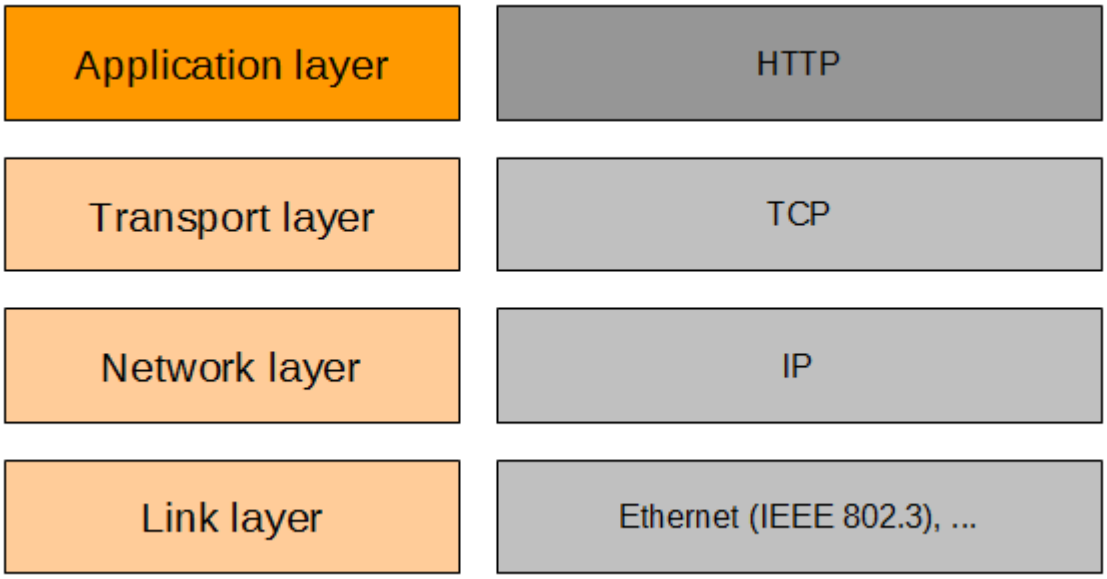
Multi tasking

The Web server needs to run as a separate thread. Therefore, a multi tasking system is required to use the emWeb Web server.

Chapter 2

HTTP backgrounds

It is a communication protocol originally designed to transfer information via hypertext pages. The development of HTTP is coordinated by the IETF (Internet Engineering Task Force) and the W3C (World Wide Web Consortium). The current protocol version is 1.1.



2.1 HTTP communication basics

HTTP is a challenge and response protocol. A client initiates a TCP connection to the Web server and sends a HTTP request. A HTTP request starts with a method token. *[RFC 2616]* defines 8 method tokens. The method token indicates the method to be performed on the requested resource. emWeb Web server supports all methods which are typically required by an embedded Web server.

HTTP method	Description
GET	The GET method means that it retrieves whatever information is identified by the Request-URI.
HEAD	The HEAD method means that it retrieves the header of the content which is identified by the Request-URI.
POST	The POST method submits data to be processed to the identified resource. The data is included in the body of the request.

The following example shows parts of a HTTP session, where a client (for example, `192.168.1.75`) asks the emWeb Web server for the hypertext page `example.html`. The request is followed by a blank line, so that the request ends with a double newline, each in the form of a carriage return followed by a line feed.

```
GET /example.html HTTP/1.1
Host: 192.168.1.75
```

The first line of every response message is the Status-Line, consisting of the protocol version followed by a numeric status code. The Status-Line is followed by the content-type, the server, expiration and the transfer-encoding. The server response ends with an empty line, followed by length of content that should be transferred. The length indicates the length of the Web page in bytes.

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: emWeb
Expires: THU, 26 OCT 1995 00:00:00 GMT
Transfer-Encoding: chunked

A3
```

Thereafter, the Web server sends the requested hypertext page to the client. The zero at the end of the Web page followed by an empty line signalizes that the transmission of the requested Web page is complete.

```
<html>
  <head>
    <title>emWeb examples</title>
  </head>
  <body>
    <center>
      <h1>Website: example.htm</h1>
    </center>
  </body>
</html>
0
```


2.2 HTTP status codes

The first line of a HTTP response is the Status-Line. It consists of the used protocol version, a status code and a short textual description of the Status-Code. The Status-Code element is a 3-digit integer result code of the attempt to understand and satisfy the request.

The first digit of the Status-Code defines the class of response. The last two digits do not have any categorization role. There are 5 values for the first digit:

- 1xx: Informational - Request received, continuing process.
- 2xx: Success - The action was successfully received, understood, and accepted.
- 3xx: Redirection - Further action must be taken in order to complete the request.
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled.
- 5xx: Server Error - The server failed to fulfill an apparently valid request.

Refer to *[RFC 2616]* for a complete list of defined status-codes. emWeb Web server supports a subset of the defined HTTP status codes. The following status codes are implemented:

Status code	Description
200	OK. The request has succeeded.
401	Unauthorized. The request requires user authentication.
404	Not found. The server has not found anything matching the Request-URI.
501	Not implemented. The server does not support the HTTP method.
503	Service unavailable. The server is currently unable to handle the request due to a temporary overloading of the server.

Chapter 3

Using the Web server sample

3.1 Overview

Ready to use examples for Microsoft Windows and enNet are supplied. Please refer to *Using the Web server Windows sample* on page 28 for the Microsoft Windows Web server.

The following emWeb examples are supplied:

Sample	Description
`IP_WebserverSample.c`	Standard Web server sample using IPv4.
`IP_WebserverSample_IPv4_IPv6_SSL.c`	Web server sample that is configurable to run both IPv4 and IPv6 with or without SSL support.
`IP_WebserverSample_IPv6.c`	Web server sample using IPv6.
`IP_WebserverSample_Secure.c`	Web server sample using IPv4 and offering SSL support.
`IP_WebserverSample_select_IPv4_IPv6.c`	Web server sample using IPv4 and IPv6.
`IP_WebserverSample_Single-Task_IPv4_IPv6.c`	Web server sample that can be used with one task and both IPv4 and IPv6.

If you use another TCP/IP stack, the samples have to be adapted.

3.2 Connection handling

Note: The following explanation does not apply to the "single task"-sample, since this sample only uses one task.

The Web server itself does not handle multiple connections. The connection handling is part of the example applications. The sample applications open a port which listens on port 80 until an incoming connection is detected in a parent task that accepts new connections (or rejects them if no more connections can be accepted).

3.2.1 Task handling

For each accepted client connection, the parent task creates a child task running `IP_WEBS_ProcessEx()` in a separated context that will then process the request of the connected client (for example a browser). This way the parent task is ready to handle further incoming connections on port 80.

Therefore the sample uses n client connections + one for the parent task.

Some browsers may open multiple connections and do not even intend to close the connection. They rather keep the connections open for further data that might be requested. To give other clients a chance, a special handling is implemented in the Web server.

The emWeb Web server has two functions for processing a connection in a child task:

- `IP_WEBS_ProcessEx()`, that allows a connection to stay open even after all data has been sent from the target. The connection will stay open as long as the client does not close it.
- `IP_WEBS_ProcessLastEx()`, that will close the connection once the target has sent all data requested. This is used by the Web server sample for the last free connection available. This ensures that at least one connection will be available after it has been served to accept further clients.

3.2.2 TCP Backlog handling

In addition to available connections that can be served directly, a feature called "backlogging" can be used.

This means additional connections will be accepted (SYN/ACK is sent from target) but not yet processed. They will be processed as soon as a free connection becomes available once a child task has served the clients request and has been closed. Connections in backlog will be kept active until the client side sends a reset due to a possible timeout in the client.

The sample applications can be used on the most targets without the need for changing any of the configuration flags. The server processes up to 22 connections using the default configuration.

Note: 22 connections means that the target can handle up to 22 clients in parallel, if every client uses only one connection. Because a single Web browser often attempts to open more than one connection to a Web server to request the files (.gif, .jpeg, etc.) which are included in the requested Web page, the number of possible parallel connected clients is less than the number of possible connections.

The 22 connections split into 20 connections that are available to be kept in the backlog of a socket (which means that up to 20 connections wait to be fetched by the application with an accept()) and up to 2 connections currently processed.

Every connection is handled in a separate task. Therefore, the Web server uses up to three tasks in the default configuration, one task which listens on port 80 and accepts connections and two tasks to process the accepted connections. To modify the number of connections, only the macro `MAX_CONNECTIONS` has to be modified.

The most of the supplied sample Web pages include dynamic content, refer to *Dynamic content* on page 29 for detailed information about the implementation of dynamic content.

3.2.3 File system

The example applications use a read-only file system to make Web pages available. Refer to *File system abstraction layer* on page for detailed information about the read-only file system.

3.3 Running the Web server example on target hardware

The emWeb Web server sample application should always be the first step to check the proper function of the Web server with your target hardware.

All source files located in the following directories (and their subdirectories) have to be added to your project for the Web server to run successfully. You also need to update the include paths.

- `Application\`
- `Config\`
- `IP\`
- `SEGGER\`

To use the sample websites shipped with your Web server, add all files located in these subdirectories and also update the include paths.

- `Shared\IP\Application\`
- `Shared\IP\IP_FS\`
- `Shared\IP\IP_FS\emFile\`
- `Shared\IP\IP_FS\FS_RO\`
- `Shared\IP\IP_FS\FS_RO\Generated\`
- `Shared\IP\IP_FS\FS_RO_2018\`
- `Shared\IP\IP_FS\FS_RO_2018\Generated\`

It is recommended that you keep the provided folder structure.

3.3.1 Setting up a Web server sample

Procedure to follow

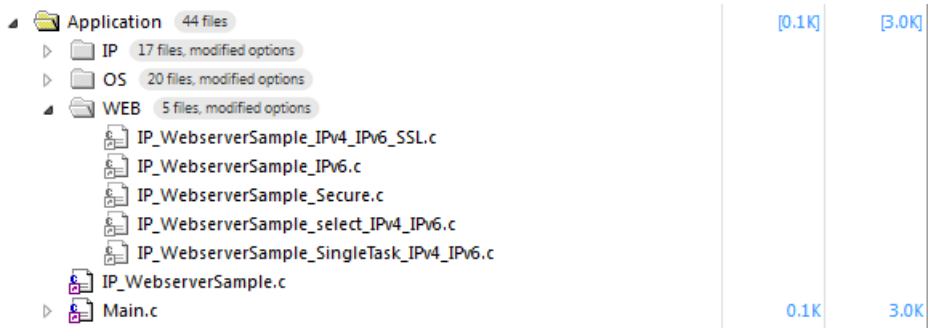
Setting up a Web server sample is an easy process that consists of the following steps:

- Step 1: Select a Web server application sample
- Step 2: Choose a sample website
- Step 3: Compile the project and start the Web server

3.3.1.1 Step 1: Select a Web server application sample

The first step is to select one of the samples that are shipped to run the Web server. There are a few different Web server samples that come with emWeb, to understand the difference between each sample, refer to *Using the Web server sample* on page 18. To keep it simple, in this example the standard Web server sample `IP_WebserverSample.c` will be used.

Exclude all application files from your project build, except for `Main.c` and `IP_WebserverSample.c`. Your project can then look like this:



3.3.1.2 Step 2: Choose a sample website

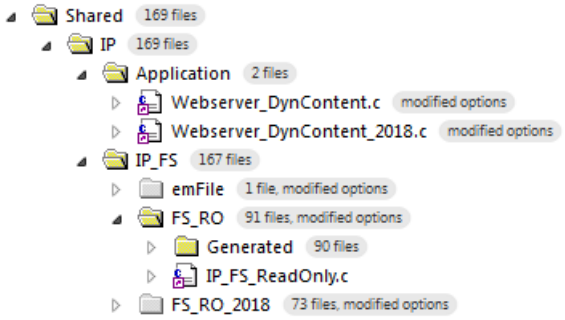
There currently are two website samples to choose from: the standard sample and the newer 2018 sample. For the sake of simplicity, we will select the standard website sample.

First, make sure you have included the dynamic content files into your project. There are two of those files, `Webserver_DynContent_2018.c` and `Webserver_DynContent.c`. You can easily change between them by setting the `WEBS_USE_SAMPLE_2018` define in your Web server configuration. You will find more on that in the chapter *Web server configuration* on page 66.

Second, you also need to choose the correct file system. There are several file system configurations that you can choose from. Two of them being read-only file systems, one for the standard Web server and one for the 2018 Web server.

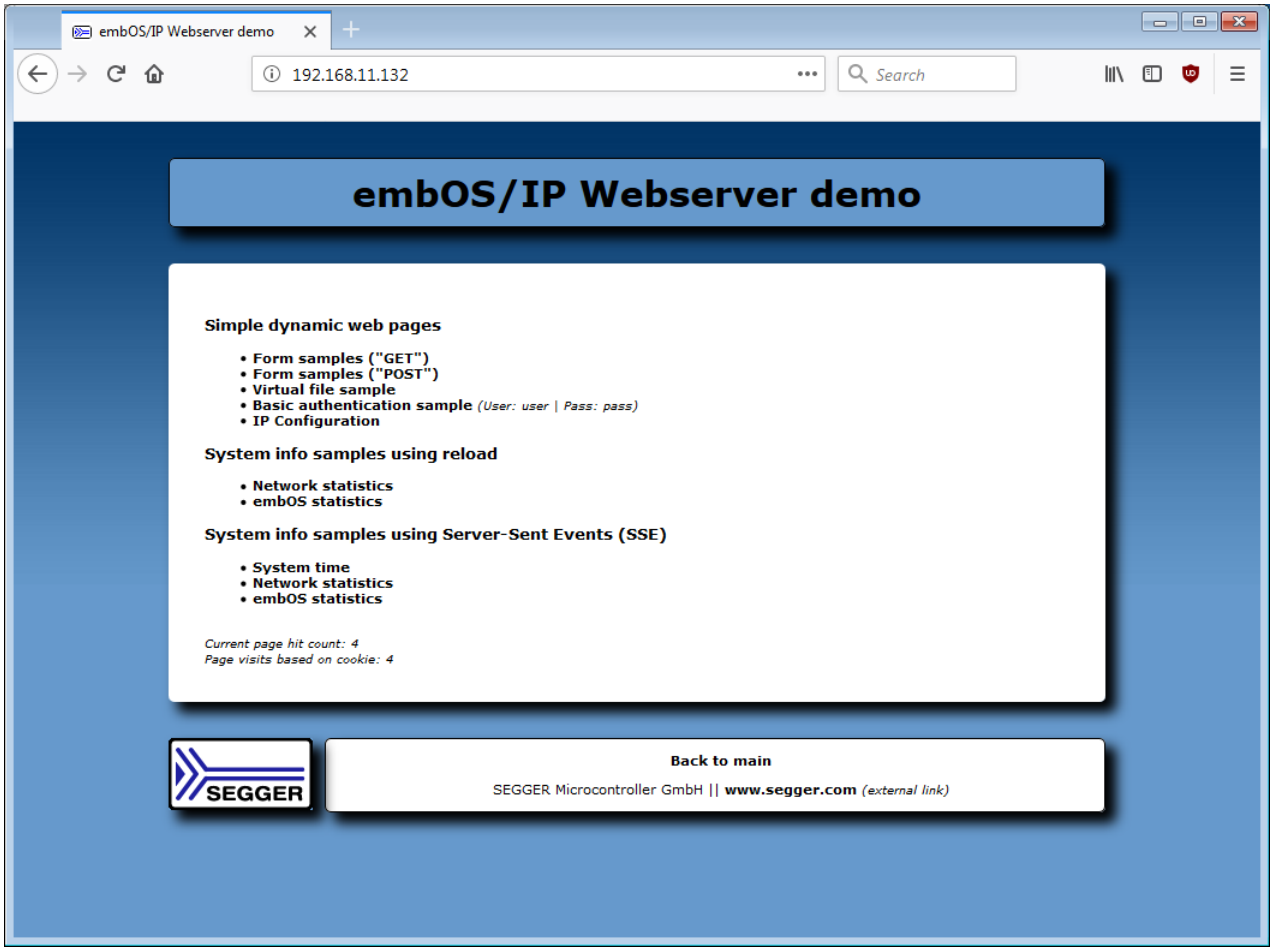
Using a read-only file system is the simplest way. If you want to use a real file system there is also the emFile abstraction layer to use on a target. How to set up the emFile file system is explained in *Changing the file system type* on page 26.

In this example, we have to use the standard read-only file system. Exclude all file system folders from your project build except `FS_RO`.



3.3.1.3 Step 3: Build the project and test it

Build the project, it should compile without errors and warnings. After you run your project, you may test the Web server by entering the IP of your target into your browser.



3.4 Changing the file system type

By default, the Web server uses the supplied read-only file system. If a real file system like emFile should be used to store the Web pages, you have to modify the function `_WebServerParentTask()` of the example `IP_WebserverSample.c`.

Excerpt from `IP_WebserverSample.c`:

```
/*********************************************************************
*
*       _WebServerParentTask
*
*/
static void _WebServerParentTask(void) {
  struct sockaddr    Addr;
  struct sockaddr_in InAddr;
  U32  Timeout;
  long hSockListen;
  long hSock;
  int  AddrLen;
  int  i;
  int  t;
  int  t0;
  int  r;
  WEBS_BUFFER_SIZES BufferSizes;

  Timeout = IDLE_TIMEOUT;
  IP_WEBS_SetFileInfoCallback(&_pfGetFileInfo);
  //
  // Assign file system
  //
  _pFS_API = &IP_FS_ReadOnly;  // To use a a real filesystem like emFile
                               // replace this line.
//  _pFS_API = &IP_FS_FS;        // Use emFile
```

The usage of the read-only file system is configured with the following line:

```
_pFS_API = &IP_FS_ReadOnly;
```

To use emFile as file system for your Web server application, add the emFile abstraction layer `IP_FS_FS.c` to your project and change the line to:

```
_pFS_API = &IP_FS_FS;
```

Make sure to add all Web server files to the root directory of your storage medium and insert the medium into your target.

In `IP_FS_emFile.c` you will find three configurable defines, which are the following:

- `MAX_PATH`, maximum length of file paths. Default is **128**.
- `IP_FS_VOLUMENAME`, label of the volume to be used. Default is **`NULL`**. If set to `NULL` all volumes configured in emFile will be visible.
- `IP_FS_ROOT_PATH`, path to use as root directory for the file system. Default is **`NULL`**. If set to `NULL`, no specific root folder is selected so the volume itself is the root directory.

Refer to *File system abstraction layer* on page for detailed information about the emFile and read-only file system abstraction layer.

3.4.1 Long file name support

The FAT file system was not designed for long file name (LFN) support, limiting file names to twelve characters (8.3).

Therefore when using the emFile abstraction layer and depending upon LFN support, it is important to note that this has to be activated manually. Also note that this requires LFN as an optional package for emFile.

To activate LFN support, simply add the following statement to the routine `_InitIfRequired()`. You will find this routine in the emFile abstraction layer file `IP_FS_emFile.c`.

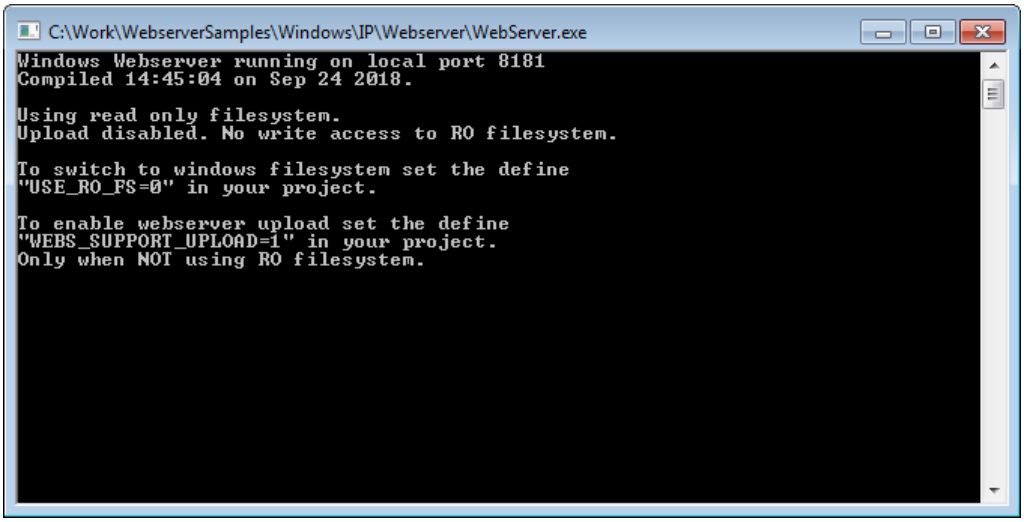
```
FS_FAT_SupportLFN();
```


3.5 Using the Web server Windows sample

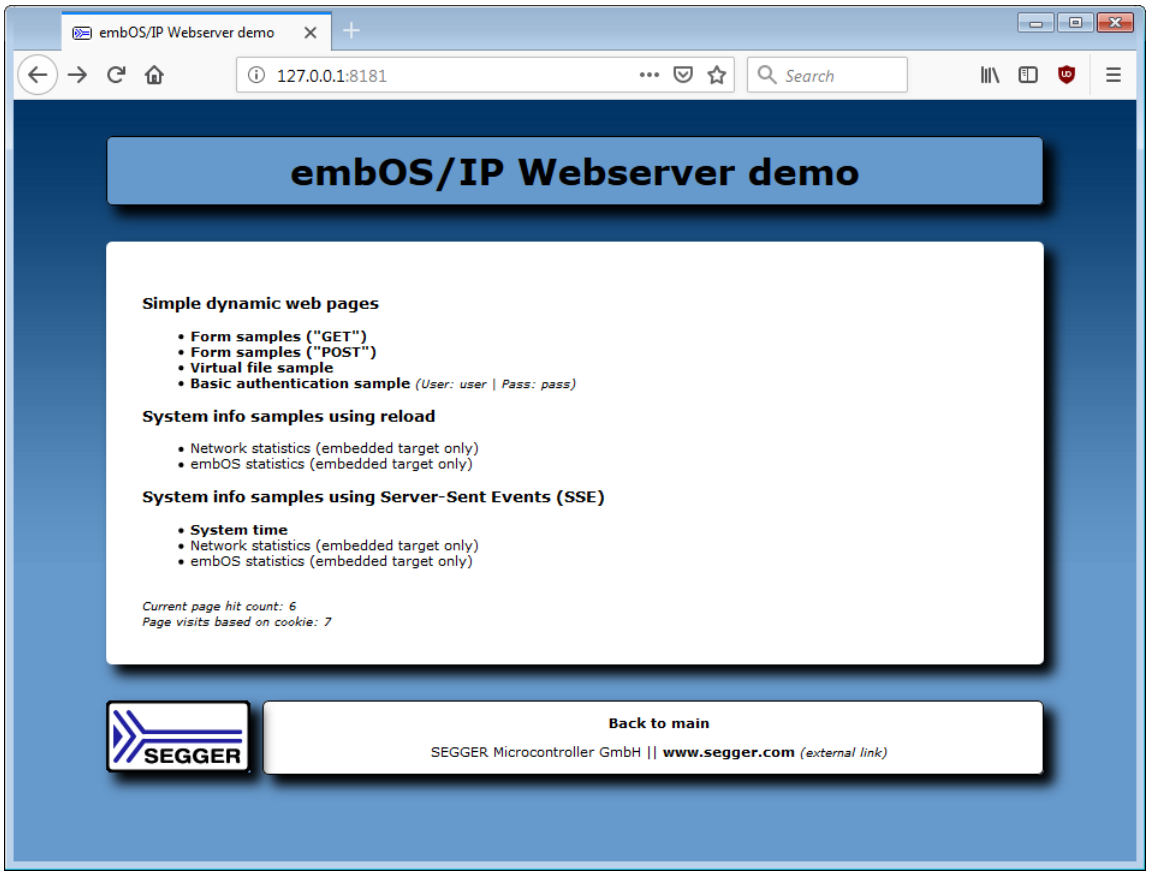
If you have MS Visual C++ 6.00 or any later version available, you will be able to work with a Windows sample project using emWeb Web server. If you do not have the Microsoft compiler, an precompiled executable of the Web server is also supplied.

Building the Web server sample program

Open the workspace `Start_Webserver.dsw` with MS Visual Studio (for example, double-clicking it). There is no further configuration necessary. You should be able to build the application without any error or warning message.



The server uses the IP address of the host PC on which it runs. Open a Web browser and connect by entering the IP address of the host (`127.0.0.1`) to connect to the Web server. The port being used is `8181`.



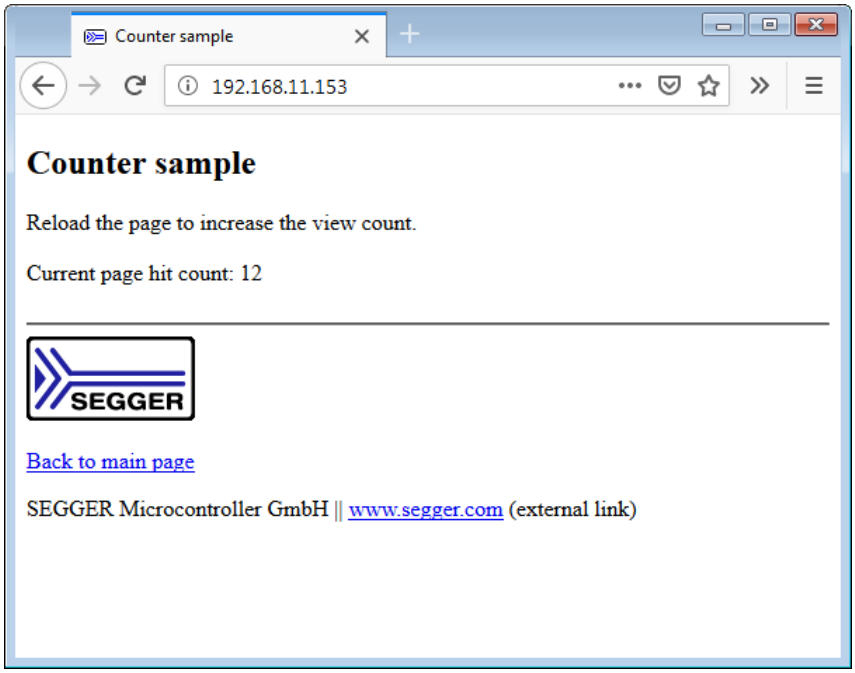
Chapter 4

Dynamic content

emWeb supports different approaches to implement dynamic content in your Web server application. A Common Gateway Interface (CGI) like interface for static HTML pages with dynamic elements and virtual files which are completely generated from the application.

4.1 Common Gateway Interface (CGI)

A Common Gateway Interface (CGI) like interface is used to implement dynamic content in Web pages. Every Web page will be parsed by the server each time a request is received. The server searches the Web page for a special tag. In the default configuration, the searched tag starts with `<!—-#exec cgi=“` and ends with `”—->`. The tag will be analyzed and the parameter will be extracted. This parameter specifies a server-side command and will be given to the user application, which can handle the command. The following screenshot shows the example page `index.htm`.



The HTML source for the page includes the following line:

```
<!--#exec cgi="Counter"-->
```

When the Web page is requested, the server parses the tag and the parameter `Counter` is searched for in an array of structures of type `WEBS_CGI`. The structure includes a string to identify the command and a pointer to the function which should be called if the parameter is found.

```
typedef struct {
  const char * sName;   // e.g. "Counter"
  void  (*pf)(WEBS_OUTPUT* pOutput, const char* sParameters, const char* sValue);
} WEBS_CGI;
```

In the example, `Counter` is a valid parameter and the function `_callback_ExecCounter` will be called. You need to implement the `WEBS_CGI` array and the callback functions in your application.

```
static const WEBS_CGI _aCGI[] = {
  {"Counter"   , _callback_ExecCounter  },
  {NULL}
};
```

`_callback_ExecCounter()` is a simple example of how to use the CGI feature. It returns a string that includes the value of a variable which is incremented with every call to `_callback_ExecCounter()`.

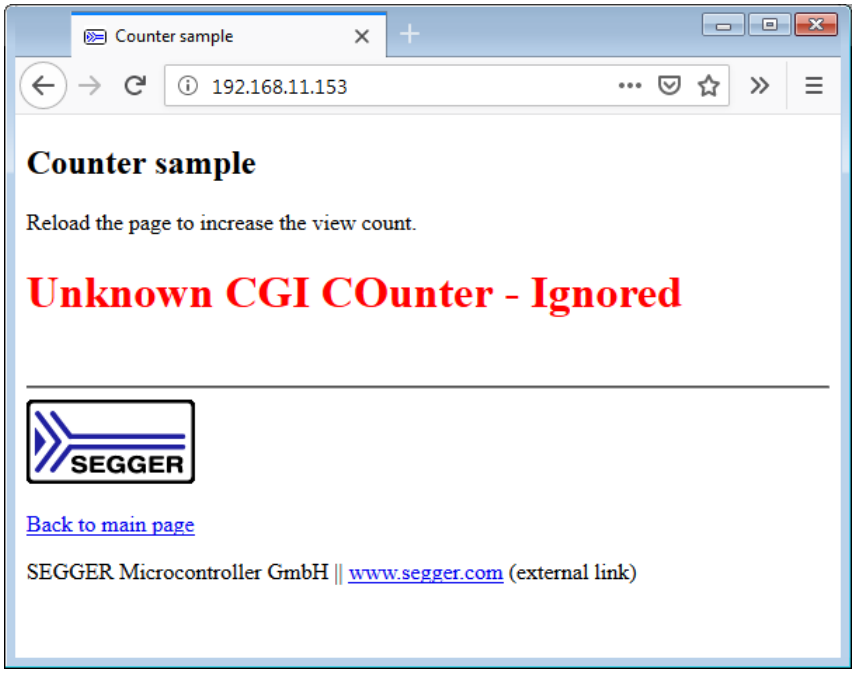
```
static void _callback_ExecCounter( WEBS_OUTPUT* pOutput,
                                   const char*  sParameters,
                                   const char*  sValue ) {
```



```
  char ac[80];

  WEBS_USE_PARA(sParameters);
  WEBS_USE_PARA(sValue);
  _Cnt++;
  SEGGER_snprintf(ac,
            sizeof(ac),
            "<br>Current page hit count: %lu",
            _Cnt);
  IP_WEBS_SendString(pOutput, ac);
}
```

If the Web page includes the CGI tag followed by an unknown command (for example, a typo like `COounter` instead of `Counter` in the source code of the Web page) an error message will be sent to the client.



4.1.1 Add new CGI functions to your Web server application

To define new CGI functions, three things have to be done.

1. Add a new command name which should be used as tag to the `WEBS_CGI` structure. For example: `UserCGI`

```
static const WEBS_CGI _aCGI[] = {
  {"Counter"   ,    _callback_ExecCounter          },
  {"GetIndex",      _callback_ExecGetIndex         },
  {"UserCGI",       _callback_ExecUserCGI          },
  {NULL,            _callback_DefaultHandler       }
};
```

2. Implement the new function in your application source code.

```
static void _callback_ExecUserCGI(       WEBS_OUTPUT * pOutput,
                                  const char         * sParameters
                                  const char         * sValue ) {
  /* Add application code here */
}
```

3. Add the new tag to the source code of your Web page:

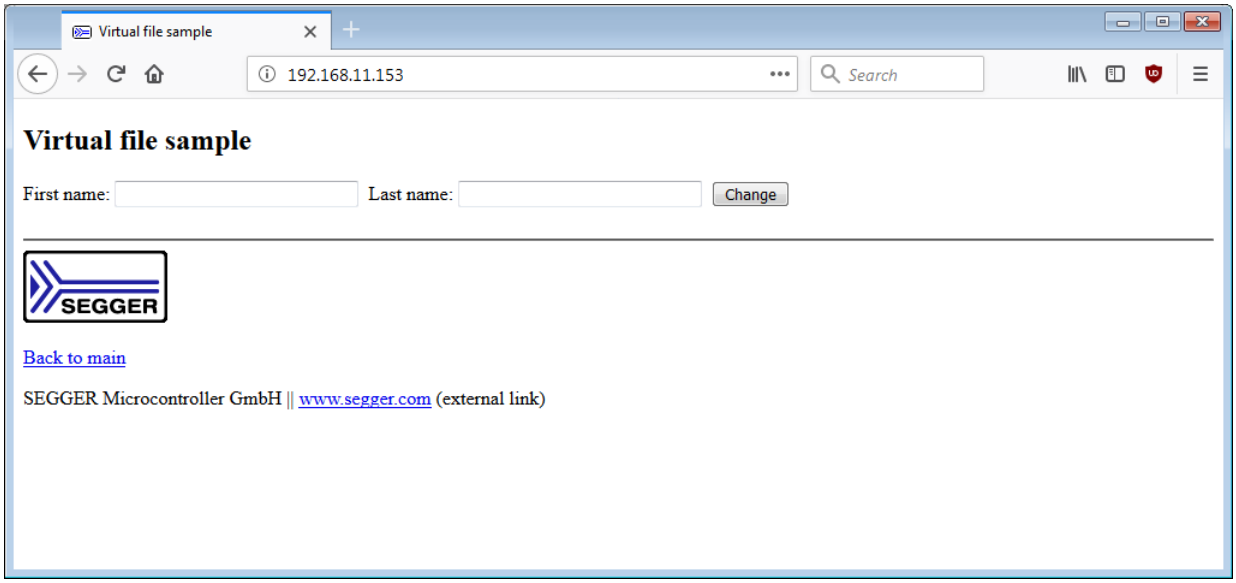
```
<!--#exec cgi="UserCGI"-->
```


4.2 Virtual files

emWeb supports virtual files. A virtual file is not a real file which is stored in the used file system. It is a function which is called instead. The function generates the content of a file and sends it to the client.

The Web server checks the extension of all requested files, the extension `.cgi` is by default used for virtual files. To change the extension that is used to detect a virtual file, refer to *IP_WEBS_SetFileInfoCallback* on page 111 for detailed information.

The emWeb Web server comes with an example (`CallVirtualFile.htm`) that requests a virtual file. The sample Web page contains a form with two input test fields, named `FirstName` and `LastName`, and a button to transmit the data to the server.



When the button on the Web page is pressed, the file Send.cgi is requested. The emWeb Web server recognizes the extension `.cgi`, checks if a virtual file with the name `Send.cgi` is defined and calls the defined function. The function in the example is `_callback_SendCGI()` and gets the string FirstName=Foo&LastName=Bar as parameter.

```
typedef struct {
  const char * sName;
  void  (*pf)(WEBS_OUTPUT * pOutput, const char * sParameters);
} WEBS_VFILES;
```

In the example, `Send.cgi` is a valid URI and the function `_callback_SendCGI()` will be called.

```
static const WEBS_VFILES _aVFiles[]  = {
  {"Send.cgi", _callback_SendCGI  },
  NULL
};
```

The virtual file `Send.cgi` gets two parameters. The strings entered in the input fields `FirstName` and `LastName` are transmitted with the URI. For example, you enter `Foo` in the first name field and Bar for last name and push the button. The browser will transmit the following string to our Web server:

```
Send.cgi?FirstName=Foo&LastName=Bar
```


You can parse the string and use it in the way you want to. In the example we parse the string and output the values on a Web page which is build from the function `_callback_CGI_Send()`.

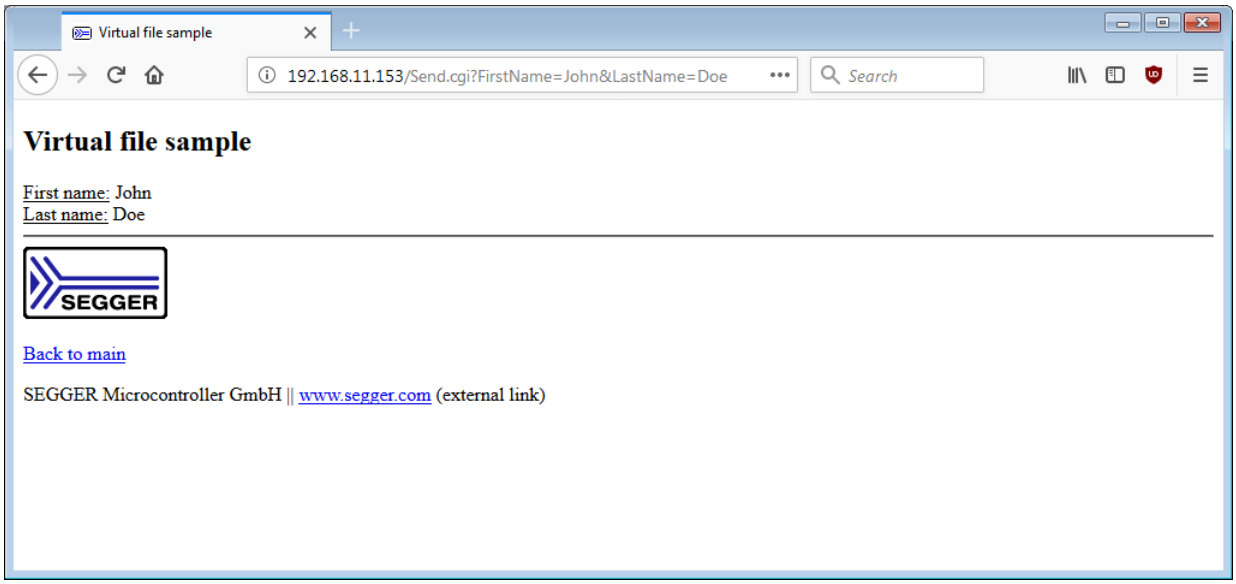
```
/*********************************************************************
*
*       WebsSample_SendPageHeader
*
*  Function description:
*    Sends the header of the virtual file.
*    The virtual files in our sample application use the same HTML layout.
*    The only difference between the virtual files is the content and that
*    each of them use an own title/heading.
*/
void WebsSample_SendPageHeader(WEBS_OUTPUT* pOutput, const char* sName) {
  IP_WEBS_SendString(pOutput, "<!DOCTYPE html><html><head><title>");
  IP_WEBS_SendString(pOutput, sName);
  IP_WEBS_SendString(pOutput, "</title>");
  IP_WEBS_SendString(pOutput, "</head><body><header><h2>");
  IP_WEBS_SendString(pOutput, sName);
  IP_WEBS_SendString(pOutput, "</h2></header>");
  IP_WEBS_SendString(pOutput, "<div>");
}

/*********************************************************************
*
*       WebsSample_SendPageFooter
*
*  Function description:
*    Sends the footer of the virtual file.
*    The virtual files in our sample application use the same HTML layout.
*/
void WebsSample_SendPageFooter(WEBS_OUTPUT* pOutput) {
  IP_WEBS_SendString(pOutput, "</div><img src=\"Logo.gif\" alt=\"Segger logo\">");
  IP_WEBS_SendString(pOutput, "<footer><p><a href=\"index.htm\">Back to main</a></p>");
  IP_WEBS_SendString(pOutput, "<p>SEGGER Microcontroller GmbH || <a href=\"http://
www.segger.com\">www.segger.com</a>");
  IP_WEBS_SendString(pOutput, "(external link)</p></footer></body></html>");
}

/*********************************************************************
*
*       _callback_CGI_Send
*/
static void _callback_CGI_Send(WEBS_OUTPUT * pOutput, const char * sParameters) {
        int    r;
  const char * pFirstName;
        int    FirstNameLen;
  const char * pLastName;
        int    LastNameLen;

  //
  // Header of the page
  //
  _SendPageHeader(pOutput, "Virtual file sample");
  //
  // Content
  //
  r = IP_WEBS_GetParaValuePtr(sParameters, 0, NULL, 0, &pFirstName, &FirstNameLen);
  r |= IP_WEBS_GetParaValuePtr(sParameters, 1, NULL, 0, &pLastName, &LastNameLen);
  if (r == 0) {
    IP_WEBS_SendString(pOutput, "<H2>First name: ");
    IP_WEBS_SendMem(pOutput, pFirstName, FirstNameLen);
    IP_WEBS_SendString(pOutput, "</H2>");
    IP_WEBS_SendString(pOutput, "<H2>Last name: ");
    IP_WEBS_SendMem(pOutput, pLastName, LastNameLen);
    IP_WEBS_SendString(pOutput, "</H2>");
  } else {
    IP_WEBS_SendString(pOutput, "<br>Error!");
  }
  //
  // Footer of the page
  //
  _SendPageFooter(pOutput);
}
```


The output of `_callback_CGI_Send()` should be similar to:



4.3 Server-Sent Events (SSE)

Server-Sent Events (SSE) are an HTML5 technology which enables a Web server to push data to Web pages over HTTP. The Web browser establishes an initial connection, which is used for the Web server updates.

The emWeb Web server supports SSE to supply the Web browser with dynamic content. A Web browser can request information via EventSource objects. The idea behind SSE is that the Web server keeps an connection to Web browser open and sends data via this connection whenever it is necessary. This means that the Web browser receives data as a stream without polling. This reduces the HTTP protocol overhead.

To subscribe to an event stream, you have create an EventSource object and pass it the URL of your stream.

```
<script>
  if(typeof(EventSource) !== "undefined") {
    var source = new EventSource("SSETime.cgi");
    source.onmessage = function(event) {
      document.getElementById("Time").innerHTML  = "<H1>" + event.data + "</H1>";
      document.getElementById("Time").innerHTML +=
      "The browser gets the system time via a Server-Sent Event (SSE).
      <br>No meta refresh (reload) required!";
    };
  } else {
    document.getElementById("Time").innerHTML =
    "Sorry, your browser does not support Server-Sent Events (SSE)...";
  }
</script>
```

The source code excerpt above creates a new EventSource object. The EventSource objects starts immediately listening for events on the given URL `SSETime.cgi`. Everytime when the Web browser will receive data, the data will be displayed on the Web page.

`SSETime.cgi` is implemented in the supplied emWeb Web server sample application (`Webserver_DynContent.c`). From the perspective of the Web server it is a virtual file. Please refer to Virtual files on page 510 for further information about virtual files. The following excerpt of `Webserver_DynContent.c` shows the implementation of the virtual file `SSETime.cgi`.

```
/*********************************************************************
*
*       _callback_CGI_SSETime
*
*  Function description:
*    Sends the system time to the Web browser every 500 ms.
*/
static void _callback_CGI_SSETime(WEBS_OUTPUT * pOutput,
                                  const char * sParameters) {
  int r;

  IP_USE_PARA(sParameters);
  //
  // Construct the SSE Header
  //
  IP_WEBS_SendHeaderEx(pOutput, NULL, "text/event-stream", 1);
  IP_WEBS_SendString(pOutput, "retry: 1000\n");
    while(1) {
      r = _SendTime((void*)pOutput);
      if (r == 0) {     // Data transmitted
        OS_Delay(500);
      } else {
        break;
      }
    }
```



```
}
```

First step to send data as an event stream is to send the MIME type `text/event-stream` to the Web browser. The Web browser attempts to reconnect to the Web server ~3 seconds after a connection is closed. You can change that timeout by sending a line beginning with `retry:`, followed by the number of milliseconds to wait before trying to reconnect. The event stream message is build in `_SendTime()`. After sending the data `OS_Delay()` suspends the task for 500ms. `_SendTime()` will be called again after the delay as long as the connection is open.

```
/*********************************************************************
*
*       _SendTime
*
*
*  Function description:
*    Sends the system time to the Web browser.
*
*  Return value:
*    0: Data successfully sent.
*   -1: Data not sent.
*    1: Data successfully sent. Connection should be closed.
*/
static int _SendTime(WEBS_OUTPUT * pOutput) {
  int r;

  //
  // Send implementation specific header to client
  //
  IP_WEBS_SendString(pOutput, "data: ");
  IP_WEBS_SendString(pOutput, "System time: ");
  IP_WEBS_SendUnsigned(pOutput, OS_GetTime32(), 10, 0);
  IP_WEBS_SendString(pOutput, "<br>");
  IP_WEBS_SendString(pOutput, "\n\n");          // End of the SSE data
  r = IP_WEBS_Flush(pOutput);
  return r;
}
```

The return value of `_SendTime()` is checked in `_callback_CGI_SSETime()`. This is necessary to prove if the data connection is still open. If the connection has been closed by the client, the endless loop will be left and the Web server will end the Web server child task.

SSE is currently not support by all popular browsers. The following Web browsers support Server-Sent Events natively.

Browser	Supported	Notes
Google Chrome	Yes	Starting with Chrome Ver. 27
MS Internet Explorer	No	—-
Mozilla Firefox	Yes	Starting with Firefox Ver. 30
Opera	Yes	Starting with Ver. 23
Safari	Yes	Starting with Ver. 5.1

Since the Microsoft Internet Explorer does not support Server-Sent Events, we use the JavaScript library `eventsource.js` in our samples to make them also usable with Microsoft Internet Explorer. `eventsource.js` can be downloaded with the following link:
https://github.com/Yaffle/EventSource/

`eventsource.js` uses the MIT license. It can be used and modified according to your needs.

SSE samples

The emWeb Web server comes with some sample Web pages to demonstrate how SSE can be used to get and visualize data from your target.

File	Description
`Shares.htm`	The sample demonstrates the usage of Server-Sent Events (SSE) and AJAX with the emWeb Web server. For detailed information about the sample refer to *AJAX* on page 39.
`SSE_IP.htm`	Shows some emWeb status information.
`SSE_OS.htm`	Shows some embOS status information.
`SSE_Time.htm`	Shows the system time.

You can find these samples in your shipment under the following direction:

- `Shared\IP\IP_FS\FS_RO\html\`

4.4 AJAX

The emWeb Web server supports AJAX. AJAX is an acronym for Asynchronous JavaScript and XML. It is the foundation to build responsive and dynamic Web applications, which look and feel like desktop applications. AJAX is a special way to communicate with a Web server. In opposite to the old fashioned synchronous way where every data transmission to or from the server needs a reload of the whole Web page, AJAX works behind the scenes. With AJAX it is possible to grab the data you want and display it instantly in a Web page. No page refreshes needed, no waiting, no flickering in the browser. AJAX works on all major browsers.

AJAX combines some well-known Web techniques like HTML and CSS, JavaScript and XML. From the perspective of a developer the really interesting part AJAX are the XMLHttpRequests. An XMLHttpRequest object is an API available to Web browser scripting languages such as JavaScript and the core component for the asynchronous communication. The name XMLHttpRequest is a little bit misleading, since XMLHttpRequest can be used to send and receive any kind of data, not just XML. In most cases the exchanged data is plain text, HTML or JSON.

To exchange data without a reload of the whole page, you need create an XMLHttpRequest object. The way to create an XMLHttpRequest object is browser dependent. Therefore, it make sense to capsule the creation in a JavaScript function, which tries to handle every browser. An example is listed below:

```
//
// Create a XMLHttpRequest.
// Tries to handle the different browser implementation
//
function _CreateRequest() {
  try {
    request = new XMLHttpRequest();
  } catch (tryMS) {
    try {
      request = new ActiveXObject("Msxml2.XMLHTTP");
    } catch (otherMS) {
      try {
        request = new ActiveXObject("Microsoft.XMLHTTP");
      } catch (failed) {
        request = null;
      }
    }
  }
  return request;
}
```

With an XMLHttpRequest object it is easy to exchange data with a Web server. The XMLHttpRequest object only needs an URI which should be requested from the server and a callback function, which will be called as soon as data will be received.

```
//
// Request the details from the server.
//
function _GetData(itemName) {
  request = _CreateRequest();   // Create an  XMLHttpRequest
  if (request == null) {
    alert("Unable to create request");
    return;
  }
  var url= "../GetData.cgi";
  request.open("GET", url, true);
  request.onreadystatechange = _SampleFunction;
  request.send(null);
}
```


In the example `GetData.cgi` is a virtual file, which will be requested from the emWeb Web server and `_SampleFunction` is registered as callback function. For further information about the implementation of a virtual file, refer to *Virtual files* on page 33.

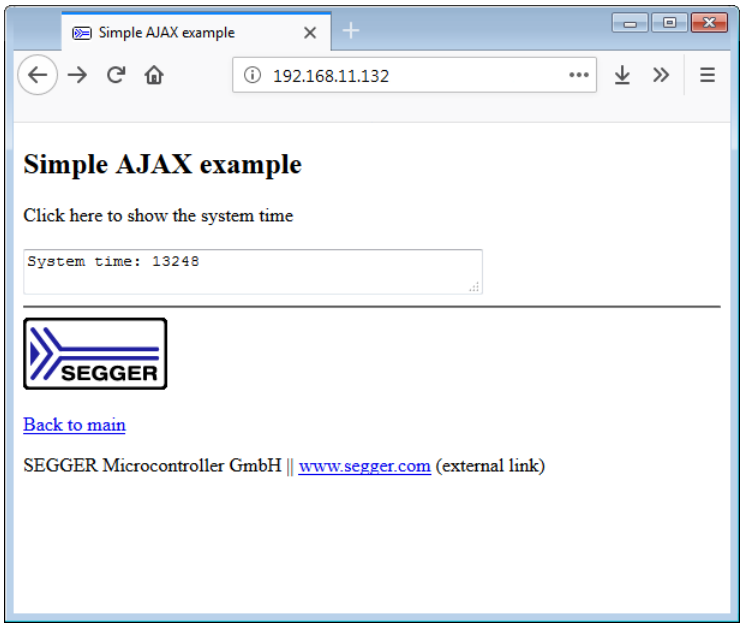
After setting the callback function, the request has to be sent. `_SampleFunction` will be called each time the readyState of the request object changes. To guarantee that the request is complete and the transmission status is ok, the readyState and HTTP status should be checked in your callback function.

```
//
// Request the details from the server.
//
function _SampleFunction() {
  if (request.readyState == 4) { // Is the request complete ?
    if (request.status == 200) { // Status OK ?
      // Do something with the received data.
    }
  }
}
```

4.4.1 Simplified AJAX example

The following example will explain how AJAX can be used to send data from your target to your browser in a very simple way.

This sample sends the current target system time to the browser when the `onclick` event has been triggered. This is done by clicking the div area above the text field.



The HTML code contains a -element with a certain ID and a title. The title is important, as it is the parameter that will be sent via the GET request. The only purpose of the ID is to make it easier to address the output text field in the JavaScript code.

```
<div id="click" title="Time">Click here to show the system time</div>
```


It then sets up the URL by adding the GET parameter. This parameter is the **Note:** `SimpleAJAX.cgi` is the virtual file used for this sample. Always make sure to use the same file name as specified in your C code.

Sending the AJAX request

The following function is most important for this sample application as it sends the AJAX request.

```
//
// Request the details from the server.
//
function _GetDetails(itemName) {
  request = _CreateRequest(); ❶
  if (request == null) {
    alert("Unable to create request");
    return;
  }
  var url= "../SimpleAJAX.cgi?text=" + escape(itemName); ❷
  request.open("GET", url, true);
  request.onreadystatechange = _DisplayDetails; ❸
  request.send(null); ❹
}
```

❶ Creating the request

The request will be created via the `_CreateRequest()` function. Since this function is alike as in the AJAX example above, please refer to *AJAX* on page 39 for deeper explanation.

❷ Setting up the request URL

Now the request URL will be set up. The URL contains the name of the virtual file used for AJAX, in this example it is `SimpleAJAX.cgi`. Then there is the request parameter, in this example it is the title property of the HTML object, as mentioned before.

❸ Opening the request and setting the event property

We open a new GET request with the URL we just created. Now, the callback function has to be set for the `onreadystatechange` event. This means when the ready state has changed, the function `_DisplayDetails` will be called.

❹ Sending the request

The last step is sending the request.

Handling the response text

When the `readyState` changes, the callback function `_DisplayDetails` will be executed. If the status is correct, the `responseText` property will be written into the text field element.

```
//
// Checks if the request was successful and updates the text field.
//
function _DisplayDetails() {
  if (request.readyState == 4) { // Is the request complete ?
    if (request.status == 200) { // Status OK ?
      textBox = document.getElementsByTagName("textarea")[0];
      textBox.innerHTML = request.responseText;
    }
  }
}
```


Setting up the onclick event

The last step regarding JavaScript code is to up the `onclick` events. This is for initializing the HTML elements by setting up the `onClick` event. When the element is clicked their corresponding function will be called.

```
//
// Initializes the elements required for the AJAX sample.
//
function _InitPage() {
  div = document.getElementById("click"); ❶

  div.onclick = function() { ❷
    _GetDetails(this.title);
  }
  _GetDetails();
}
window.onload = _InitPage; ❸
```

❶ Fetching the HTML object

First we fetch the correct HTML object that should be clickable by its ID.

❷ Setting up the onclick event

We set up the `onclick` event property with the `_GetDetails()` function and the HTML elements' title as the parameter. Therefore, when the div is clicked, the function will be executed.

❸ Setting up the onload event

Now we set the `onload` event for the website with the `_InitPage()` function. This means once the page is loaded, the function will be executed and therefore the `onclick` event for the div element will be set.

C callback routine

Finally, there is a callback routine in the dynamic content C file. This routine sends the current system time to the browser if the GET parameter matches.

```
/*********************************************************************
*
*       _callback_CGI_AJAX_Time
*
*  Function description:
*    Sends the current system time to the browser.
*/
static void _callback_CGI_AJAX_Time(WEBS_OUTPUT* pOutput,
                                    const char* sParameters) {
  char acPara[10];

  IP_WEBS_GetParaValue(sParameters, 0, NULL, 0, acPara, sizeof(acPara)); ❶
  WEBS_USE_PARA(pOutput);

  if(strcmp(acPara, "Time") == 0) { ❷
    IP_WEBS_SendString(pOutput, "System time: ");
    IP_WEBS_SendUnsigned(pOutput, OS_GetTime32(), 10, 0); ❸
  }
}
```

❶ Fetch request parameter

First, we have to store the request parameter in a char array.

❷ Handle request parameter

Next, we process the request parameter by checking if it equals “Time”. This parameter is the `title` attribute of the clicked element.

❸ Send response text

The last step is to send the correct response text that will then be displayed in the output div.

Virtual files

Using SSE requires a virtual file, which has to be added with its corresponding callback routine to the virtual file array in the dynamic content file.

```
/*********************************************************************
*
*       _aVFiles
*
*  Function description
*    Defines all virtual files used by the web server.
*/
static const WEBS_VFILES _aVFiles[]  = {
  {"SimpleAJAX.cgi", _callback_CGI_AJAX_Time },
  { NULL,            NULL                    }
};
```


4.5 Combined use of SSE and AJAX

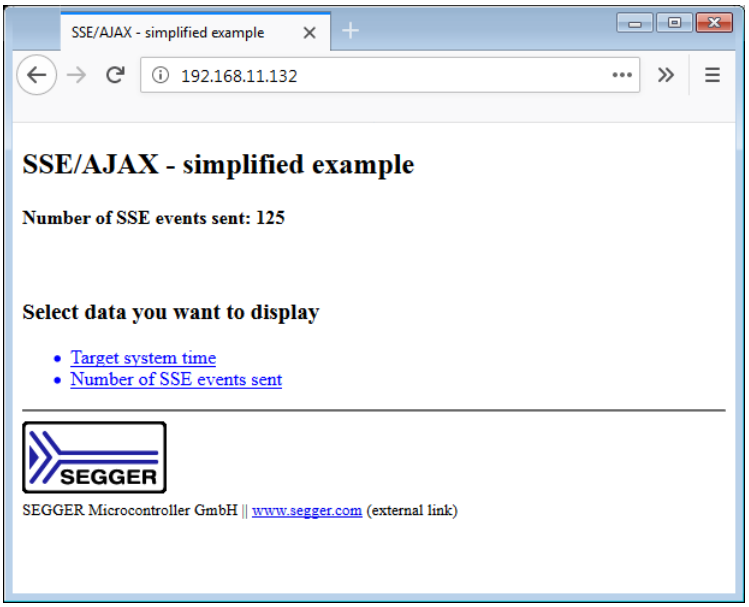
You can also combine the use of Server-Sent Events (SSE) and AJAX. This will be demonstrated with some examples, a simplified one and two advanced examples.

4.5.1 Simplified AJAX/SSE sample

The following example will explain how AJAX can be used in conjunction with SSE in a simple way.

This sample is similar to the `SSE_Simple` example, since it also sends the target system time and the count of SSE events sent to the browser via Server-Sent Events.

The main difference is, and that is where AJAX comes into place, that you can choose which information should be displayed. This is done by clicking a certain HTML element and is handled via the `onclick` event.



As shown above, you simply click on a certain list element to show the desired data.

The HTML code contains a div element used for displaying the data and two list elements for selecting which data should be shown.

```
<div id="output"></div>
<br>
<h3>Select data you want to display</h3>
<ul>
  <li id="Time">Target system time</li>
  <li id="NumEvents">Number of SSE events sent</li>
</ul>
```


AJAX and SSE callback routines

First, we will take a look at how the callback routines necessary for SSE and AJAX work. You will find these routines in the dynamic content C file.

The following callback handles the AJAX part of this sample and processes the request parameter.

```
/*********************************************************************
*
*       _callback_CGI_AJAX_Data
*
*  Function description:
*    Processes the AJAX data.
*/
static void _callback_CGI_AJAX_Data(WEBS_OUTPUT* pOutput,
                                    const char* sParameters) {
  char acPara[20];

  IP_WEBS_GetParaValue(sParameters, 0, NULL, 0, acPara, sizeof(acPara)); ❶
  WEBS_USE_PARA(pOutput);

  if(strcmp(acPara, "Time") == 0) { ❷
    _DisplaySSECount = 0; ❸
    IP_WEBS_SendFormattedString(pOutput,
                                "<h4>System time: %d</h4>",
                                OS_GetTime32()); ❹
  } else if(strcmp(acPara, "NumEvents") == 0) {
    _DisplaySSECount = 1;
    IP_WEBS_SendFormattedString(pOutput,
                                "<h4>Number of SSE events sent: %d</h4>",
                                _NumSSESent);
  }
}
```

❶ Fetch request parameter

First, we have to store the request parameter in a char array.

❷ Handle request parameter

Next, we can process the request parameter by checking if it either equals “`Time`” or “`NumEvents`”. This parameter is the `id` attribute of the clicked element.

❸ Set flag

We also need to set a flag that indicates which data is displayed at the moment. With the help of this flag, we only send information via SSE that is actually required. The use of this flag will be explained a bit deeper later on.

❹ Send response text

The last step is to send the correct response text that will then be displayed in the output div.

4.5.1.1 C code

Now we will take a look at the callback routine required for SSE.

```
/*********************************************************************
*
*       _SendTime
*
*  Function description:
*    Sends the system time and the total number of SSE events
*    to the Web browser.
*
*  Return value:
*    0: Data successfully sent.
*   -1: Data not sent.
*    1: Data successfully sent. Connection should be closed.
*/
static int _SendSSE(WEBS_OUTPUT* pOutput) {
  int  r;
  U32  Time;

#ifdef _WIN32
  Time = GetTickCount();
#else
  Time = OS_GetTime32();
#endif
  //
  // Send implementation specific header to client
  //
  IP_WEBS_SendString(pOutput, "data: ");
  if(_DisplaySSECount == 0) { ❷
    IP_WEBS_SendUnsigned(pOutput, Time, 10, 0); // Send system time
  } else {

  IP_WEBS_SendUnsigned(pOutput, _NumSSESent, 10, 0); // Send number of SSE events
  }
  IP_WEBS_SendString(pOutput, "\r\n");
  IP_WEBS_SendString(pOutput, "\n\n"); // End of the SSE data
  r = IP_WEBS_Flush(pOutput);
  _NumSSESent++; ❸
  return r;
}

/*********************************************************************
*
*       _callback_CGI_SSESimple
*
*  Function description:
*    Sends the SSE data.
*/
static void _callback_CGI_SSE_AJAX_Simple(WEBS_OUTPUT* pOutput,
                                          const char* sParameters) {
  int r;

  WEBS_USE_PARA(sParameters);
  //
  // Construct the SSE Header
  //
  IP_WEBS_SendHeaderEx(pOutput, NULL, "text/event-stream", 1);
  IP_WEBS_SendString(pOutput, "retry: 1000\n");
  // Normally, the browser attempts to reconnect to the server
 ~3 seconds after each connection is closed. We change that timeout 1 sec.
  for (;;) {
    r = _SendSSE(pOutput); ❶
    if (r == 0) {
  // Data transmitted, Web browser is still waiting for new data.
#ifdef _WIN32
```



```
      Sleep(500);
#else
      OS_Delay(500);
#endif
    } else { // Even if the data transmission was successful,
             // it could be necessary to close the connection after transmission.
      break; // This is normally the case if the Web server otherwise
             // could not process new connections.
    }
  }
}
```

❶ Send the SSE header and SSE data

After constructing the SSE header, we send the actual SSE data via the `_SendSSE()` helper function.

❷ Check flag

By checking the flag set in the AJAX callback routine (see above), we only need to send the data that is actually required. This is either the target system time or the total count of all SSE events sent.

❸ Increase counter

After the SSE event has been sent, the counter will be increased.

Virtual files

We also need to add both virtual files with their callbacks to the virtual file array.

```
/*********************************************************************
*
*       _aVFiles
*
*  Function description
*    Defines all virtual files used by the web server.
*/
static const WEBS_VFILES _aVFiles[]  = {
  {"SSE_Data_Simple.cgi", _callback_CGI_SSE_AJAX_Simple },
  {"AJAX_Data_Simple.cgi", _callback_CGI_AJAX_Data      },
  { NULL, NULL }
};
```


4.5.1.2 JavaScript code

The following section will explain how the JavaScript code used in this sample works. The function `_CreateRequest()` is identical to the function in the previous AJAX chapters, it will therefore be skipped. Please refer to *AJAX* on page 39 for an explanation.

Setting up the onclick event

The first step is to set the function that will be called when the `onclick` event is triggered for each list element.

```
//
// Initializes the elements required for AJAX.
//
function _InitPage() {
  liElem = document.getElementsByTagName("li");

  for(var i = 0; i < liElem.length; i++) { ❶
    li = liElem[i];

    li.onclick = function() {
      _GetDetails(this.id); ❷
    }
    _GetDetails();
  }
}
window.onload = _InitPage; ❸
```

❶ Iterating through all list elements

We fetch all <li> elements present in the HTML code and iterate through them. This makes it a lot easier, rather than adding the function manually to each element.

❷ Setting up the onclick event

For each <li> element we set up the `_GetDetails()` function. The parameter is the `id` property of the current list element.

❸ Setting up the onload event

Now we set the `onload` event for the website with the `_InitPage()` function. This means once the page is loaded, the function will be executed and therefore the `onclick` event for each list element will be set.

Sending the AJAX request

```
//
// Request the details from the server.
//
function _GetDetails(itemName) {
  request = _CreateRequest(); ❶
  if (request == null) {
    alert("Unable to create request");
    return;
  }
  var url= "../AJAX_Data_Simple.cgi?text=" + escape(itemName); ❷
  request.open("GET", url, true);
  request.onreadystatechange = _DisplayDetails; ❸
  request.send(null); ❹
}
```


❶ Creating the request

The request will be created via the `_CreateRequest()` function. Since this function is alike as in the AJAX example above, please refer to *AJAX* on page 39 for deeper explanation.

❷ Setting up the request URL

The request URL contains the name of the virtual file used for AJAX, in this example it is `AJAX_Data_Simple.cgi`. Then there is the request parameter, in this example being the `id` of the list element.

❸ Opening the request and setting the event property

We open a new GET request with the URL we just created. Now, the callback function has to be set for the `onreadystatechange` event. This means when the ready state has changed, the function `_DisplayDetails` will be called.

❹ Sending the request

The last step is sending the request.

Handling the response text

This callback function is called when the `onreadystatechange` event has been triggered. The div element with the id "`output`" is retrieved from the document. If the request status is correct, the contents of the div element will be changed to the response text from the request.

```
//
// Checks if the request was successful and updates the text field.
//
function _DisplayDetails() {
  if (request.readyState == 4) { // Is the request complete ?
    if (request.status == 200) { // Status OK ?
      textBox = document.getElementById("output");
      textBox.innerHTML = request.responseText;
    }
  }
}
```

Processing the SSE data

The last step is to process the data sent to the browser via SSE.

```
if(typeof(EventSource) !== "undefined") {
  var source = new EventSource("SSE_Data_Simple.cgi"); ❶
  source.onmessage = function(event) {
    out = document.getElementById("output");

    if(out.innerHTML.includes("Number of SSE events sent")) { ❷
      out.innerHTML = "<h4>Number of SSE events sent: " + event.data + "</h4>";
    } else {
      out.innerHTML = "<h4>System time: " + event.data + "</h4>";
    }
  };
} else {
  document.getElementById("output").innerHTML = "Sorry, your browser does not
 support Server-Sent Events (SSE)...";
}
```

❶ Retrieving the event data

We fetch the SSE data by using the `EventSource` interface with the virtual file we set up for SSE. In this case it is `SSE_Data_Simple.cgi`.

Note: Be careful not to mix up both virtual files. Make sure you use the virtual file linked with the SSE callback to process SSE data and not the virtual file used for AJAX.

❷ HTML manipulation

The output div now contains the response text and depending on which information is displayed, we can determine which SSE data should be written into the output div.

The SSE data can be accessed with `event.data`. For instance, you could split the data string if needed, an example would be a comma separated value list. In this example this is not necessary, since only one value is sent at a time.

4.5.2 Additional AJAX and SSE samples

The emWeb Web server comes with some advanced sample Web pages to demonstrate how AJAX can be used to get and visualize data from your target.

File	Description
`Products.htm`	Simple AJAX sample. The Web page shows pictures of SEGGER middleware products. On click on one of the pictures an XMLHTTPRequest is created and additional product related information is requested from the emWeb Web server.
`Shares.htm`	The sample demonstrates the usage of Server-Sent Events (SSE) and AJAX with the emWeb Web server. The displayed stock quotes table is random data generated by the server and updated every second via SSE. The graph is updated via AJAX. Every update of the stock prices table triggers the graph library to request the last 30 stock prices of the selected company to redraw the graph. All stock quotes are fictional. The goal of this example is to demonstrate how simple it is to visualize any kind of data with the emWeb Web server. The sample uses the open source JavaScript library RGraph for the visualization of the stock quotes. RGraph is an HTML5 charts library, which uses the MIT license. The latest version of the library can be downloaded from http://www.rgraph.net/.

All samples are hardware independent and tested with popular Web browsers like Internet Explorer, Firefox and Chrome.

For further information about AJAX, XMLHttpRequest handling, and data visualization we recommend one of the many available reference books.

Chapter 5

Form handling

The emWeb Web server supports both `POST` and `GET` actions to receive form data from a client. POST submits data to be processed to the identified resource. The data is included in the body of the request. GET is normally only used to requests a resource, but it is also possible to use GET for actions in Web applications. Data processing on server side might create a new resource or update existing resources or both.

Every HTML form consists of input items like textfields, buttons, checkboxes, etc. Each of these input items has a `name` tag. When the user places data in these items in the form, that information is encoded into the form data. Form data is a stream of `<name>=<value>` pairs separated by the `&` character. The value each of the input item is given by the user is called the value. The `<name>=<value>` pairs are URL encoded, which means that spaces are changed into "+" and special characters are encoded into hexadecimal values. Refer to *[RFC 1738]* for detailed information about URL encoding. The parsing and decoding of form data is handled by the emWeb Web server. Thereafter, the server calls a callback function with the decoded and parsed strings as parameters. The responsibility to implement the callback function is on the user side.

Valid characters for CGI function names:

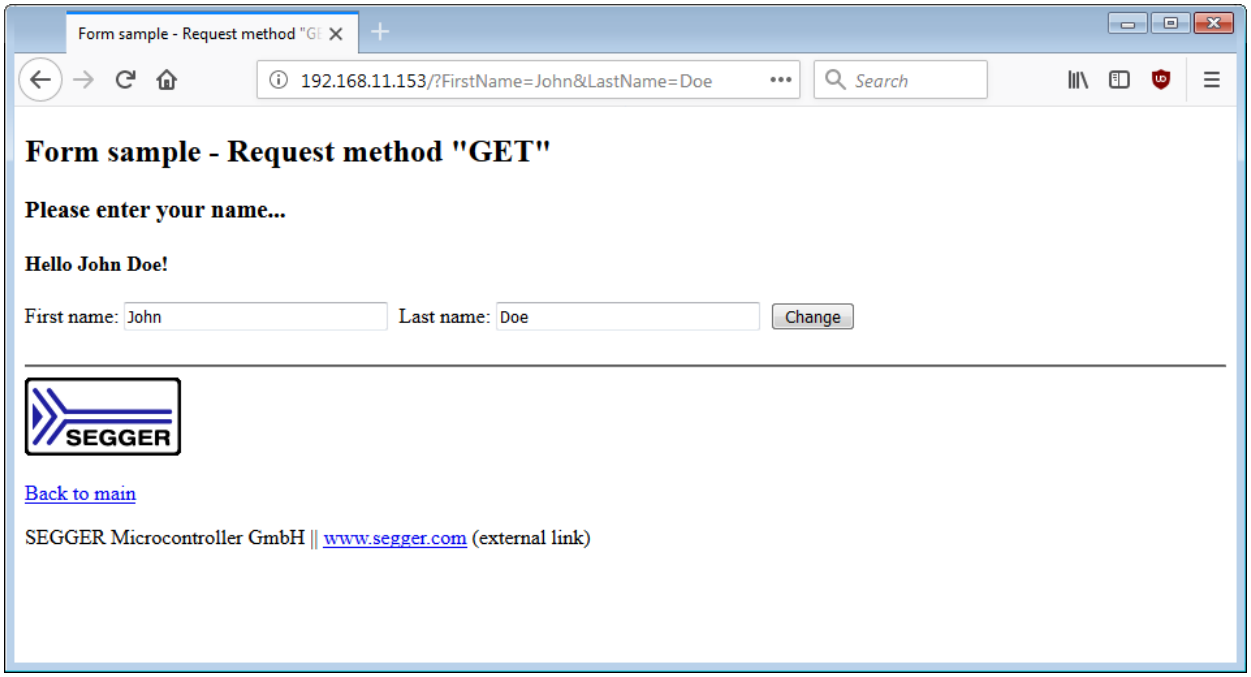
- A-Z
- a-z
- 0-9
- . _ -

Valid characters for CGI parameter values:

- A-Z
- a-z
- 0-9
- All URL encoded characters
- . _ - * ( ) ! $

5.1 Simple form processing sample

The following example shows the handling of the output of HTML forms with your Web server application. The example Web page `FormGET.htm` implements a form with three inputs, two text fields and one button.



An excerpt of the HTML code of the Web page as it is added to the server is listed below:

```
<hr>
  <H1>Please enter your name...</H1>
  <H2>Hello <!--#exec cgi="FirstName"--> <!--#exec cgi="LastName"-->!</H2>
  <form action="" method="GET">
    <label for="FirstName">First name: </label>
    <input name="FirstName" type="text" size="30" maxlength="30"
           value="<!--#exec cgi="FirstName"-->"> 
    <label for="LastName">Last name: </label>
    <input name="LastName"  type="text" size="30" maxlength="30"
           value="<!--#exec cgi="LastName"-->"> 
    <input type="submit" value="Change">
  </form>
<hr>
```

The action field of the form can specify a resource that the browser should reference when it sends back filled-in form data. If the action field defines no resource, the current resource will be requested again.

If you request the Web page from the emWeb Web server and check the source of the page in your Web browser, the CGI parts “<!—-#exec cgi=”FirstName“—->” and “<!—-#exec cgi=”LastName“—->” will be executed before the page will be transmitted to the server, so that in the example the values of the value= fields will be empty strings.

The HTML code of the Web page as seen by the Web browser is listed below:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Virtual file sample</title>
    <link href="Styles.css" rel="stylesheet">
  </head>
```



```
  <body>
    <header>Virtual file sample</header>
    <div class="content">
      <form action="Send.cgi" method="GET">
        <label for="FirstName">First name: </label>
        <input name="FirstName" type="text" size="30" maxlength="30"
 value=""> 
        <label for="LastName">Last name: </label>
        <input name="LastName"  type="text" size="30" maxlength="30"
 value=""> 
        <input type="submit" value="Change">
      </form>
    </div>
    <img src="Logo.gif" alt="Segger company logo" class="logo">
    <footer>
      <p>
        <a href="index.htm">
        Back to main</a></p><p>SEGGER Microcontroller GmbH &amp; Co. KG ||
        <a href="http://www.segger.com">www.segger.com</a>
        <span class="hint">(external link)</span>
      </p>
    </footer>
  </body>
</html>
```

To start form processing, you have to fill in the `FirstName` and the `LastName` field and click the `Send` button. In the example, the browser sends a `GET` request for the resource referenced in the form and appends the form data to the resource name as an URL encoded string. The form data is separated from the resource name by `?`. Every `<name>=<value>` pair is separated by `&`.

For example, if you type in the `FirstName` field `John` and `Doe` in the LastName field and confirm the input by clicking the `Send` button, the following string will be transmitted to the server and shown in the address bar of the browser.

```
http://192.168.11.37/FormGET.htm?FirstName=John&LastName=Doe
```

Note: If you use `POST` as HTTP method, the `<name>=<value>` pairs are not shown in the address bar of the browser. The `<name>=<value>` pairs are in this case included in the entity body.

The emWeb Web server parses the form data. The `<name>` field specifies the name of a CGI function which should be called to process the `<value>` field. The server checks therefore if an entry is available in the `WEBS_CGI` array.

```
static const WEBS_CGI _aCGI[] = {
  {"FirstName",   _callback_ExecFirstName},
  {"LastName",    _callback_ExecLastName },
  {NULL}
};
```

If an entry can be found, the specified callback function will be called.

The callback function for the parameter FirstName is defined as follow:

```
/*********************************************************************
*
*       Static data
*
**********************************************************************
*/
static char _acFirstName[12];

/*********************************************************************
*
*       _callback_FirstName
```



```
*/
static void _callback_ExecFirstName(        WEBS_OUTPUT * pOutput,
                                    const char        * sParameters,
                                    const char        * sValue ) {
  if (sValue == NULL) {
    IP_WEBS_SendString(pOutput, _acFirstName);
  } else {
    _CopyString(_acFirstName, sValue, sizeof(_acFirstName));
  }
}
```

The function returns a string if `sValue` is `NULL`. If `sValue` is defined, it will be written into a character array. Because HTTP transmission methods `GET` and `POST` only transmit the value of filled input fields, the same function can be used to output a stored value of an input field or to set a new value. The example Web page shows after entering and transmitting the input the string Hello John Doe above the input fields until you enter and transmit another name to the server.

Chapter 6

Authentication

6.1 Basic Authentication

"HTTP/1.0", includes the specification for a Basic Access Authentication scheme. The basic authentication scheme is a non-secure method of filtering unauthorized access to resources on an HTTP server, because the user name and password are passed over the network as clear text. It is based on the assumption that the connection between the client and the server can be regarded as a trusted carrier. As this is not generally true on an open network, the basic authentication scheme should be used accordingly.

The basic access authentication scheme is described in:

RFC#	Description
[RFC 2617]	HTTP Authentication: Basic and Digest Access Authentication Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc2617.txt*

The "basic" authentication scheme is based on the model that the client must authenticate itself with a user-ID and a password for each realm. The realm value should be considered an opaque string which can only be compared for equality with other realms on that server. The server will service the request only if it can validate the user-ID and password for the protection space of the Request-URI. There are no optional authentication parameters.

Upon receipt of an unauthorized request for a URI within the protection space, the server should respond with a challenge like the following:

```
WWW-Authenticate: Basic realm="Embedded Web server"
```

where "`Embedded Web server`" is the string assigned by the server to identify the protection space of the Request-URI. To receive authorization, the client sends the user-ID and password, separated by a single colon (":") character, within a base64 encoded string in the credentials.

If the user agent wishes to send the user-ID "`user`" and password "`pass`", it would use the following header field:

```
Authorization: Basic dXNlcjpwYXNz
```


6.1.1 Basic Authentication example

The client requests a resource for which authentication is required:

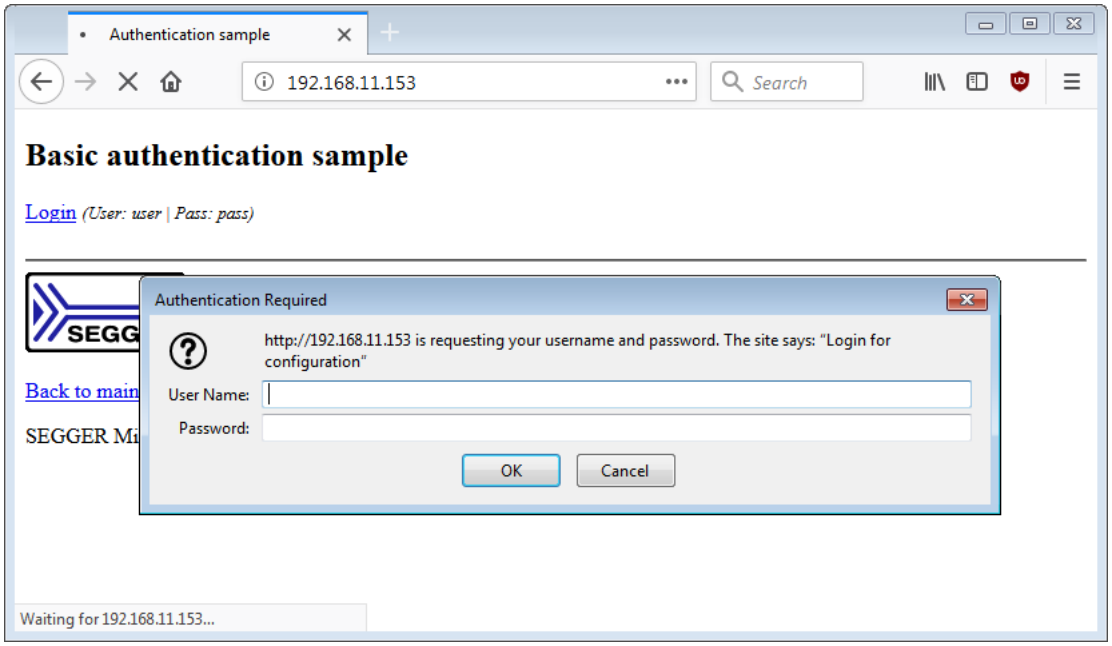
```
GET /conf/Authen.htm HTTP/1.1
Host: 192.168.11.37
```

The server answers the request with a “401 Unauthorized” status page. The header of the 401 error page includes an additional line WWW-Authenticate. It includes the realm for which the proper user name and password should be transmitted from the client (for example, a Web browser).

```
HTTP/1.1 401 Unauthorized
Date: Mon, 04 Feb 2008 17:00:44 GMT
Server: emWeb
Accept-Ranges: bytes
Content-Length: 695
Connection: close
Content-Type: text/html
X-Pad: avoid browser bug
WWW-Authenticate: Basic realm="Embedded Web server"

<HTML>
<HEAD><TITLE>401 Unauthorized</TITLE></HEAD>
<BODY>
<H1>401 Unauthorized</H1>
Browser not authentication-capable or authentication failed.<P>
</BODY>
</HTML>
```

The client interprets the header and opens a dialog box to enter the user name and password combination for the realm of the resource.



Note: The emWeb Web server example always uses the following user name and the password combination:

- User Name: `user`
- Password: `pass`

Enter the proper user name/password combination for the requested realm and confirm with the OK button. The client encodes the user name/password combination to a base64

encoded string and requests the resource again. The request header is enhanced by the following line: `Authorization: Basic dXNlcjpwYXNz`

```
GET /conf/Authen.htm HTTP/1.1
Host: 192.168.11.37
Authorization: Basic dXNlcjpwYXNz
```

The server decodes the user name/password combination and checks if the decoded string matches to the defined user name/password combination of the realm. If the strings are identical, the server delivers the page. If the strings are not identical, the server answers again with a "401 Unauthorized" status page.

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: emWeb
Expires: THU, 26 OCT 1995 00:00:00 GMT
Transfer-Encoding: chunked

200
<HTML>
  <HEAD>
    <TITLE>Web server configuration</TITLE>
  </HEAD>
  <BODY>
    <!-- Content of the page -->
  </BODY>
</HTML>
0
```

6.1.2 Configuration of the Basic Authentication

The emWeb Web server checks the access rights of every resource before returning it. The user can define different realms to separate different parts of the Web server resources. An array of `WEBS_ACCESS_CONTROL` structures has to be implemented in the user application. Refer to *Structure `WEBS_ACCESS_CONTROL`* on page 157 for detailed information about the elements of the `WEBS_ACCESS_CONTROL` structure. If no authentication should be used, the array includes only one entry for the root path.

```
WEBS_ACCESS_CONTROL _aAccessControl_Basic[] = {
  { "/", NULL, NULL, NULL },
  NULL , NULL, NULL, NULL
};
```

To define a realm "`conf`", an additional `WEBS_ACCESS CONTROL` entry has to be implemented.

```
WEBS_ACCESS_CONTROL _aAccessControl_Basic[] = {
  { "/conf/", "Login for configuration", "user:pass", NULL },
  { "/", NULL, NULL, NULL },
  NULL , NULL, NULL, NULL
};
```

The string "`Login for configuration`" defines the realm. "`user:pass`" is the user name/password combination stored in one string.

6.2 Digest Authentication

The Digest Authentication uses the same mechanisms used by the *Basic Authentication* on page 58 but aims for an improved security level. This is reached by not transferring the password in plain text. In addition it allows to store the password in the web server in hashed form instead of plain text as well.

Digest authentication is based on MD5 hashing to prevent the password being sent plain text. Several variables are known to both sides, transferred in plain text. These known variables are hashed on their own and then being hashed with the password not transferred in plain text on each side. Only if both resulting hashes are equal the server allows access to a protected path.

While MD5 can be brute forced, like many other password based schemes its security is based on the length of the password used. A longer and good randomized password requires more time to be brute forced.

Current MD5 brute force tools running on GPUs are said to be able to brute force a password of 7 characters within 6 minutes. So by adding more characters this will result in much longer run times, multiplied by the number of valid letters in the password and multiplied by the number of characters added.

While Digest Authentication is not the most secure way to prevent unauthorized access to the Web server it can be used to harden the security by combining it with SSL. SSL itself will secure the connection while Digest Authentication prevents plain text passwords to be stored on the target.

6.2.1 Configuration of the Digest Authentication

The Digest Authentication is setup similar to the *Configuration of the Basic Authentication* on page 60 except that only the username and a hash MD5(Username:Realm:Password) is stored. Refer to *Structure WEBS_ACCESS_CONTROL* on page 157 for detailed information about the elements of the `WEBS_ACCESS_CONTROL` structure.

The following example shows the realm “`conf`” being added for the following parameters:

- Username: “user”
- Password: “pass”
- Realm: “Login for configuration”

```
WEBS_ACCESS_CONTROL aAccessControl_Digest[] = {
  {"/conf/","Login for configuration","user","59294bd4737af087eea5da392cc23c14"},
  { "/"    , NULL                    , NULL , NULL },
  { NULL   , NULL                    , NULL , NULL }
};
```

The Digest Authentication HA1 string can be calculated by using the following example:

```
char acHA1[33];
const char* sUserRealmPass = "User:Login for configuration:Pass";
IP_WEBS_AUTH_DIGEST_CalcHA1(sUserRealmPass, strlen(sUserRealmPass), &acHA1[0], sizeof(acHA1));
acHA1[32] = 0;
```

> **Note**
>
> Unlike Basic Authentication the Realm can not be changed without recalculating all hashes in the table which would require the password to be known.

Digest Authentication can be enabled for the Web server sample applications such as `IP_WebserverSample.c` by enabling the `WEBS_USE_AUTH_DIGEST` define in the `WEBS_Conf.h` file.

Chapter 7

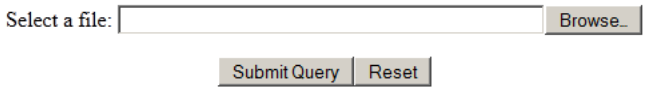
File upload

The emWeb Web server supports file uploads from the client. For this to be possible a real file system has to be used and the define `WEBS_SUPPORT_UPLOAD` has to be defined to "1". To enable file upload in your application `IP_WEBS_AddUpload()` has to be called.

From the application side uploading a file in general is the same as for other form data as described in *Form handling* on page 52. For file uploading a `<form>` field with encoding of type `multipart/form-data` is needed. An upload form field may contain additional input fields that will be parsed just as if using a non upload formular and can be parsed in your callback using `IP_WEBS_GetParaValue()` or by using `IP_WEBS_GetParaValuePtr()`.

7.1 Simple form upload sample

The following example shows the handling of file uploads with your Web server application. The example Web page `Upload.htm` implements a form with a file upload field.

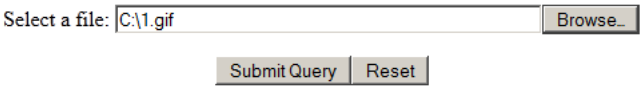


The HTML code of the Web page as it is added to the server is listed below:

```
<HTML>
  <BODY>
    <CENTER>
      <P>
        <form action="Upload.cgi" method="post" enctype="multipart/form-data">
          <p>Select a file: <input name="Data" type="file">
          </p>
          <input type="submit"><input type="reset">
        </form>
      </P>
    </CENTER>
  </BODY>
</HTML>
```

The action field of the form can specify a resource that the browser should reference when it has finished handling the file upload. If the action field defines no resource, the current resource will be requested again.

To upload a file, you have to select a file by using the browse button and select a file to upload and click the Send button. In the example, the browser sends a POST request for the resource referenced in the form and appends the form and file data in an encoded string.



The emWeb Web server parses additional form data passed besides the file to be uploaded. This works the same as handling form data described in *Form handling* on page 52. The action parameter of the `<form>` field specifies the name of a virtual file that should be processed. A callback can then be used to provide an answer page referring the state of the upload. The example below shows how to check the success of an upload using a virtual file provided by the `WEBS_VFILES` array:

```
static const WEBS_VFILES _aVFiles[]  = {
 {"Upload.cgi", _callback_CGI_UploadFile  },
  { NULL, NULL }
};
```

If an entry can be found, the specified callback function will be called.

The callback function for the file `Upload.cgi` is defined as follow:

```
/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

/*********************************************************************
*
*       _callback_CGI_UploadFile
*/
static  void  _callback_CGI_UploadFile(      WEBS_OUTPUT* pOutput,
```



```
                                        const char*        sParameters) {
        int    r;
  const char * pFileName;
        int    FileNameLen;
  const char * pState;       // Can be 0: Upload failed;
                             // 1: Upload succeeded; Therefore we do not need to
                             //    know the length, it will always be 1.

  IP_WEBS_SendString(pOutput, "<HTML><BODY>");
  r
  = IP_WEBS_GetParaValuePtr(sParameters, 0, NULL, 0, &pFileName, &FileNameLen);
  r |= IP_WEBS_GetParaValuePtr(sParameters, 1, NULL, 0, &pState   , NULL);
  if (r == 0) {
    IP_WEBS_SendString(pOutput, "Upload of \"");
    IP_WEBS_SendMem(pOutput, pFileName, FileNameLen);
    if (*pState == '1') {
      IP_WEBS_SendString(pOutput, "\" successful!<br>");
      IP_WEBS_SendString(pOutput, "<a href=\"");
      IP_WEBS_SendMem(pOutput, pFileName, FileNameLen);
      IP_WEBS_SendString(pOutput, "\">Go to ");
      IP_WEBS_SendMem(pOutput, pFileName, FileNameLen);
      IP_WEBS_SendString(pOutput, "</a><br>");
    } else {
      IP_WEBS_SendString(pOutput, "\" not successful!<br>");
    }
  } else {
    IP_WEBS_SendString(pOutput, "Upload not successful!");
  }
 IP_WEBS_SendString(pOutput, "</BODY></HTML>");
}
```

In addition to the provided form fields from the upload form used two additional entries will be added to the end of the parameter list available for parsing:

- The original filename of the file uploaded
- The status of the upload process. This can be 0: Upload failed or 1: Upload succeeded.

The example Web page shows after the upload has been finished.

Upload of "1.gif" successful!
Go to 1.gif

The source of the Web page as seen by the Web browser is listed below:

```
<HTML><BODY>
Upload of "1.gif" successful!<br>
<a href="1.gif">Go to 1.gif</a><br>
</BODY></HTML>
```


Chapter 8

Web server configuration

The emWeb Web server can be used without changing any of the compile time flags. All compile time configuration flags are preconfigured with valid values, which match the requirements of most applications.

8.1 Web server compile time configuration

The emWeb Web server can be used without changing any of the compile time flags. All compile time configuration flags are preconfigured with valid values, which match the requirements of most applications.

The following types of configuration macros exist:

Binary switches "B"

Switches can have a value of either 0 or 1, for deactivated and activated respectively. Actually, anything other than 0 works, but 1 makes it easier to read a configuration file. These switches can enable or disable a certain functionality or behavior. Switches are the simplest form of configuration macros.

Numerical values "N"

Numerical values are used somewhere in the source code in place of a numerical constant. A typical example is the configuration of the sector size of a storage medium.

Alias "A"

A macro which operates like a simple text substitute. An example would be the define `U8`, which the preprocessor would replace with `unsigned char`.

Function replacements "F"

Macros can basically be treated like regular functions although certain limitations apply, as a macro is still put into the source code as simple text replacement. Function replacements are mainly used to add specific functionality to a module which is highly hardware-dependent. This type of macro is always declared using brackets (and optional parameters).

8.2 Web server compile time configuration switches

Type	Symbolic name	Default	Description
F	WEBS_WARN	--	Defines a function to output warnings. In debug configurations (DEBUG = 1) WEBS_WARN maps to IP_Warnf_Application().
F	WEBS_LOG	--	Defines a function to output logging messages. In debug configurations (DEBUG = 1) WEBS_LOG maps to IP_Logf_Application().
N	WEBS_IN_BUFFER_SIZE	256	Defines the size of the input buffer. The input buffer is used to store the HTTP client requests. Please refer to *Web server runtime configuration* on page 72 for further information about the usage.
N	WEBS_OUT_BUFFER_SIZE	1460	Defines the size of the output buffer. The output buffer is used to store the HTTP response. Please refer to *Web server runtime configuration* on page 72 for further information about the usage.
N	WEBS_TEMP_BUFFER_SIZE	512	Defines the size of file input buffer and for form parameters
N	WEBS_PARA_BUFFER_SIZE	256	Defines the size of the buffer used to store the parameter/value string that is given to a virtual file. If virtual files are not used in your application, remove the definition from WEBS_Conf.h to save RAM. Please refer to *Web server runtime configuration* on page 72 for further information about the usage.
N	WEBS_FILENAME_BUFFER_SIZE	64	Defines the size of the buffer used to store the requested URI/filename to access on the filesystem. Please refer to *Web server runtime configuration* on page 72 for further information about the usage.
N	WEBS_TEMP_BUFFER_SIZE	512	Defines the size of the TEMP buffer used internally by the Web server.
B	WEBS_USE_AUTH_DIGEST	0/1	Defines if standard plain text basic authentication will be used for protected paths or MD5 encrypted digest authentication.
N	WEBS_AUTH_BUFFER_SIZE	32	Defines the size of the buffer used to store the authentication string. Refer to *Basic Authentication* on page 58 for detailed information about authentication.
B	WEBS_SUPPORT_UPLOAD	0/1	Defines if file upload is enabled. Defaults to 0: Not enabled, for source code shipments and 1: Enabled, for object shipments. If you do not use

Type	Symbolic name	Default	Description
			the upload feature, define WEBS_SUPPORT_UPLOAD to 0.
N	WEBS_UPLOAD_FILENAME_BUFFER_SIZE	64	Defines the size of the output buffer.
N	WEBS_URI_BUFFER_SIZE	0	Defines the size of the buffer used to store the "full URI" of the accessed resource. By default this feature is disabled.
N	WEBS_MAX_ROOT_PATH_LEN	12	Maximum allowed root path length of the Web server in multiples of a CPU native unit (typically int). If the root path of the Web server is the root of your media you can comment out this define or set it to zero. Example: For the root path "/httpdocs" the define needs to be set to at least 9. As this is not a multiple of an int, set it to 12.

Status message Web pages

The status message Web pages are visualizations of the information transmitted to the client in the header of the Web server response. Because these visualizations are not required for the functionality of the Web server, the macros can be empty.

Type	Symbolic name	Default
A	WEBS_401_PAGE	“<HTML>\n” \ “<HEAD>\n” \ “<TITLE>401 Unauthorized</TITLE>\n” \ “</HEAD>\n” \ “<BODY>\n” \ “<H1>401 Unauthorized</H1>\n” \ “Browser not authentication-capable” \ “or authentication failed.\n” \ “</BODY>\n” \ “</HTML>\n”
A	WEBS_404_PAGE	“<HTML>\n” \ “<HEAD>\n” \ “<TITLE>404 Not Found</TITLE>\n” \ “</HEAD>\n” \ “<BODY>\n” \ “<H1>404 Not Found</H1>\n” \ “The requested document was not ” \ “found on this server.\n” \ “</BODY>\n” \ “</HTML>\n”
A	WEBS_501_PAGE	“<HTML>\n” \ “<HEAD>\n” \ “<TITLE>501 Not implemented</TITLE>\n” \ “</HEAD>\n” \ “<BODY>\n” \ “<H1>Command is not implemented</H1>\n” \ “</BODY>\n” \ “</HTML>\n”
A	WEBS_503_PAGE	“<HTML>\n” \ “<HEAD>\n” \ “<TITLE>503 Connection limit reached</TITLE>\n” \ </HEAD>\n“ \ ”<BODY>\n“ \ ”<H1>503 Connection limit reached</H1>\n“ \ ”The max. number of simultaneous connections to “ \ ”this server reached.\n“ \ ” Please try again later. \n“ \ ”</BODY>\n“ \ ”</HTML>\n“

8.3 Web server runtime configuration

The input buffer, the output buffer, the parameter buffer, the filename buffer and the maximum root path length are runtime configurable. Up to version 2.20h compile time switches `WEBS_IN_BUFFER_SIZE`, `WEBS_OUT_BUFFER_SIZE` and `WEBS_PARA_BUFFER_SIZE` were used to configure the sizes of the buffers. These compile time switches along with new switches like `WEBS_MAX_ROOT_PATH_LEN` are still available to guarantee compatibility to previous versions and are used as default values for the buffer sizes in applications where the runtime configuration function `IP_WEBS_ConfigBufSizes()` is not called. For further information, please refer to *`IP_WEBS_ConfigBufSizes`* on page 82.

Chapter 9

API functions

In this chapter, you will find a description of each emWeb function.

9.1 Overview

The table below lists the available API functions within their respective categories.

Function	Description
Main configuration functions	
IP_WEBS_AddFileTypeHook()	This function registers a FileType hook that can be used to extend or overwrite file types stored in _aFileType[].
IP_WEBS_AddPreContentOutputHook()	This function registers a hook with a callback that is called before content (including header) is generated by the web server.
IP_WEBS_AddProgressHook()	This function registers a hook with a callback that is called on during several progression steps when serving a request.
IP_WEBS_AddRequestNotifyHook()	This function registers a hook with a callback that is called on each request to pass some information like URI and METHOD used to the application.
IP_WEBS_ConfigBufSizes()	Sets the buffer size used by the Web server tasks.
IP_WEBS_ConfigFindGZipFiles()	Enables checking if a requested file is available gzip compressed.
IP_WEBS_ConfigRootPath()	This function sets the root path in the filesystem used to serve pages relative from this path.
IP_WEBS_ConfigStaticEncodedFiletypes()	Enables checking if a requested file is available encoded/compressed.
IP_WEBS_ConfigUploadRootPath()	Sets the the root path in the filesystem used to upload files relative from this path.
IP_WEBS_CountRequiredMem()	Counts the memory required for one thread.
Generic functions	
IP_WEBS_Flush()	Flushes the output buffer.
IP_WEBS_Init()	Initializes the Web server application context.
IP_WEBS_OnConnectionLimit()	Outputs an error message to the connected client.
IP_WEBS_Process()	Thread functionality of the web server.
IP_WEBS_ProcessEx()	Thread functionality of the web server.
IP_WEBS_ProcessLast()	Thread functionality of the web server.
IP_WEBS_ProcessLastEx()	Processes a HTTP request of a client and closes the connection thereafter.
IP_WEBS_Redirect()	Sends a page from the file system to the browser instead of the regular content.
IP_WEBS_Reset()	Resets internal variables for a clean start of the Web Server.
IP_WEBS_RetrieveUserContext()	Retrieves a previously stored user context for the current output session.
IP_WEBS_SendFormattedString()	Sends a string with placeholders that will be filled using SEGGER_vsnprintfEx().

Function	Description
IP_WEBS_Send204NoContent()	This function sends a “204 No Content” response for a captive portal test.
IP_WEBS_SendHeader()	Outputs a valid HTML header.
IP_WEBS_SendHeaderEx()	Outputs a valid HTML header.
IP_WEBS_SendLocationHeader()	Sends a header with a redirection code for the browser.
IP_WEBS_SendMem()	Sends data to a connected target.
IP_WEBS_SendString()	Sends a zero-terminated string to a connected target.
IP_WEBS_SendStringEnc()	Encodes and sends a zero-terminated string to a connected target.
IP_WEBS_SendUnsigned()	Sends an unsigned value to the client.
IP_WEBS_SetErrorPageCallback()	Sets a callback that gets executed when an error page is sent.
IP_WEBS_SetFileInfoCallback()	Sets a callback that can be used to pass file system information about a requested file back to the web server.
IP_WEBS_SetHeaderCacheControl()	Sets the string to be sent for the cache-control field in the header.
IP_WEBS_SetUploadFileSystemAPI()	Sets the FS API to use for file uploads.
IP_WEBS_SetUploadMaxFileSize()	Sets the maximum file size per file that can be uploaded.
IP_WEBS_StoreUserContext()	Stores a user context for the current output session.
CGI/virtual file related functions	
IP_WEBS_AddVFileHook()	Registers a function table containing callbacks to check and serve simple virtual file content that is not further processed by the Web server.
IP_WEBS_CompareFilenameExt()	Checks if the given filename has the given extension.
IP_WEBS_ConfigSendVFileHeader()	Configures behavior of automatically sending a header containing a MIME type associated to the requested files extension based on an internal list for a requested virtual file.
IP_WEBS_ConfigSendVFileHookHeader()	Configures behavior of automatically sending a header containing a MIME type associated to the requested files extension based on an internal list for a requested file being served by a registered VFile hook.
IP_WEBS_DecodeAndCopyStr()	Checks if a string includes url encoded characters, decodes the characters and copies them into destination buffer.
IP_WEBS_DecodeString()	Checks if a string includes url encoded characters, decodes the characters.
IP_WEBS_GetDecodedStrLen()	Returns the length of a HTML encoded string when decoded excluding null character.

Function	Description
IP_WEBS_GetNumParas()	Returns the number of parameters/value pairs.
IP_WEBS_GetParaValue()	Parses a string for valid paramter/value pairs and writes results in buffer.
IP_WEBS_GetParaValuePtr()	Parses a string for valid parameter/value pairs and returns a pointer to the requested parameter and the length of the parameter string without termination.
IP_WEBS_GetConnectInfo()	Retrieves the connect info that has been passed to the Web Server process function from the application.
IP_WEBS_GetURI()	Retrieves the requested URI up to the '?' char or the "full URI" including '?' character and all characters up to the next space.
IP_WEBS_UseRawEncoding()	Overrides the previously selected encoding of the Web Server to use RAW encoding.
Basic/Digest Authentication related functions	
IP_WEBS_AUTH_DIGEST_CalcHA1()	Calculates the auth digest MD5 hash for the HA1 hash value to use in the WEBS_ACCESS_CONTROL table.
IP_WEBS_AUTH_DIGEST_GetURI()	Retrieves the URI sent in the client authorization.
IP_WEBS_GetProtectedPath()	Retrieves the protected path that is tried to access by the current request.
IP_WEBS_UseAuthDigest()	Use auth digest instead of auth basic for HTTP authentication.
METHOD extension related functions	
IP_WEBS_METHOD_AddHook()	Registers a callback to serve special content upon call of a METHOD.
IP_WEBS_METHOD_CopyData()	This function can be called from a METHOD hook callback to copy the received amount of data as stated by the "Content-Length" field of the header.
WebSocket extension related functions	
IP_WEBS_WEBSOCKET_AddHook()	This function registers a WebSocket hook that can be used to connect to a WebSocket implementation.
HEADER extension related functions	
IP_WEBS_HEADER_AddFieldHook()	Registers a callback that will be notified if the registered header field is received.
IP_WEBS_HEADER_CopyData()	Copies the requested amount of data from line with the current header field.
IP_WEBS_HEADER_GetFindToken()	Searches the current header line for a token and copies its value into a buffer.
IP_WEBS_HEADER_SetCustomFields()	Sets a custom string of header fields to be included with the next header sent.
Upload related functions	
IP_WEBS_AddUpload()	Adds the upload functionality to the Web server.

Function	Description
IP_WEBS_ChangeUploadMaxFileSize()	This function allows to change the maximum file size allowed for the file that is in upload.
IP_WEBS_GetUploadFilename()	Copies the original filename of an upload into the given buffer.
IP_WEBS_SetUploadAPI()	Sets the upload API of type WEBS_UPLOAD_API to use.
Utility functions	
IP_UTIL_BASE64_Decode()	Performs BASE-64 decoding according to RFC3548.
IP_UTIL_BASE64_Encode()	Performs BASE-64 encoding according to RFC3548.
IP_UTIL_BASE64_EncodeChunk()	Performs chunked BASE-64 encoding according to RFC3548.

9.2 IP_WEBS_AddFileTypeHook()

Description

This function registers a FileType hook that can be used to extend or overwrite file types stored in _aFileType[] .

Prototype

```
void IP_WEBS_AddFileTypeHook(      WEBS_FILE_TYPE_HOOK * pHook,
                             const char                * sExt,
                             const char                * sContent);
```

Parameters

Parameter	Description
pHook	Pointer to a structure of WEBS_FILE_TYPE_HOOK.
sExt	String containing the extension.
sContent	String containing the MIME type assigned to the extension.

Additional information

The function can be used to extend or override the basic list of file extension to MIME type correlations included in the Web server. It might be necessary to extend the this list in case you want to serve a yet unknown file format. The header sent for this file in case a client requests it will be generated based on this information.

Refer to *Structure WEBS_FILE_TYPE_HOOK* on page 163 for detailed information about the structure WEBS_FILE_TYPE_HOOK.

Example

```
static WEBS_FILE_TYPE_HOOK _FileTypeHook;

int main(void){
  //
  // Register *.new files to be treated as binary that will
  // be offered to be downloaded by the browser.
  //
  IP_WEBS_AddFileTypeHook(&_FileTypeHook, "new", "application/octet-stream");
}
```


9.3 IP_WEBS_AddPreContentOutputHook()

Description

This function registers a hook with a callback that is called before content (including header) is generated by the web server.

Prototype

```
void IP_WEBS_AddPreContentOutputHook(WEBS_PRE_CONTENT_OUTPUT_HOOK * pHook,
                                     IP_WEBS_pfPreContentOutput     pf,
                                     U32                            Mask);
```

Parameters

Parameter	Description
pHook	Pointer to a resource of WEBS_PRE_CONTENT_OUTPUT_HOOK.
pf	Callback to execute before content is sent in a response.
Mask	Mask of Bitwise-OR'd values when to execute the callback: • WEBS_PRE_DYNAMIC_CONTENT_OUTPUT

Additional information

This hook can be used to intercept the web server right before sending content to its best knowledge. As all form data has been processed at this time this can be used to decide if all form data is valid like for a user:pass combination processed via a form. For example upon invalid login data it is possible to redirect from within the callback back to the login page using IP_WEBS_Redirect().

Refer to *Structure WEBS_PRE_CONTENT_OUTPUT_HOOK* on page 165 for detailed information about the structure WEBS_PRE_CONTENT_OUTPUT_HOOK.

9.4 IP_WEBS_AddProgressHook()

Description

This function registers a hook with a callback that is called on during several progression steps when serving a request.

Prototype

```
void IP_WEBS_AddProgressHook(WEBS_PROGRESS_HOOK * pHook,
                             IP_WEBS_pfProgress   pf);
```

Parameters

Parameter	Description
pHook	Pointer to a structure of WEBS_PROGRESS_HOOK
pf	Callback to execute on a request.

Additional information

A progress hook can be used for example to allocate some memory to store data between form handling callbacks on BEGIN of serving the request. The memory can then be freed again when finally the END progress is reported and the request is fully progressed.

Additional information

Refer to *Structure WEBS_PROGRESS_HOOK* on page 166 for detailed information about the structure WEBS_PROGRESS_HOOK.

9.5 IP_WEBS_AddRequestNotifyHook()

Description

This function registers a hook with a callback that is called on each request to pass some information like URI and METHOD used to the application.

Prototype

```
void IP_WEBS_AddRequestNotifyHook(WEBS_REQUEST_NOTIFY_HOOK * pHook,
                                  IP_WEBS_pfRequestNotify    pf);
```

Parameters

Parameter	Description
pHook	Pointer to a structure of WEBS_REQUEST_NOTIFY_HOOK
pf	Callback to execute on a request.

Additional information

Refer to *Structure WEBS_REQUEST_NOTIFY_HOOK* on page 168 for detailed information about the structure WEBS_REQUEST_NOTIFY_HOOK.

9.6 IP_WEBS_ConfigBufSizes()

Description

Sets the buffer size used by the Web server tasks.

Prototype

```
void IP_WEBS_ConfigBufSizes(WEBS_BUFFER_SIZES * pBufferSizes);
```

Parameters

Parameter	Description
pBufferSizes	Configuration of buffer sizes.

Additional information

The structure WEBS_BUFFER_SIZES is defined as follow:

```
typedef struct WEBS_BUFFER_SIZES {
  U32 NumBytesInBuf;
  U32 NumBytesOutBuf;
  U32 NumBytesFilenameBuf;
  U32 MaxRootPathLen;
  U32 NumBytesParaBuf;
} WEBS_BUFFER_SIZES;
```

Since version 3.00, the buffers used by the Web server are runtime configurable. Earlier versions of the Web server used compile time switches to define the buffer sizes. If IP_WEBS_ConfigBufSizes() will not be called, the values of the compile time switches will be used to configure the buffer sizes.

We recommend at least 256 bytes for the input buffer and 512 bytes for the output buffer. If virtual files are not used in your application, the parameter buffer and others can be set to 0 to save RAM.

9.7 IP_WEBS_ConfigFindGZipFiles()

Description

Enables checking if a requested file is available gzip compressed.

Prototype

```
void IP_WEBS_ConfigFindGZipFiles(const char * sExtension,
                                 int    ReplaceLastChar);
```

Parameters

Parameter	Description
sExtension	Extension including period letter to add to the end of the URI requested. The resulting URI will be tried to open and sent back gzip compressed.
ReplaceLastChar	Replace the last character of the requested URI with the first character of the extension.

Additional information

The browser needs to accept gzip compressed content and will tell the web server if this is supported.

Only files that do not contain dynamic content can be used. When enabled and allowed by the browser, a gzip compressed version of the requested file is tried to be opened. If this fails, the original filename is tried.

Examples:

- URI = "events.js"; sExtension = ".gz"; ReplaceLastChar = 0; => "events.js.gz"
- URI = "events.js"; sExtension = "z"; ReplaceLastChar = 0; => "events.jsz"
- URI = "events.js"; sExtension = "z"; ReplaceLastChar = 1; => "events.jz"

9.8 IP_WEBS_ConfigStaticEncodedFiletypes()

Description

Enables checking if a requested file is available encoded/compressed.

Prototype

```
void IP_WEBS_ConfigStaticEncodedFiletypes
                                    (const WEBS_STATIC_ENCODED_FILETYPES * paList);
```

Parameters

Parameter	Description
paList	List of WEBS_STATIC_ENCODED_FILETYPES entries. The list has to end with a NULLed entry that is used as end marker.

Additional information

Overrides IP_WEBS_ConfigFindGZipFiles() if used.

The browser provides encoding supported via the “Accept-Encoding” field. Content for a known encoding can only be delivered if it is offered in the same request.

Only files that do not contain dynamic content can be used. When enabled and allowed by the browser, an encoded/compressed version of the requested file is tried to be opened. If this fails, the original filename is tried.

For an example on how to use the members of the WEBS_STATIC_ENCODED_FILETYPES structure, please refer to the example provided for IP_WEBS_ConfigFindGZipFiles() .

Example

```
static const WEBS_STATIC_ENCODED_FILETYPES _aEncodedContent[] = {
  { "br"  , ".br", 0u },  // Brotli
  { "gzip", ".gz", 0u },  // GZIP
  { NULL  , NULL , 0u }   // End of list
};

IP_WEBS_ConfigStaticEncodedFiletypes(_aEncodedContent);
```


9.9 IP_WEBS_ConfigRootPath()

Description

This function sets the root path in the filesystem used to serve pages relative from this path.

Prototype

```
int IP_WEBS_ConfigRootPath(const char * sRootPath);
```

Parameters

Parameter	Description
sRootPath	Root path to prepend to URI.

Return value

0 O.K.
1 Error.

Additional information

By default the root path used is the root path of your filesystem. Configuring a root path can be used to separate the Web server from other services like an FTP server root folder. A classic use sample would be that all Web pages are stored in the subfolder "/httpdocs". In this case you can call `IP_WEBS_ConfigRootPath`("/httpdocs") to load all Web pages relative from this folder in your filesystem instead from the root folder of your filesystem.

Other services like an FTP server can be used to grant access to the root folder of your filesystem to allow access to the /httpdocs subfolder and other files in your filesystem as well.

The root path can be as long as the maximum root path length configured. If `IP_WEBS_ConfigBufSizes()` will not be called to set the maximum root path length, the default `WEBS_MAX_ROOT_PATH_LEN` will be used.

9.10 IP_WEBS_ConfigUploadRootPath()

Description

Sets the the root path in the filesystem used to upload files relative from this path.

Prototype

```
int IP_WEBS_ConfigUploadRootPath(const char * sUploadRootPath);
```

Parameters

Parameter	Description
sUploadRootPath	Root path to prepend to upload.

Return value

0 O.K.
1 Error.

Additional information

By default uploads are placed in the root folder of your filesystem. Configuring an upload root path can be used to redirect uploads to another path like an "/upload" folder.

The upload root path can be as long as the maximum path that can be used with the upload filename defined by `WEBS_UPLOAD_FILENAME_BUFFER_SIZE` (minus 1 byte for string termination).

9.11 IP_WEBS_CountRequiredMem()

Description

Counts the memory required for one thread. This can be used to determine the total required memory pool size for a configuration.

Prototype

```
U32 IP_WEBS_CountRequiredMem(WEBS_CONTEXT * pContext);
```

Parameters

Parameter	Description
pContext	Context keeping track of configured settings. Can be NULL.

Return value

Amount of memory required for internals for handling one thread.

Additional information

In addition to the memory requirement calculated for the Web server internals some additional memory might be required for managing a memory pool.

9.12 IP_WEBS_Flush()

Description

Flushes the output buffer.

Prototype

```
int IP_WEBS_Flush(WEBS_OUTPUT * pOutput);
```

Parameters

Parameter	Description
pOutput	Pointer to the WEBS_OUTPUT structure.

Return value

1 Data transmitted, connection will be closed.
0 Data transmitted, connection will be kept open after transmission.
-1 Error. Data not transmitted, connection will be closed.

Additional information

Normally, the stack handles all the data transmission. IP_WEBS_Flush() should only be used in special use cases like implementing Server-Sent Events, where data transmission should be done immediately.

9.13 IP_WEBS_Init()

Description

Initializes the Web server application context. Has to be called if `IP_WEBS_ProcessEx()` and `IP_WEBS_ProcessLastEx()` are used for the task handling.

Prototype

```
void IP_WEBS_Init(      WEBS_CONTEXT     * pContext,
                  const WEBS_IP_API      * pIP_API,
                  const WEBS_SYS_API     * pSYS_API,
                  const IP_FS_API        * pFS_API,
                  const WEBS_APPLICATION * pApplication);
```

Parameters

Parameter	Description
pContext	Application specific Web server context.
pIP_API	Pointer to a structure holding the IP related API functions.
pSYS_API	Pointer to a structure holding the system related API functions.
pFS_API	Pointer to a structure holding the file system related API functions.
pApplication	Web server application settings.

Additional information

The parameter pContext is a structure holding all the required function pointers to the routines used to send and receive bytes from/to the client, access the file system, allocate and free memory, etc.

```
typedef struct WEBS_CONTEXT {
  const WEBS_IP_API      *pIP_API;
  const WEBS_SYS_API     *pSYS_API;
  const IP_FS_API        *pFS_API;
  const WEBS_APPLICATION *pApplication;
        void             *pWebsPara;
        void             *pUpload;
} WEBS_CONTEXT;
```

WEBS_IP_API includes all functions, which are required for the used IP stack and is defined as follow:

```
typedef struct WEBS_IP_API {
  IP_WEBS_tSend    pfSend;
  IP_WEBS_tReceive pfReceive;
} WEBS_IP_API;

typedef int (*IP_WEBS_tSend)    (const unsigned char  *pData,
                                 int    len,
                                 void  *pConnectInfo );
typedef int (*IP_WEBS_tReceive) (const unsigned char  *pData,
                                 int    len,
                                 void  *pConnectInfo );
```

The send and receive functions should return the number of bytes successfully sent/received to/from the client. The pointer pConnectInfo is passed to the send and receive routines. It can be used to pass a pointer to a structure containing connection information or to pass a socket number.

WEBS_SYS_API includes all functions, which are required to allocate and free memory and is defined as follow:

```
typedef struct WEBS_SYS_API {
  IP_WEBS_tAlloc   pfAlloc;
  IP_WEBS_tFree    pfFree;
} WEBS_SYS_API;

typedef void *(*IP_WEBS_tAlloc) (               U32    NumBytesReq );

typedef void  (*IP_WEBS_tFree)  (               void  *p);
```

The alloc function returns a void pointer to the allocated space, or NULL if there is insufficient memory available.

For details about the parameter pFS_API and the IP_FS_API structure, refer to *File system abstraction layer* on page . For details about the parameter pApplication and the WEBS_APPLICATION structure, refer to *Structure WEBS_APPLICATION* on page 158.

The pWebsPara and pUpload should not be changed. The stack fills the structures, if necessary.

IP_WEBS_Init() has to be called if IP_WEBS_ProccessEx() and IP_WEBS_ProcessLastEx() are used for the connection handling. Refer to *IP_WEBS_ProcessEx* on page 95 and *IP_WEBS_ProcessLastEx* on page 97 for detailed information.

Example

```
/* Excerpt from IP_WebserverSample.c*/
/*********************************************************************
*
*     _WebServerChildTask
*
*/
static void _WebServerChildTask(void *pContext) {
  WEBS_CONTEXT ChildContext;
  long hSock;
  int Opt;

  hSock    = (long)pContext;
  Opt      = 1;
  setsockopt(hSock, SOL_SOCKET, SO_KEEPALIVE, &Opt, sizeof(Opt));
  //
  // Initialize the context of the child task.
  //
  IP_WEBS_Init(&ChildContext, &_Webs_IP_API, &_FS_API, &_Application);
  if (_ConnectCnt < MAX_CONNECTIONS) {
    IP_WEBS_ProcessEx(&ChildContext, pContext, NULL);
  } else {
    IP_WEBS_ProcessLastEx(&ChildContext, pContext, NULL);
  }
  _closesocket(hSock);
  OS_EnterRegion();
  _AddToConnectCnt(-1);
  OS_Terminate(0);
  OS_LeaveRegion();
}

/*********************************************************************
*
*       _WebServerParentTask
*
*/
static void _WebServerParentTask(void) {
  struct sockaddr    Addr;
  struct sockaddr_in InAddr;
  U32  Timeout;
  long hSockListen;
  long hSock;
  int  AddrLen;
  int  i;
```



```
  int  t;
  int  t0;
  WEBS_IP_API  Webs_IP_API;
  WEBS_SYS_API Webs_SYS_API;
  WEBS_BUFFER_SIZES BufferSizes;

  Timeout = IDLE_TIMEOUT;
  IP_WEBS_SetFileInfoCallback(&_pfGetFileInfo);
  //
  // Assign file system
  //
  _pFS_API = &IP_FS_ReadOnly;  // To use a a real filesystem like emFile
                               // replace this line.
//  _pFS_API = &IP_FS_FS;        // Use emFile
//  IP_WEBS_AddUpload();         // Enable upload
  //
  // Configure buffer size.
  //
  IP_MEMSET(&BufferSizes, 0, sizeof(BufferSizes));
  BufferSizes.NumBytesInBuf       = WEBS_IN_BUFFER_SIZE;
  //
  // Use max. MTU configured for the last interface added minus worst
  // case IPv4/TCP/VLAN headers. Calculation for the memory pool
  // is done under assumption of the best case headers with - 40 bytes.
  //
  BufferSizes.NumBytesOutBuf      = IP_TCP_GetMTU(_IFaceId) - 72;
  BufferSizes.NumBytesParaBuf     = WEBS_PARA_BUFFER_SIZE;
  BufferSizes.NumBytesFilenameBuf = WEBS_FILENAME_BUFFER_SIZE;
  BufferSizes.MaxRootPathLen      = WEBS_MAX_ROOT_PATH_LEN;
  //
  // Configure the size of the buffers used by the Webserver child tasks.
  //
  IP_WEBS_ConfigBufSizes(&BufferSizes);
  //
  // Give the stack some more memory to enable the dynamical memory
  // allocation for the Web server child tasks
  //
  IP_AddMemory(_aPool, sizeof(_aPool));
  //
  // Get a socket into listening state
  //
  hSockListen = socket(AF_INET, SOCK_STREAM, 0);
  if (hSockListen == SOCKET_ERROR) {
    while(1); // This should never happen!
  }
  memset(&InAddr, 0, sizeof(InAddr));
  InAddr.sin_family      = AF_INET;
  InAddr.sin_port        = htons(80);
  InAddr.sin_addr.s_addr = INADDR_ANY;
  bind(hSockListen, (struct sockaddr *)&InAddr, sizeof(InAddr));
  listen(hSockListen, BACK_LOG);
  //
  // Loop once per client and create a thread for the actual server
  //
  do {
Next:
    //
    // Wait for an incoming connection
    //
    hSock = 0;
    AddrLen = sizeof(Addr);
    if ((hSock = accept(hSockListen, &Addr, &AddrLen)) == SOCKET_ERROR) {
      continue;    // Error
    }
    //
    // Create server thread to handle connection.
    // If connection limit is reached, keep trying for some time before giving up
    // and outputting an error message
    //
    t0 = OS_GetTime32() + 1000;
    do {
      if (_ConnectCnt < MAX_CONNECTIONS) {
        for (i = 0; i < MAX_CONNECTIONS; i++) {
          U8 r;
          r = OS_IsTask(&_aWebTasks[i]);
          if (r == 0) {
```



```
            setsockopt(hSock, SOL_SOCKET, SO_RCVTIMEO, &Timeout, sizeof(Timeout));
            OS_CREATETASK_EX(&_aWebTasks[i], "Webserver Child", _WebServerChildTask,
                             TASK_PRIO_WEBS_CHILD, _aWebStacks[i], (void *)hSock);
            _AddToConnectCnt(1);
            goto Next;
          }
        }
      }
      //
      // Check time out
      //
      t = OS_GetTime32();
      if ((t - t0) > 0) {
        IP_WEBS_OnConnectionLimit(_Send, _Recv, (void*)hSock);
        _closesocket(hSock);
        break;
      }
      OS_Delay(10);
    } while(1);
  }  while(1);
}
```


9.14 IP_WEBS_OnConnectionLimit()

Description

Outputs an error message to the connected client.

Prototype

```
void IP_WEBS_OnConnectionLimit(IP_WEBS_tSend      pfSend,
                               IP_WEBS_tReceive   pfReceive,
                               void             * pConnectInfo);
```

Parameters

Parameter	Description
pfSend	Pointer on the function used to send.
pfReceive	Pointer on the function used to receive.
pConnectInfo	Pointer on the connection info (typically the socket).

Additional information

This function is typically called by the application if the connection limit is reached. Refer to `IP_WEBS_Process()` and `IP_WEBS_ProcessLast()` for further information.

Example

Pseudo code:

```
//
// Call IP_WEBS_Process() or IP_WEBS_ProcessLast() if multiple or just
// one more connection is available
//
do {
  if (NumAvailableConnections > 1) {
    IP_WEBS_Process();
    return;
  } else if (NumAvailableConnections == 1) {
    IP_WEBS_ProcessLast();
    return;
  }
  Delay();
} while (!Timeout)
//
// No connection available even after waiting => Output error message
//
IP_WEBS_OnConnectionLimit();
```


9.15 IP_WEBS_Process()

Description

Thread functionality of the web server. Returns when the connection has been handled or an error occurred.

Prototype

```
int IP_WEBS_Process(      IP_WEBS_tSend      pfSend,
                          IP_WEBS_tReceive   pfReceive,
                          void             * pConnectInfo,
                    const IP_FS_API        * pFS_API,
                    const WEBS_APPLICATION * pApplication);
```

Parameters

Parameter	Description
pfSend	Pointer to the function to be used by the server to send data to the client.
pfReceive	Pointer to the function to be used by the server to receive data from the client.
pConnectInfo	Connection info (typically socket handle).
pFS_API	API for file system operations.
pApplication	Web Server application settings.

Return value

= 1 Connection detached.
= 0 O.K.
< 0 Error.

Additional information

A connection detached return value means that an additional functionality like WebSockets has been enabled and the connection context has been dispatched to another resource and should therefore not be closed by the web server thread.

The following types are used as function pointers to the routines used to send and receive bytes from/to the client:

```
typedef int (*IP_WEBS_tSend)    ( const unsigned char * pData,
                                                  int    len,
                                                  void * pConnectInfo );
typedef int (*IP_WEBS_tReceive) ( const unsigned char * pData,
                                                  int    len,
                                                  void * pConnectInfo );
```

The send and receive functions should return the number of bytes successfully sent/received to/from the client. The pointer pConnectInfo is passed to the send and receive routines. It can be used to pass a pointer to a structure containing connection information or to pass a socket number. For details about the parameter pFS_API and the IP_WEBS_FS_API structure, refer to File system abstraction layer chapter. For details about the parameter pApplication and the WEBS_APPLICATION structure, refer to *Structure WEBS_APPLICATION* on page 158.

Refer to *IP_WEBS_ProcessLast* on page 96 and *IP_WEBS_OnConnectionLimit* on page 93 for further information.

9.16 IP_WEBS_ProcessEx()

Description

Thread functionality of the web server. Returns when the connection has been handled or an error occurred.

Prototype

```
int IP_WEBS_ProcessEx(      WEBS_CONTEXT * pContext,
                            void         * pConnectInfo,
                      const char         * sRootPath);
```

Parameters

Parameter	Description
pContext	Context keeping track of configured settings.
pConnectInfo	Connection info (typical socket handle).
sRootPath	String with the root path to use for HTML pages. Example: "/httpdocs"

Return value

= 1 Connection detached.
= 0 O.K.
< 0 Error.

Additional information

IP_WEBS_ProcessEx() is a more flexible version of IP_WEBS_Process() and should be used instead. The parameter pContext is a structure holding all the required function pointers to the routines used to send and receive bytes from/to the client, access the file system, allocate and free memory, etc.

It has to be initialized before usage. Refer to IP_WEBS_Init() for further information.

Refer to IP_WEBS_ProcessLastEx() and IP_WEBS_OnConnectionLimit() for further information about the thread handling of the Web server.

A connection detached return value means that an additional functionality like WebSockets has been enabled and the connection context has been dispatched to another resource and should therefore not be closed by the web server thread.

9.17 IP_WEBS_ProcessLast()

Description

Thread functionality of the web server.

Prototype

```
int IP_WEBS_ProcessLast(      IP_WEBS_tSend      pfSend,
                              IP_WEBS_tReceive   pfReceive,
                              void             * pConnectInfo,
                        const IP_FS_API        * pFS_API,
                        const WEBS_APPLICATION * pApplication);
```

Parameters

Parameter	Description
pfSend	Function pointer to socket send routine.
pfReceive	Function pointer to socket receive routine.
pConnectInfo	Connection info (typical socket handle).
pFS_API	API for file system operations.
pApplication	Web Server application settings.

Return value

= 1 Connection detached.
= 0 O.K.
< 0 Error.

Additional information

This is typically called for the last available connection. In contrast to IP_WEBS_Process(), this function closes the connection as soon as the command is completed in order to not block the last connection longer than necessary and avoid Connection-limit errors.

A connection detached return value means that an additional functionality like WebSockets has been enabled and the connection context has been dispatched to another resource and should therefore not be closed by the web server thread.

The following types are used as function pointers to the routines used to send and receive bytes from/to the client:

```
typedef int (*IP_WEBS_tSend)    (const unsigned char * pData,
                                 int    len,
                                 void * pConnectInfo);

typedef int (*IP_WEBS_tReceive) (const unsigned char * pData,
                                 int    len,
                                 void * pConnectInfo);
```

The send and receive functions should return the number of bytes successfully sent/received to/from the client. The pointer pConnectInfo is passed to the send and receive routines. It can be used to pass a pointer to a structure containing connection information or to pass a socket number. For details about the parameter pFS_API and the IP_WEBS_FS_API structure, refer to *File system abstraction layer* on page . For details about the parameter pApplication and the WEBS_APPLICATION structure, refer to *Structure WEBS_APPLICATION* on page 158. Refer to *IP_WEBS_Process* on page 94 and *IP_WEBS_OnConnectionLimit* on page 93 for further information.

9.18 IP_WEBS_ProcessLastEx()

Description

Processes a HTTP request of a client and closes the connection thereafter.

Prototype

```
int IP_WEBS_ProcessLastEx(      WEBS_CONTEXT * pContext,
                                void         * pConnectInfo,
                          const char         * sRootPath);
```

Parameters

Parameter	Description
pContext	Context keeping track of configured settings.
pConnectInfo	Connection info (typical socket handle).
sRootPath	String with the root path to use for HTML pages. Example: "/httpdocs"

Return value

= 1 Connection detached.
= 0 O.K.
< 0 Error.

Additional information

Thread functionality of the web server. This is typically called for the last available connection. IP_WEBS_ProcessLastEx() is a more flexible version of IP_WEBS_Process() and should be used instead. In contrast to IP_WEBS_Process(), this function closes the connection as soon as the command is completed in order to not block the last connection longer than necessary and avoid connection-limit errors.

The parameter pContext is a structure holding all the required function pointers to the routines used to send and receive bytes from/to the client, access the file system, allocate and free memory, etc.:

It has to be initialized before usage. Refer to IP_WEBS_Init() for further information.

Refer to IP_WEBS_ProcessLastEx() and IP_WEBS_OnConnectionLimit() for further information about the thread handling of the Web server.

A connection detached return value means that an additional functionality like WebSockets has been enabled and the connection context has been dispatched to another resource and should therefore not be closed by the web server thread.

9.19 IP_WEBS_Redirect()

Description

Sends a page from the file system to the browser instead of the regular content. This way a redirect depending on a result can be realized. The MIME type will be determined automatically from the file name or can be overridden.

Prototype

```
int IP_WEBS_Redirect(      WEBS_OUTPUT * pOutput,
                     const char        * sFileName,
                     const char        * sMIMEType);
```

Parameters

Parameter	Description
pOutput	Out context of the connection
sFileName	FileName of the file to redirect to. Will be used for MIME type recognition for header.
sMIMEType	MIME type to be used in header. May be NULL.

Return value

< 0 Error
= 0 O.K.

Additional information

The function shall only be called if no other data has been sent out before. The page that will be sent is parsed for CGIs the same way as it would be parsed when being directly being accessed by the browser. However the URL accessed by the browser will remain the same and the browser will show the same URL as address.

Example

```
/*********************************************************************
*
*       _CGI_Redirect
*/
static void _CGI_Redirect(WEBS_OUTPUT *pOutput, const char *sParameters) {
  IP_WEBS_Redirect(pOutput, "/index.htm", NULL);  // Redirect back to index
}
```


9.20 IP_WEBS_Reset()

Description

Resets internal variables for a clean start of the Web Server.

Prototype

```
void IP_WEBS_Reset(void);
```

Additional information

As the Web Server is not directly connected to the IP stack itself it can not register to the IP stacks de-initialize process. Once the stack has been de-initialized this routine shall be called before re-initializing the IP stack and using the Web Server again.

9.21 IP_WEBS_RetrieveUserContext()

Description

Retrieves a previously stored user context for the current output session.

Prototype

```
void *IP_WEBS_RetrieveUserContext(WEBS_OUTPUT * pOutput);
```

Parameters

Parameter	Description
pOutput	Out context of the connection.

Return value

Pointer to the previously stored user context or NULL.

Additional information

A user context retrieved will not reset the stored context. The user stored context remains valid until either set to NULL by the user or the connection being closed.

In case a browser reuses an already opened connection the user context is not reset. This can be used to identify a connection reuse or to exchange data within the same connection. It is user responsibility to make sure that the user context is set back to NULL by the last callback if this behavior is not desired.

9.22 IP_WEBS_SendFormattedString()

Description

Sends a string with placeholders that will be filled using `SEGGER_vsnprintfEx()`.

Prototype

```
int IP_WEBS_SendFormattedString(      WEBS_OUTPUT * pOutput,
                                const char        * sFormat,
                                                    ...);
```

Parameters

Parameter	Description
`pOutput`	Connection context.
`sFormat`	Formatted string that might contain placeholders.

Return value

Number of characters (without termination) that would have been stored if the buffer had been large enough.

Additional information

Allows sending a string containing placeholders without the need to have the final string created into a temporary buffer in the application. The output buffer is used directly which saves a buffer and avoids unnecessary copy operations from the application to the output buffer.

9.23 IP_WEBS_Send204NoContent()

Description

This function sends a "204 No Content" response for a captive portal test.

Prototype

```
int IP_WEBS_Send204NoContent(WEBS_OUTPUT * pOutput);
```

Parameters

Parameter	Description
pOutput	Out context of the connection.

Additional information

The function can be used to send a "204 No Content" response as answer to a captive portal test. A captive portal test is a test if the internet can be reached, for example when using a WiFi connection. For this a known domain that serves the resource "domain.tld/generate_204" is accessed. If the request succeeds, this means there is internet access.

If a captive portal test fails, this means there is no or very poor internet connectivity and the WiFi client might switch to another network automatically to test if it has better internet access.

Implementing a capitve portal test on an embedded device might be necessary if you run a WiFi AccessPoint on the target to prevent the WiFi client to disconnect from a network that has no internet access.

Typically the captive portal page served by the webserver is used together with distributing the targets IP address as primary DNS server via a DHCP server on the AP/target and accepting and faking all DNS answers to resolve to the target itself. This way all HTTP(S) connections will end being served the web pages from the target. This is often used with hotel WiFi networks to provide a login page.

9.24 IP_WEBS_SendHeader

Description

Outputs a valid HTML header.

Prototype

```
int IP_WEBS_SendHeader(      WEBS_OUTPUT * pOutput,
                       const char        * sFileName,
                       const char        * sMIMEType);
```

Parameters

Parameter	Description
pOutput	Out context of the connection.
sFileName	String containing the file name including extension to be written to the header.
sMIMEType	MIME type to be used in header. If set, sFileName will be ignored.

Return value

= 0 O.K., header sent.

Additional information

This function can be used in case automatically generating and sending a header has been switched off using IP_WEBS_ConfigSendVFileHeader() or IP_WEBS_ConfigSendVFileHookHeader(). Typically this is the first function you call from your callback generating content for a virtual file or a VFile hook registered callback providing content before you send any other data.

Depending on the MIME type used the browser may wait for new data forever if the connection is not closed after all data has been transferred. For example "application/octet-stream" will leave the browser waiting forever if transfer size is not sent in the header. Therefore, IP_WEBS_SendHeader() informs the Web server to close the connection after sending the data.

A typical header may look as follows:

```
HTTP/1.1 200 OK
Content-Type: image/gif
Server: emNet
Expires: 1 JAN 1995 00:00:00 GMT
Transfer-Encoding: chunked
```


9.25 IP_WEBS_SendHeaderEx()

Description

Outputs a valid HTML header.

Prototype

```
int IP_WEBS_SendHeaderEx(      WEBS_OUTPUT * pOutput,
                         const char        * sFileName,
                         const char        * sMIMEType,
                               U8            ReqKeepCon);
```

Parameters

Parameter	Description
pOutput	Out context of the connection.
sFileName	String containing the file name including extension to be written to the header.
sMIMEType	MIME type to be used in header. If set, sFileName will be ignored.
ReqKeepCon	• 1: If possible, keep connection open after data transmission. • 0: Close connection, after data transmission.

Return value

0 O.K., header sent, Connection will be kept open after transmission.
1 O.K., header sent, Connection will be closed after transmission.

Additional information

This function can be used in case automatically generating and sending a header has been switched off using IP_WEBS_ConfigSendVFileHeader() or IP_WEBS_ConfigSendVFileHookHeader(). Typically this is the first function you call from your callback generating content for a virtual file or a VFile hook registered callback providing content before you send any other data.

Depending on the MIME type used the browser may wait for new data forever if the connection is not closed after all data has been transferred. For example "application/octet-stream" will leave the browser waiting forever if transfer size is not sent in the header. Therefore, IP_WEBS_SendHeaderEx() informs the Web server to close the connection after sending the data.

9.26 IP_WEBS_SendLocationHeader()

Description

Sends a header with a redirection code for the browser.

Prototype

```
void IP_WEBS_SendLocationHeader(      WEBS_OUTPUT * pOutput,
                                const char        * sURI,
                                const char        * sCodeDesc);
```

Parameters

Parameter	Description
pOutput	Out context of the connection.
sURI	URI where to redirect the browser.
sCodeDesc	Any code and description that supports the “Location: <URI>” field like: • “301 Moved Permanently” • “302 Found” • “303 See Other”

Additional information

A redirect like 303 can be used to forward the user to the same page after POST data has been submitted, resulting in no resending of the POST data upon a refresh of the page.

9.27 IP_WEBS_SendMem()

Description

Sends data to a connected target.

Prototype

```
int IP_WEBS_SendMem(      WEBS_OUTPUT * pOutput,
                    const char        * s,
                          unsigned      NumBytes);
```

Parameters

Parameter	Description
pOutput	Pointer to the WEBS_OUTPUT structure.
s	Pointer to a memory location that should be transmitted.
NumBytes	Number of bytes that should be sent.

Return value

0 O.K.

9.28 IP_WEBS_SendString()

Description

Sends a zero-terminated string to a connected target.

Prototype

```
int IP_WEBS_SendString(      WEBS_OUTPUT * pOutput,
                       const char        * s);
```

Parameters

Parameter	Description
pOutput	Pointer to the WEBS_OUTPUT structure.
s	Pointer to a string that should be transmitted.

Return value

0 O.K.

9.29 IP_WEBS_SendStringEnc()

Description

Encodes and sends a zero-terminated string to a connected target.

Prototype

```
int IP_WEBS_SendStringEnc(      WEBS_OUTPUT * pOutput,
                          const char        * s);
```

Parameters

Parameter	Description
pOutput	Pointer to the WEBS_OUTPUT structure.
s	Pointer to a string that should be transmitted.

Return value

0 O.K.

Additional information

This function encodes the string s with URL encoding, which means that spaces are changed into "+" and special characters are encoded to hexadecimal values. Refer to [RFC 1738] for detailed information about URL encoding.

9.30 IP_WEBS_SendUnsigned()

Description

Sends an unsigned value to the client.

Prototype

```
int IP_WEBS_SendUnsigned(WEBS_OUTPUT * pOutput,
                         unsigned      v,
                         unsigned      Base,
                         int           NumDigits);
```

Parameters

Parameter	Description
pOutput	Pointer to the WEBS_OUTPUT structure.
v	Value that should be sent.
Base	Numerical base of the value v.
NumDigits	Number of digits that should be sent. 0 can be used as a wildcard.

Return value

0 O.K.

9.31 IP_WEBS_SetErrorPageCallback()

Description

Sets a callback that gets executed when an error page is sent.

Prototype

```
void IP_WEBS_SetErrorPageCallback(IP_WEBS_pfSendErrorPage   pf,
                                  void                    * pConfig);
```

Parameters

Parameter	Description
pf	Pointer to a callback function to send an error page.
pConfig	Reserved for future API extension. Use NULL.

Additional information

If the callback returns with WEBS_NO_ERROR_PAGE_SENT, the default content of the WEBS_XXX_PAGE define will be sent.

Custom error pages are only supported if the browser supports at least HTTP/1.1 for chunked encoding, for HTTP/1.0 only RAW encoding is available and the default content of the WEBS_XXX_PAGE define will be sent.

9.32 IP_WEBS_SetFileInfoCallback()

Description

Sets a callback that can be used to pass file system information about a requested file back to the web server.

Prototype

```
void IP_WEBS_SetFileInfoCallback(IP_WEBS_pfGetFileInfo pf);
```

Parameters

Parameter	Description
pf	Pointer to a callback function.

Additional information

The function can be used to change the default behavior of the Web server. If the file info callback function is set, the Web server calls it to retrieve the file information. The file information are used to decide how to handle the file and to build the HTML header. By default (no file info callback function is set), the Web server parses every file with the extension .htm to check if dynamic content is included; all requested files with the extension .cgi are recognized as virtual files. Beside of that, the Web server sends by default the expiration date of a Web site in the HTML header. The default expiration date (THU, 01 JAN 1995 00:00:00 GMT) is in the past, so that the requested Webpage will never be cached. This is a reasonable default for Web pages with dynamic content. If the callback function returns 0 for DateExp, the expiration date will not be included in the header. For static Webpages, it is possible to add the optional "Last-Modified" header field. The "Last-Modified" header field is not part of the header by default. Refer to Structure IP_WEBS_FILE_INFO for detailed information about the structure IP_WEBS_FILE_INFO.

Example

```
static void _GetFileInfo(const char * sFilename, IP_WEBS_FILE_INFO * pFileInfo){
  int v;
  //
  // .cgi files are virtual, everything else is not
  //
  v = IP_WEBS_CompareFilenameExt(sFilename, ".cgi");
  pFileInfo->IsVirtual = v ? 0 : 1;
  //
  // .htm files contain dynamic content, everything else is not
  //
  v = IP_WEBS_CompareFilenameExt(sFilename, ".htm");
  pFileInfo->AllowDynContent = v ? 0 : 1;
  //
  // If file is a virtual file or includes dynamic content,
  // get current time and date stamp as file time
  //
  pFileInfo->DateLastMod = _GetTimeDate();
  if (pFileInfo->IsVirtual || pFileInfo->AllowDynContent) {
    //
    // Set last-modified and expiration time and date
    //
    pFileInfo->DateExp     = _GetTimeDate(); // If
 "Expires" HTTP header field should

  // be transmitted, set expiration date.
  } else {
    pFileInfo->DateExp     = 0xEE210000; // Expiration far away (01 Jan 2099)
                                         // if content is static
  }
```



```
}
```


9.33 IP_WEBS_SetHeaderCacheControl()

Description

Sets the string to be sent for the cache-control field in the header. The field itself has to be part of the string, e.g.: "Cache-Control: no-cache
".

Prototype

```
void IP_WEBS_SetHeaderCacheControl(const char * sCacheControl);
```

Parameters

Parameter	Description
sCacheControl	Cache control field content.

9.34 IP_WEBS_SetUploadFileSystemAPI()

Description

Sets the FS API to use for file uploads.

Prototype

```
void IP_WEBS_SetUploadFileSystemAPI(const IP_FS_API * pFS_API);
```

Parameters

Parameter	Description
pFS_API	FS API of type IP_FS_API to use.

Additional information

Using different file system APIs for files to serve and files being uploaded can become handy as it allows serving static pages from a read only file system while using a writeable file system for uploads like a new firmware image.

9.35 IP_WEBS_SetUploadMaxFileSize()

Description

Sets the maximum file size per file that can be uploaded.

Prototype

```
void IP_WEBS_SetUploadMaxFileSize(U32 MaxFileSize);
```

Parameters

Parameter	Description
MaxFileSize	Maximum number of bytes per uploaded file.

9.36 IP_WEBS_StoreUserContext()

Description

Stores a user context for the current output session. The user context can be used to exchange data between CGI callbacks of the same connection.

Prototype

```
void IP_WEBS_StoreUserContext(WEBS_OUTPUT * pOutput,
                              void        * pContext);
```

Parameters

Parameter	Description
pOutput	Out context of the connection.
pContext	User context to store.

Additional information

Sometimes it might be necessary to exchange information between several callbacks that will be called one after another when a Web page is processed or form data is submitted. The user can use this mechanism to store data into the current connection context in one callback and retrieve the data from another callback of the same connection. Callbacks such as CGIs will be called in the order they are referenced by the Web page. Therefore the order of their accesses is known and can be used in dynamic memory allocation.

Examples

A sample using pseudo code is shown below.

```
/*********************************************************************
*
*       _CGI_1
*
*  Notes
*    This is the first callback accessed for the operation requested
*    by the browser. This is a perfect place to allocate some memory.
*/
static void _CGI_1(WEBS_OUTPUT *pOutput, const char *sParameters, const char *sValue)
{
  char *s;

  s = (char*)OS_malloc(13);                    // Allocate memory for data as
                                               // data has to remain valid outside
                                               // of this routine.
  strcpy(s, "Hello world!");                   // Fill with data
  IP_WEBS_StoreUserContext(pOutput, (void*)s); // Store pointer to text for other
                                               // callback to access.
}

/*********************************************************************
*
*       _CGI_2
*
*  Notes
*    This is the last callback accessed for the operation requested
*    by the browser. This is a perfect place to free the previously
*    allocated memory.
*/
static void _CGI_2(WEBS_OUTPUT *pOutput, const char *sParameters, const char *sValue)
{
  char *s;

  s = (char*)IP_WEBS_RetrieveUserContext(pOutput);  // Retrieve previously stored
                                                    // data.
  printf("%s", s);                                  // Output data.
  IP_WEBS_StoreUserContext(pOutput, NULL);          // Invalidate user context.
  OS_free((void*)s);                                // Free allocated memory.
```



```
}
```


9.37 IP_WEBS_AddVFileHook()

Description

Registers a function table containing callbacks to check and serve simple virtual file content that is not further processed by the Web server.

Prototype

```
void IP_WEBS_AddVFileHook(WEBS_VFILE_HOOK        * pHook,
                          WEBS_VFILE_APPLICATION * pVFileApp,
                          U8                       ForceEncoding);
```

Parameters

Parameter	Description
pHook	Pointer to a structure of WEBS_VFILE_HOOK
pVFileApp	Pointer to a structure of WEBS_VFILE_APPLICATION
ForceEncoding	When set to HTTP_ENCODING_RAW chunked encoding will not be used. Necessary for some implementations such as UPnP. Further options might be added later.

Additional information

The function can be used to serve simple dynamically generated content for a requested file name that is simply sent back as generated by the application and is not further processed by the Web server.

Refer to *Structure WEBS_VFILE_HOOK* on page 161 for detailed information about the structure WEBS_VFILE_HOOK. Refer to *Structure WEBS_VFILE_APPLICATION* on page 160 for detailed information about the structure WEBS_VFILE_APPLICATION.

Example

```
/* Excerpt from IP_WebserverSample_UPnP.c */
/*********************************************************************
*
*       _UPnP_GenerateSend_upnp_xml
*
* Function description
*   Send the content for the requested file using the callback provided.
*
* Parameters
*   pContextIn     - Send context of the connection processed for
*                    forwarding it to the callback used for output.
*   pf             - Function pointer to the callback that has to be
*                    for sending the content of the VFile.
*     pContextOut    - Out context of the connection processed.
*     pData          - Pointer to the data that will be sent
*     NumBytes       - Number of bytes to send from pData. If NumBytes
*                      is passed as 0 the send function will run a strlen()
*                      on pData expecting a string.
*
* Notes
*   (1) The data does not need to be sent in one call of the callback
*       routine. The data can be sent in blocks of data and will be
*       flushed out automatically at least once returning from this
*       routine.
*/
static void _UPnP_GenerateSend_upnp_xml(void * pContextIn,
  void (*pf) (void * pContextOut, const char * pData, unsigned NumBytes))
{
  char ac[128];
```



```
  pf(pContextIn, "<?xml version=\"1.0\"?>\r\n"
                 "<root xmlns=\"urn:schemas-upnp-org:device-1-0\">\r\n"
                   "<specVersion>\r\n"
                     "<major>1</major>\r\n"
                     "<minor>0</minor>\r\n"
                     "</specVersion>\r\n", 0);
}

/* Excerpt from IP_WebserverSample_UPnP.c */
//
// UPnP webserver VFile hook
//
static WEBS_VFILE_HOOK _UPnP_VFileHook;

/* Excerpt from IP_WebserverSample_UPnP.c */
/*********************************************************************
*
*       _UPnP_CheckVFile
*
* Function description
*   Check if we have content that we can deliver for the requested
*   file using the VFile hook system.
*
* Parameters
*   sFileName      - Name of the file that is requested
*   pIndex         - Pointer to a variable that has to be filled with
*                    the index of the entry found in case of using a
*                    filename<=>content list.
*                    Alternative all comparisons can be done using the
*                    filename. In this case the index is meaningless
*                    and does not need to be returned by this routine.
*
* Return value
*   0              - We do not have content to send for this filename,
*                    fall back to the typical methods for retrieving
*                    a file from the Web server.
*   1              - We have content that can be sent using the VFile
*                    hook system.
*/
static int _UPnP_CheckVFile(const char * sFileName, unsigned * pIndex) {
  unsigned i;

  //
  // Generated VFiles
  //
  if (strcmp(sFileName, "/upnp.xml") == 0) {
    return 1;
  }
  //
  // Static VFiles
  //
  for (i = 0; i < SEGGER_COUNTOF(_VFileList); i++) {
    if (strcmp(sFileName, _VFileList[i].sFileName) == 0) {
      *pIndex = i;
      return 1;
    }
  }
  return 0;
}

/*********************************************************************
*
*       _UPnP_SendVFile
*
* Function description
*   Send the content for the requested file using the callback provided.
*
* Parameters
```



```
*   pContextIn     - Send context of the connection processed for
*                    forwarding it to the callback used for output.
*   Index          - Index of the entry of a filename<=>content list
*                    if used. Alternative all comparisons can be done
*                    using the filename. In this case the index is
*                    meaningless. If using a filename<=>content list
*                    this is faster than searching again.
*   sFileName      - Name of the file that is requested. In case of
*                    working with the Index this is meaningless.
*   pf             - Function pointer to the callback that has to be
*                    for sending the content of the VFile.
*     pContextOut    - Out context of the connection processed.
*     pData          - Pointer to the data that will be sent
*     NumBytes       - Number of bytes to send from pData. If NumBytes
*                      is passed as 0 the send function will run a strlen()
*                      on pData expecting a string.
*/
static void _UPnP_SendVFile( void* pContextIn,
  unsigned Index,
  const char* sFileName,
  void (*pf) ( void* pContextOut, const char* pData, unsigned NumBytes ))
{

  (void)sFileName;
  //
  // Generated VFiles
  //
  if (strcmp(sFileName, "/upnp.xml") == 0) {
    _UPnP_GenerateSend_upnp_xml(pContextIn, pf);
    return;
  }
  //
  // Static VFiles
  //
  pf(pContextIn, _VFileList[Index].pData, _VFileList[Index].NumBytes);
}

static WEBS_VFILE_APPLICATION _UPnP_VFileAPI = {
  _UPnP_CheckVFile,
  _UPnP_SendVFile
};

/* Excerpt from IP_WebserverSample_UPnP.c */

/*********************************************************************
*
*       MainTask
*/
void MainTask(void);
void MainTask(void) {
  //
  // Activate UPnP with VFile hook for needed XML files
  //
  IP_WEBS_AddVFileHook(&_UPnP_VFileHook, &_UPnP_VFileAPI, HTTP_ENCODING_RAW);
}
```


9.38 IP_WEBS_CompareFilenameExt()

Description

Checks if the given filename has the given extension

Prototype

```
char IP_WEBS_CompareFilenameExt(const char * sFilename,
                                const char * sExt);
```

Parameters

Parameter	Description
sFilename	Null-terminated filename, such as “Index.html”
sExt	Null-terminated filename extension with dot, such as “.html”

Return value

= 0 Match.
≠ 0 Mismatch.

Additional information

The test is case-sensitive, meaning:

```
IP_WEBS_CompareFilenameExt("Index.html", ".html")     ---> Match
IP_WEBS_CompareFilenameExt("Index.htm",  ".html")     ---> Mismatch
IP_WEBS_CompareFilenameExt("Index.HTML", ".html")     ---> Mismatch
IP_WEBS_CompareFilenameExt("Index.html", ".HTML")     ---> Mismatch
```


9.39 IP_WEBS_ConfigSendVFileHeader()

Description

Configures behavior of automatically sending a header containing a MIME type associated to the requested files extension based on an internal list for a requested virtual file.

Prototype

```
void IP_WEBS_ConfigSendVFileHeader(U8 OnOff);
```

Parameters

Parameter	Description
`OnOff`	Default: On. • 0: Off, header will not be automatically generated and sent. • 1: On, header will be automatically generated.

Additional information

In case you decide not to let the Web server generate a header with the best content believed to be known you will either have to completely send a header on your own or sending a header using the function `IP_WEBS_SendHeader()`. Sending a header has to be done before sending any other content.

9.40 IP_WEBS_ConfigSendVFileHookHeader()

Description

Configures behavior of automatically sending a header containing a MIME type associated to the requested files extension based on an internal list for a requested file being served by a registered VFile hook.

Prototype

```
void IP_WEBS_ConfigSendVFileHookHeader(U8 OnOff);
```

Parameters

Parameter	Description
`OnOff`	Default: On. • 0: Off, header will not be automatically generated and sent. • 1: On, header will be automatically generated.

Additional information

In case you decide not to let the Web server generate a header with the best content believed to be known you will either have to completely send a header on your own or sending a header using the function `IP_WEBS_SendHeader()`. Sending a header has to be done before sending any other content.

9.41 IP_WEBS_DecodeAndCopyStr()

Description

Checks if a string includes url encoded characters, decodes the characters and copies them into destination buffer.

Prototype

```
void IP_WEBS_DecodeAndCopyStr(      char * pDest,
                                    int    DestLen,
                              const char * pSrc,
                                    int    SrcLen);
```

Parameters

Parameter	Description
pDest	Buffer to store the decoded string.
DestLen	Size of the destination buffer.
pSrc	Source string that should be decoded.
SrcLen	Size of the source string.

Additional information

Destination string is 0-terminated. Source and destination buffer can be identical.

pSrc	SrcLen	pDest	DestLen
“FirstName=J%F6rg”	16	“FirstName=Jörg”	15
“FirstName=John”	14	“FirstName=John”	15

9.42 IP_WEBS_DecodeString()

Description

Checks if a string includes url encoded characters, decodes the characters.

Prototype

```
int IP_WEBS_DecodeString(const char * s);
```

Parameters

Parameter	Description
s	Pointer to a zero-terminated string.

Return value

< 0 Error.
= 0 String does not include url encoded characters. No change.
> 0 Length of the decoded string excluding the terminating null character.

9.43 IP_WEBS_GetDecodedStrLen()

Description

Returns the length of a HTML encoded string when decoded excluding null character.

Prototype

```
int IP_WEBS_GetDecodedStrLen(const char * s,
                             int    Len);
```

Parameters

Parameter	Description
s	Pointer to a string.
Len	Length of the source string excluding terminating null character.

Return value

< 0 Error.
> 0 Length of decoded string excluding the terminating null character.

9.44 IP_WEBS_GetNumParas()

Description

Returns the number of parameters/value pairs.

Prototype

```
int IP_WEBS_GetNumParas(const char * sParameters);
```

Parameters

Parameter	Description
sParameters	Null-terminated parameter string, such as “Var1=Val1&Var2=Val2”

Return value

= -1 No parameter/value pair / error in parameter string detected
> 0 Number of parameter/value pairs

Additional information

Parameters are separated from values by a “=”. If a string includes more as one parameter/value pair, the parameter/value pairs are separated by a “&”. For example, if the virtual file Send.cgi gets two parameters, the string should be similar to the following: “Send.cgi?FirstName=Foo&LastName=Bar”.

sParameter is in this case FirstName=Foo&LastName=Bar. If you call `IP_WEBS_GetNumParas()` with this string, the return value will be 2.

9.45 IP_WEBS_GetParaValue()

Description

Parses a string for valid paramter/value pairs and writes results in buffer.

Prototype

```
int IP_WEBS_GetParaValue(const char * sBuffer,
                         int    ParaNum,
                         char * sPara,
                         int    ParaLenIn,
                         char * sValue,
                         int    ValueLenIn);
```

Parameters

Parameter	Description
sBuffer	Zero-terminated string that should be parsed
ParaNum	Zero-based index of the parameter/value pairs.
sPara	Buffer to store the parameter name. (Optional, can be NULL.)
ParaLenIn	Size of the buffer to store the parameter. (0 if sPara is NULL.)
sValue	Buffer to store the value. (Optional, can be NULL.)
ValueLenIn	Size of the buffer to store the value. (0 if sValue is NULL.)

Return value

= 0 OK
> 0 Error

Additional information

A valid string is in the following format:

```
<Param0>=<Value0>&<Param1>=<Value1>& ... <Paramn>=<Valuen>
```

If the parameter value string is FirstName=John&LastName=Doo and parameter 0 should be copied, sPara will be FirstName and sValue John. If parameter 1 should be copied, sPara will be LastName and sValue Doo.

9.46 IP_WEBS_GetParaValuePtr()

Description

Parses a string for valid parameter/value pairs and returns a pointer to the requested parameter and the length of the parameter string without termination.

Prototype

```
int IP_WEBS_GetParaValuePtr(const char  * sBuffer,
                            int     ParaNum,
                      const char ** ppPara,
                            int   * pParaLen,
                      const char ** ppValue,
                            int   * pValueLen);
```

Parameters

Parameter	Description
sBuffer	Zero-terminated parameter/value string that should be parsed.
ParaNum	Zero-based index of the parameter/value pairs.
ppPara	Pointer to the pointer locating the start of the requested parameter name. (Optional, can be NULL.)
pParaLen	Pointer to a buffer to store the length of the parameter name without termination. (Optional, can be NULL.)
ppValue	Pointer to the pointer locating the start of the requested parameter value. (Optional, can be NULL.)
pValueLen	Pointer to a buffer to store the length of the parameter value without termination. (Optional, can be NULL.)

Return value

= 0 OK
> 0 Error

Additional information

A valid string is in the following format:

```
<Param0>=<Value0>&<Param1>=<Value1>& ... <Paramn>=<Valuen>
```

This function can be used in case you simply want to check or use the parameters passed by the client without modifying them. Depending on your application this might save you a lot of stack that otherwise would have to be wasted for copying the same data that is already perfectly present to another location. This saves execution time as of course the data will not have to be copied.

Example

```
/* Excerpt from IP_WebserverSample.c */
/*********************************************************************
*
*       _callback_CGI_Send
*/
static void _callback_CGI_Send(WEBS_OUTPUT * pOutput, const char * sParameters) {
        int    r;
  const char * pFirstName;
        int    FirstNameLen;
  const char * pLastName;
        int    LastNameLen;
```



```
  IP_WEBS_SendString(pOutput,  "<HTML><HEAD><TITLE>Virtual  file  example</TITLE></HEAD>");
  IP_WEBS_SendString(pOutput, "<style type=\"text/css\"> \
    H1, H1, H2, H4 { color: white; font-family: Helvetica; } \
    PRE {  color: white;  margin-left: 2%; ; font-size=150%} \
    BODY{padding:0px; margin:0px; text-align:center; font-family:Verdana, Helvetica,
sans-serif; background:#6699CC url(bg.png) repeat-x; font-size:11px; color:white } \
    A:link { font-weight:bold; color:white; text-decoration:none; } \
    A:visited { font-weight:bold; color:silver; text-decoration:none; } \
    A:focus { font-weight:bold; color:white; text-decoration:underline; } \
    A:hover { font-weight:bold; color:silver; text-decoration:none; } \
    A:active { font-weight:bold; color:white; text-decoration:underline; }\
    </style>");
  IP_WEBS_SendString(pOutput, "<BODY><CENTER><HR><H1>Virtual file example</H1><HR></
CENTER><BR><BR><BR>");
  r  = IP_WEBS_GetParaValuePtr(sParameters, 0, NULL, 0, &pFirstName, &FirstNameLen);
  r |= IP_WEBS_GetParaValuePtr(sParameters, 1, NULL, 0, &pLastName, &LastNameLen);
  if (r == 0) {
    IP_WEBS_SendString(pOutput, "First name: ");
    IP_WEBS_SendMem(pOutput, pFirstName, FirstNameLen);
    IP_WEBS_SendString(pOutput, "<BR>Last name: ");
    IP_WEBS_SendMem(pOutput, pLastName, LastNameLen);
  } else {
    IP_WEBS_SendString(pOutput, "<BR>Error!");
  }
  IP_WEBS_SendString(pOutput, "<BR><BR><BR>");
  IP_WEBS_SendString(pOutput, "<HR><CENTER><A HREF=\"index.htm\">Back to main</A></
CENTER><IMG SRC=\"logo.gif\" ALT=\"Segger logo\">   SEGGER Microcontroller
GmbH &amp; Co. KG   <A HREF=\"http://www.segger.com\">www.segger.com</A></
BODY></HTML>");
}
```


9.47 IP_WEBS_GetConnectInfo()

Description

Retrieves the connect info that has been passed to the Web Server process function from the application.

Prototype

```
void *IP_WEBS_GetConnectInfo(WEBS_OUTPUT * pOutput);
```

Parameters

Parameter	Description
pOutput	Out context of the connection.

Return value

Connection info passed to process function.

9.48 IP_WEBS_GetURI()

Description

Retrieves the requested URI up to the '?' char or the "full URI" including '?' character and all characters up to the next space.

Prototype

```
char *IP_WEBS_GetURI(WEBS_OUTPUT * pOutput,
                     char          GetFullURI);
```

Parameters

Parameter	Description
pOutput	Out context of the connection.
GetFullURI	Switch to select between URI and "full URI". URI contains the resource address up to any delimiter such as "?". The "full URI" contains the complete resource address accessed up to the next whitespace after the resource address including "?" and following characters. 0: URI. 1: "full URI"

Return value

Pointer to URI or "full URI". NULL in case "full URI" was requested but no buffer was available to store the "full URI".

Additional information

To support storing the "full URI" the define WEBS_URI_BUFFER_SIZE needs to be set. If it is not set or its size is too small, requesting the "full URI" will always return NULL.

9.49 IP_WEBS_UseRawEncoding()

Description

Overrides the previously selected encoding of the Web Server to use RAW encoding. This also means closing the connection after answering the request.

Prototype

```
void IP_WEBS_UseRawEncoding(WEBS_OUTPUT * pOutput);
```

Parameters

Parameter	Description
pOutput	Out context of the connection.

9.50 IP_WEBS_AUTH_DIGEST_CalcHA1()

Description

Calculates the auth digest MD5 hash for the HA1 hash value to use in the `WEBS_ACCESS_CONTROL` table.

Prototype

```
void IP_WEBS_AUTH_DIGEST_CalcHA1(const char     * pInput,
                                 unsigned   InputLen,
                                 char     * pBuffer,
                                 unsigned   BufferSize);
```

Parameters

Parameter	Description
`pInput`	Input string of “User:realm:Pass”.
`InputLen`	Length of input string without termination.
`pBuffer`	Pointer to buffer that is at least 32 bytes to hold the unterminated ASCII MD5 hash.
`BufferSize`	32 bytes or more.

9.51 IP_WEBS_AUTH_DIGEST_GetURI()

Description

Retrieves the URI sent in the client authorization.

Prototype

```
void IP_WEBS_AUTH_DIGEST_GetURI(WEBS_OUTPUT             * pOutput,
                                WEBS_AUTH_DIGEST_OUTPUT * pDigestOutput,
                                char                    * pBuffer,
                                unsigned                * pNumBytes);
```

Parameters

Parameter	Description
pOutput	Connection context.
pDigestOutput	Auth digest context.
pBuffer	Where to store the URI.
pNumBytes	in Size of buffer. out Number of characters stored.

Additional information

This function is only valid to be used from an auth digest callback and can be used if no new nonce is to be generated (which means the client has sent an authorization line) to retrieve this field value for a look up e.g. in a nonce cache.

9.52 IP_WEBS_GetProtectedPath()

Description

Retrieves the protected path that is tried to access by the current request.

Prototype

```
char *IP_WEBS_GetProtectedPath(WEBS_OUTPUT * pOutput);
```

Parameters

Parameter	Description
pOutput	Output context

Return value

No protected path is accessed: NULL. Protected path is accessed : Path string.

9.53 IP_WEBS_UseAuthDigest()

Description

Use auth digest instead of auth basic for HTTP authentication.

Prototype

```
void IP_WEBS_UseAuthDigest(const WEBS_AUTH_DIGEST_APP_API * pAPI);
```

Parameters

Parameter	Description
pAPI	Auth digest application API to use.

Additional information

Refer to *Structure WEBS_AUTH_DIGEST_APP_API* on page 170 for more information.

9.54 IP_WEBS_METHOD_AddHook()

Description

Registers a callback to serve special content upon call of a METHOD.

Prototype

```
void IP_WEBS_METHOD_AddHook(      WEBS_METHOD_HOOK * pHook,
                                  IP_WEBS_pfMethod   pf,
                            const char             * sURI);
```

Parameters

Parameter	Description
pHook	Pointer to a structure of WEBS_METHOD_HOOK.
pf	Callback function for special METHOD implementation
sURI	Location that will execute the registered METHOD hook instead of handling it like a typical web site.

Additional information

The function can be used to implement Web applications that need to make use of METHODs in a special way such as REST (REpresentational State Transfer) that uses GET and POST in a different way they are typically used by a Web server. Refer to *Structure WEBS_METHOD_HOOK* on page 164 for detailed information about the structure WEBS_METHOD_HOOK. Refer to *Callback IP_WEBS_pfMethod* on page 172 for detailed information about the callback parameters of IP_WEBS_pfMethod.

Typically one URI on the server is used to serve such a special need and this function allows redefining METHODs for a specific URI for such cases. Locations within this URI such as /URI/1 in case /URI has been defined for the hook are served by the hook as well. In case further hooks are placed inside paths of other hooks the hook with the deepest path matching the requested URI will be used.

Example

```
/* Excerpt from IP_WebserverSample.c */
/*********************************************************************
*
*       _REST_cb
*
*  Function description
*    Callback for demonstrational REST implementation using a METHOD
*    hook. As there is no clearly defined standard for REST this
*    implementation shall act as sample and starting point on how
*    REST support could be implemented by you.
*
*  Parameters
*    pContext      - Context for incoming data
*    pOutput       - Context for outgoing data
*    sMethod       - String containing used METHOD
*    sAccept       - NULL or string containing value of "Accept" field of HTTP header
*    sContentType  - NULL or string containing value of "Content-Type" field of
*                    HTTP header
*    sResource     - String containing URI that was accessed
*    ContentLen    - 0 or length of data submitted by client
*
*  Return value
*    0             - O.K.
*    Other         - Error
*/
static int _REST_cb(       void        *pContext,
                           WEBS_OUTPUT *pOutput,
                     const char        *sMethod,
                     const char        *sAccept,
                     const char        *sContentType,
```



```
                      const char        *sResource,
                            U32          ContentLen ) {
  int  Len;
  char acAccept[128];
  char acContentType[32];

  //
  // Strings located at sAccept and sContentType need to be copied to
  // another location before calling any other Web Server API as they
  // will be overwritten.
  //
  if (sAccept) {
    _CopyString(acAccept, sAccept, sizeof(acAccept));
  }
  if (sContentType) {
    _CopyString(acContentType, sContentType, sizeof(acContentType));
  }
  //
  // Send implementation specific header to client
  //
  IP_WEBS_SendHeader(pOutput, NULL, "application/REST");
  //
  // Output information about the METHOD used by the client
  //
  IP_WEBS_SendString(pOutput, "METHOD:       ");
  IP_WEBS_SendString(pOutput, sMethod);
  IP_WEBS_SendString(pOutput, "\n");
  //
  // Output information about which URI has been accessed by the client
  //
  IP_WEBS_SendString(pOutput, "URI:          ");
  IP_WEBS_SendString(pOutput, sResource);
  IP_WEBS_SendString(pOutput, "\n");
  //
  // Output information about "Accept" field given in header sent by client, if any
  //
  if (sAccept) {
    IP_WEBS_SendString(pOutput, "Accept:       ");
    IP_WEBS_SendString(pOutput, acAccept);
    IP_WEBS_SendString(pOutput, "\n");
  }
  //
  // Output information about "Content-Type" field given in header sent by
  // client, if any
  //
  if (sContentType) {
    IP_WEBS_SendString(pOutput, "Content-Type: ");
    IP_WEBS_SendString(pOutput, acContentType);
  }
  //
  // Output content sent by client, or content previously sent by client that has
  // been saved
  //
  if ((_acRestContent[0] || ContentLen) && sContentType) {
    IP_WEBS_SendString(pOutput, "\n");
  }
  if (_acRestContent[0] || ContentLen) {
    IP_WEBS_SendString(pOutput, "Content:\n");
  }
  if (ContentLen) {
    //
    // Update saved content
    //
    Len = SEGGER_MIN(sizeof(_acRestContent) - 1, ContentLen);
    IP_WEBS_METHOD_CopyData(pContext, _acRestContent, Len);
    _acRestContent[Len] = 0;
  }
  if (_acRestContent[0]) {
    IP_WEBS_SendString(pOutput, _acRestContent);
  }
  return 0;
}

/*********************************************************************
*
*       MainTask
```



```
*/
void MainTask(void);
void MainTask(void) {
  //
  // Register URI "http://<ip>/REST" for demonstrational REST implementation
  //
  IP_WEBS_METHOD_AddHook(&_MethodHook, _REST_cb, "/REST");
}
```


9.55 IP_WEBS_METHOD_CopyData()

Description

This function can be called from a METHOD hook callback to copy the received amount of data as stated by the "Content-Length" field of the header.

Prototype

```
int IP_WEBS_METHOD_CopyData(void     * pContext,
                            void     * pBuffer,
                            unsigned   NumBytes);
```

Parameters

Parameter	Description
pContext	METHOD context for incoming data.
pBuffer	Buffer to store data into.
NumBytes	Number of bytes to read.

Return value

< 0	Error
= 0	Connection closed
> 0	Number of bytes read

Additional information

The function can be used to implement Web applications that need to make use of METHODs in a special way such as REST (REpresentational State Transfer) that uses GET and POST in a different way they are typically used by a Web server. Refer to *Structure WEBS_METHOD_HOOK* on page 164 for detailed information about the structure WEBS_METHOD_HOOK. Refer to *Callback IP_WEBS_pfMethod* on page 172 for detailed information about the callback parameters of IP_WEBS_pfMethod.

Typically one URI on the server is used to serve such a special need and this function allows redefining METHODs for a specific URI for such cases. Locations within this URI such as /URI/1 in case /URI has been defined for the hook are served by the hook as well. In case further hooks are placed inside paths of other hooks the hook with the deepest path matching the requested URI will be used.

9.56 IP_WEBS_WEBSOCKET_AddHook()

Description

This function registers a WebSocket hook that can be used to connect to a WebSocket implementation.

Prototype

```
void IP_WEBS_WEBSOCKET_AddHook(      WEBS_WEBSOCKET_HOOK   * pHook,
                               const IP_WEBS_WEBSOCKET_API * pAPI,
                               const char                  * sURI,
                               const char                  * sProto);
```

Parameters

Parameter	Description
pHook	Pointer to a structure of WEBS_WEBSOCKET_HOOK.
pAPI	Web Server WebSocket API layer to use.
sURI	Resource to register for WebSocket usage instead of handling it like a typical website.
sProto	WebSocket protocol name. If no sub protocol is desired, use an empty string. Not optional!

Additional information

This function is only meant to register an URI for using it with the WebSocket protocol. A WebSocket stack is required for handling the protocol itself.

9.57 IP_WEBS_HEADER_AddFieldHook()

Description

Registers a callback that will be notified if the registered header field is received.

Prototype

```
void IP_WEBS_HEADER_AddFieldHook(      WEBS_OUTPUT            * pOutput,
                                       WEBS_HEADER_FIELD_HOOK * pHook,
                                       IP_WEBS_pfHeaderField    pf,
                                 const char                   * sField);
```

Parameters

Parameter	Description
pOutput	Connection context.
pHook	Pointer to a structure of WEBS_HEADER_FIELD_HOOK.
pf	Callback function that gets notified if the registered header field is received.
sField	Header field to catch, e.g.: "Cookie: ".

Additional information

This function can be used to notify a callback for example if an HTTP cookie is received via the header field "Cookie: ". Multiple hooks can be registered to the same context but each header is only allowed to be registered once. Hooks need to be registered for each new request again. Use IP_WEBS_AddProgressHook() if unsure when a request begins/ends. Other callback mechanism can be used as well.

The callback pf can return either WEBS_OK to remove the current line with the header field from the input buffer or return with WEBS_HEADER_FIELD_UNTOUCHED if, and only if, the line has not been modified by reading it form the input buffer or something else. In case of WEBS_HEADER_FIELD_UNTOUCHED the line will then be processed as it would normally by the server.

9.58 IP_WEBS_HEADER_CopyData()

Description

Copies the requested amount of data from line with the current header field.

Prototype

```
int IP_WEBS_HEADER_CopyData(WEBS_OUTPUT * pOutput,
                            void        * pBuffer,
                            unsigned      BufferSize,
                            unsigned    * pNumBytesLeft);
```

Parameters

Parameter	Description
pOutput	Connection context.
pBuffer	Buffer where to store data from input buffer into.
BufferSize	Buffer size.
pNumBytesLeft	Where to store the number of bytes left to read from the current line. Can be NULL.

Return value

< 0 Error.
≥ 0 Number of bytes read.

Additional information

This function can be called multiple times until pNumBytesLeft tells you that the end of the line has been reached.

9.59 IP_WEBS_HEADER_GetFindToken()

Description

Searches the current header line for a token and copies its value into a buffer.

Prototype

```
int IP_WEBS_HEADER_GetFindToken(      WEBS_OUTPUT * pOutput,
                                const char        * sToken,
                                      int           TokenLen,
                                      char        * pBuffer,
                                      int           BufferSize);
```

Parameters

Parameter	Description
pOutput	Connection context.
sToken	Token to find in current header line e.g. “pagecnt”.
TokenLen	Length of token to find without string termination.
pBuffer	Buffer where to store the value if the token is found. The value is not automatically string terminated. Can be NULL to check for a token and its length.
BufferSize	Size of buffer at pValBuffer .

Return value

< 0	Token not found.
≥ 0	Token found, length of value (0 if token without value “HttpOnly;”). If pBuffer is not big enough, the copy is aborted but but the length of the value if it would fit is returned.

Additional information

This function is meant to be used from an IP_WEBS_HEADER_AddFieldHook() callback once a header field to catch has been found. This function is an alternative to using IP_WEBS_HEADER_CopyData() as it allows to copy only the token you are looking for instead of copying the whole line from the input buffer. It does not only allow to process a single token with less memory but also allows to check for tokens more easily.

9.60 IP_WEBS_HEADER_SetCustomFields()

Description

Sets a custom string of header fields to be included with the next header sent.

Prototype

```
void IP_WEBS_HEADER_SetCustomFields(      WEBS_OUTPUT * pOutput,
                                    const char        * sAddFields);
```

Parameters

Parameter	Description
pOutput	Connection context.
sAddFields	String with additional header fields. The pointer has to remain valid until the request has been completely processed (or at least the header has been sent). Use IP_WEBS_AddProgressHook() to get notified about the END event of the request.

Additional information

This function can be used to set custom header fields to be sent like a "Set-Cookie:" header field to create an HTTP cookie.

Remark on usage for cookies: Only one cookie token=value pair can be set in one "Set-Cookie" line. To add multiple token=value pairs you have to add multiple "Set-Cookie" lines.

9.61 IP_WEBS_AddUpload()

Description

Adds the upload functionality to the Web server

Prototype

```
int IP_WEBS_AddUpload(void);
```

Return value

1	Upload added.
0	Upload not added. (`WEBS_SUPPORT_UPLOAD = 0`)

Additional information

A real file system like emFile is required to upload files. Calling this function has no effect if the compile time switch `WEBS_SUPPORT_UPLOAD` is not defined to 1. In library versions of the Web server `WEBS_SUPPORT_UPLOAD` is always defined to 1.

9.62 IP_WEBS_ChangeUploadMaxFileSize()

Description

This function allows to change the maximum file size allowed for the file that is in upload.

Prototype

```
void IP_WEBS_ChangeUploadMaxFileSize(WEBS_OUTPUT * pOutput,
                                     U32           MaxFileSize);
```

Parameters

Parameter	Description
pOutput	Connection context.
MaxFileSize	Maximum number of bytes to accept for the file upload.

Additional information

This function is only valid to be called from the IP_WEBS_pfModifyTempFilename callback. It can be used after peeking on the file name that is uploaded and identifying its extension to limit the maximum upload size before actually receiving the data. This can be used to prevent large image uploads for avatars and other things that do not make sense for the application.

9.63 IP_WEBS_GetUploadFilename()

Description

Copies the original filename of an upload into the given buffer.

Prototype

```
unsigned IP_WEBS_GetUploadFilename(WEBS_OUTPUT * pOutput,
                                   char        * pBuffer,
                                   U32           BufferSize);
```

Parameters

Parameter	Description
pOutput	Connection context.
pBuffer	Buffer where to store the filename.
BufferSize	Size of destination buffer.

Return value

Number of characters copied including string termination: > 0 Error : = 0

Additional information

This function is only valid to be called from the IP_WEBS_pfModifyTempFilename callback.

9.64 IP_WEBS_SetUploadAPI()

Description

Sets the upload API of type `WEBS_UPLOAD_API` to use.

Prototype

```
void IP_WEBS_SetUploadAPI(const WEBS_UPLOAD_API * pAPI);
```

Parameters

Parameter	Description
`pAPI`	Pointer to upload API.

9.65 IP_UTIL_BASE64_Decode()

Description

Performs BASE-64 decoding according to RFC3548.

Prototype

```
int IP_UTIL_BASE64_Decode(const U8  * pSrc,
                                int   SrcLen,
                                U8  * pDest,
                                int * pDestLen);
```

Parameters

Parameter	Description
pSrc	Pointer to the data to encode.
SrcLen	Number of bytes to encode.
pDest	Pointer to the destination buffer.
pDestLen	Pointer to destination buffer size, receives the number of bytes used in destination buffer.

Return value

> 0 Number of source bytes encoded, further call required.
= 0 All bytes encoded.
< 0 Error.

Additional information

For more information, refer to http://tools.ietf.org/html/rfc3548 and http://tools.ietf.org/html/rfc2440 .

9.66 IP_UTIL_BASE64_Encode()

Description

Performs BASE-64 encoding according to RFC3548.

Prototype

```
int IP_UTIL_BASE64_Encode(const U8  * pSrc,
                                int   SrcLen,
                                U8  * pDest,
                                int * pDestLen);
```

Parameters

Parameter	Description
pSrc	Pointer to the data to encode.
SrcLen	Number of bytes to encode.
pDest	Pointer to the destination buffer.
pDestLen	Pointer to destination buffer size, receives the number of bytes used in destination buffer.

Return value

> 0	Number of source bytes encoded, further call required.
= 0	All bytes encoded.
< 0	Error.

Additional information

For more information, refer to http://tools.ietf.org/html/rfc3548 and http://tools.ietf.org/html/rfc2440.

9.67 IP_UTIL_BASE64_EncodeChunk()

Description

Performs chunked BASE-64 encoding according to RFC3548.

Prototype

```
int IP_UTIL_BASE64_EncodeChunk(      IP_UTIL_BASE64_CONTEXT * pContext,
                               const U8                     * pSrc,
                                     int                      SrcLen,
                                     U8                     * pDest,
                                     int                    * pDestLen,
                                     char                     IsLastChunk);
```

Parameters

Parameter	Description
pContext	Context holding continue information between chunks. Needs to be initialized with zero by application.
pSrc	Pointer to the data to encode.
SrcLen	Number of bytes to encode.
pDest	Pointer to the destination buffer.
pDestLen	Pointer to destination buffer size, receives the number of bytes used in destination buffer.
IsLastChunk	Is this the last chunk of the data to encode ? • = 0: Further chunks will follow, carry over less than 3 bytes from the input into the next chunk that will be encoded. • ≠ 0: Last chunk, store all bytes left even if are not a multiple of 3.

Return value

> 0	Number of source bytes encoded, further call required.
= 0	All bytes encoded.
< 0	Error.

Additional information

For more information, refer to http://tools.ietf.org/html/rfc3548 and http://tools.ietf.org/html/rfc2440.

Chapter 10

Data structures

10.1 Overview

Structure / Callback	Description
WEBS_CGI	Used to store the CGI command names and the pointer to the proper callback functions.
WEBS_ACCESS_CONTROL	Used to store information for the HTTP Basic/Digest Authentication scheme.
WEBS_APPLICATION	Used to store application-specific parameters.
IP_WEBS_FILE_INFO	Used to store file-specific parameters.
WEBS_VFILE_APPLICATION	Used to check if the application can provide content for a simple VFile.
WEBS_VFILE_HOOK	Used to send application generated content from the application upon request of a specific file name.
WEBS_FILE_TYPE	Used to extend or overwrite the file extension to MIME type correlation.
WEBS_FILE_TYPE_HOOK	Used to extend or overwrite the file extension to MIME type correlation.
WEBS_METHOD_HOOK	Used to extend the usage of METHODs in the Web server for a given URI.
WEBS_PRE_CONTENT_OUTPUT_HOOK	Used to intercept the Web server before content is generated and sent.
WEBS_PROGRESS_HOOK	Used to get notified of the current progression of the requests served.
WEBS_PROGRESS_INFO	Used to pass information to a callback registered to be notified about the progression.
WEBS_REQUEST_NOTIFY_HOOK	Used to get notified of incoming requests to the Web server.
WEBS_REQUEST_NOTIFY_INFO	Used to pass information to a callback registered to be notified upon a request to the Web server.
WEBS_AUTH_DIGEST_APP_API	Used to interact with the application when Digest Authentication is used.
IP_WEBS_WEBSOCKET_API	Used for interaction between webserver and a WebSocket implementation.
IP_WEBS_pfMethod	Used to extend the usage of METHODs in the Web server for a given URI.
IP_WEBS_pfPreContentOutput	Called before content is generated and sent by the Web server in regular cases like VFiles and files from a filesystem.
IP_WEBS_pfRequestNotify	Called upon a request to the Web server.

10.2 Structure WEBS_CGI

Description

Used to store the CGI command names and the pointer to the proper callback functions.

Prototype

```
typedef struct {
  const char * sName;
  void  (*pf)(WEBS_OUTPUT * pOutput, const char * sParameters);
} WEBS_CGI;
```

Member	Description
sName	Name of the CGI command.
pf	Pointer to a callback function.

Additional information

Refer to *Common Gateway Interface (CGI)* on page 30 for detailed information about the use of this structure.

10.3 Structure WEBS_ACCESS_CONTROL

Description

Used to store information for the HTTP Basic/Digest Authentication scheme.

Prototype

```
typedef struct {
  const char* sPath;
  const char* sRealm;
  const char* sUserPass;
  const char* sCredentialsHash;
} WEBS_ACCESS_CONTROL;
```

Member	Description
sPath	A string which defines the path of the resources.
sRealm	A string which defines the realm which requires authentication. Optional, can be NULL.
sUserPass	A string containing the user name[/password] combination. Basic auth: "User:Pass". Digest auth: "User". Optional, can be NULL.
sCredentialsHash	A string containing the Digest credentials hash. Basic auth: Not used. Digest auth: MD5("User:Realm:Pass"). Optional, can be NULL.

Additional information

If sRealm is initialized with NULL, sUserPass is not interpreted by the Web server. Refer to *Basic Authentication* on page 58 for detailed information about the HTTP Basic Authentication scheme and *Digest Authentication* on page 61 about the HTTP Digest Authentication scheme.

10.4 Structure WEBS_APPLICATION

Description

Used to store application-specific parameters.

Prototype

```
typedef struct {
  const WEBS_CGI* paCGI;
  WEBS_ACCESS_CONTROL* paAccess;
  void  (*pfHandleParameter)(       WEBS_OUTPUT* pOutput,
                             const char         sPara,
                             const char*        sValue );
  const WEBS_VFILES* paVFiles;
} WEBS_APPLICATION;
```

Member	Description
paCGI	Pointer to an array of structures of type WEBS_CGI.
paAccess	Pointer to an array of structures of type WEBS_ACCESS_CONTROL.
pfHandleParameter	Fallback callback in case paCGI is NULL.
paVFiles	Pointer to an array of structures of type WEBS_VFILES.

10.5 Structure IP_WEBS_FILE_INFO

Description

Used to store file-specific parameters.

Prototype

```
typedef struct {
const IP_FS_API* pFS_API;  // Allows override of FS API to use for a file.
      U32 DateLastMod;     // Used for "Last modified" header field
      U32 DateExp;         // Used for "Expires" header field
      U8  IsVirtual;
      U8  AllowDynContent;
} IP_WEBS_FILE_INFO;
```

Member	Description
pFS_API	File system API override.
DateLastMod	The date when the file has been last modified.
DateExp	The date of the expiration of the validity.
IsVirtual	Flag to indicate if a file is virtual or not. Valid values are 0 for non-virtual, 1 for virtual files.
AllowDynContent	Flag to indicate if a file should be parsed for dynamic content or not. 0 means that the file should not be parsed for dynamic content, 1 means that the file should be parsed for dynamic content.

10.6 Structure WEBS_VFILE_APPLICATION

Description

Used to check if the application can provide content for a simple VFile.

Prototype

```
typedef struct WEBS_VFILE_APPLICATION {
  int  (*pfCheckVFile)(const char* sFileName, unsigned* pIndex);
  void (*pfSendVFile) (void* pContextIn,
                       unsigned Index,
                       const char* sFileName,
                       void (*pf) (void* pContextOut,
                                   const char* pData,
                                   unsigned NumBytes));
} WEBS_VFILE_APPLICATION;
```

Member	Description
pfCheckVFile	Pointer to a callback for checking if content for a requested file name can be served.
pfSendVFile	Pointer to a callback for actually sending the content for the requested file name using the provided callback pf. In case NumBytes is passed ’0’ the callback expects to find a string and will automatically run strlen() to find out the length of the string internally. In case NumBytes is not passed ’>0’ only NumBytes from the start of pData will be sent.

10.7 Structure WEBS_VFILE_HOOK

Description

Used to send application generated content from the application upon request of a specific file name.

Prototype

```
typedef struct WEBS_VFILE_HOOK {
  struct WEBS_VFILE_HOOK*         pNext;
         WEBS_VFILE_APPLICATION* pVFileApp;
} WEBS_VFILE_HOOK;
```

Member	Description
pNext	Pointer to the previously registered element of WEBS_VFILE_HOOK.
pVFileApp	Pointer to an element of type WEBS_VFILE_APPLICATION.

Additional information

Refer to *Structure WEBS_VFILE_HOOK* on page 161 for detailed information about the structure WEBS_VFILE_HOOK. Refer to *Structure WEBS_VFILE_APPLICATION* on page 160 for detailed information about the structure WEBS_VFILE_APPLICATION.

10.8 Structure WEBS_FILE_TYPE

Description

Used to extend or overwrite the file extension to MIME type correlation.

Prototype

```
typedef struct WEBS_FILE_TYPE {
  const char* sExt;
  const char* sContent;
} WEBS_FILE_TYPE;
```

Member	Description
sExt	String containing the extension without leading dot.
sContent	String containing the MIME type associated to the extension.

10.9 Structure WEBS_FILE_TYPE_HOOK

Description

Used to extend or overwrite the file extension to MIME type correlation.

Prototype

```
typedef struct WEBS_FILE_TYPE_HOOK {
  struct WEBS_FILE_TYPE_HOOK*  pNext;
         WEBS_FILE_TYPE        FileType;
} WEBS_FILE_TYPE_HOOK;
```

Member	Description
pNext	Pointer to the previously registered element of WEBS_FILE_TYPE_HOOK.
FileType	Element of Structure WEBS_FILE_TYPE.

Additional information

Refer to *Structure WEBS_FILE_TYPE* on page 162 for detailed information about the structure WEBS_FILE_TYPE.

10.10 Structure WEBS_METHOD_HOOK

Description

Used to extend the usage of METHODs in the Web server for a given URI.

Prototype

```
typedef struct WEBS_METHOD_HOOK {
  struct WEBS_METHOD_HOOK* pNext;
         IP_WEBS_pfMethod  pf;
  const  char*             sURI;
} WEBS_FILE_TYPE_HOOK;
```

Member	Description
pNext	Pointer to the previously registered element of WEBS_METHOD_HOOK.
pf	Pointer to callback handling the requested method of type Callback IP_WEBS_pfMethod.
sURI	URI registered for METHODs callback.

10.11 Structure WEBS_PRE_CONTENT_OUTPUT_HOOK

Description

Used to intercept the Web server before content is generated and sent.

Prototype

```
typedef struct WEBS_PRE_CONTENT_OUTPUT_HOOK {
  struct WEBS_PRE_CONTENT_OUTPUT_HOOK* pNext;
         IP_WEBS_pfPreContentOutput    pf;
} WEBS_PRE_CONTENT_OUTPUT_HOOK;
```

Member	Description
pNext	Pointer to the previously registered element of WEBS_PRE_CONTENT_OUTPUT_HOOK.
pf	Pointer to callback of type Callback IP_WEBS_pfPreContentOutput.

10.12 Structure WEBS_PROGRESS_HOOK

Description

Used to get notified of the current progression of the requests served.

Prototype

```
typedef struct WEBS_PROGRESS_HOOK {
  struct WEBS_PROGRESS_HOOK* pNext;
         IP_WEBS_pfProgress  pf;
} WEBS_PROGRESS_HOOK;
```

Member	Description
pNext	Pointer to the previously registered element of WEBS_PROGRESS_HOOK.
pf	Pointer to callback handling the requested method of type Callback IP_WEBS_pfProgress.

10.13 Structure WEBS_PROGRESS_INFO

Description

Used to pass information to a callback registered to be notified about the progression.

Prototype

```
typedef struct WEBS_PROGRESS_INFO {
  WEBS_OUTPUT* pOutput;
  U8           Status;
} WEBS_PROGRESS_INFO;
```

Member	Description
pOutput	Connection context.
Status	Current progression status: - WEBS_PROGRESS_STATUS_BEGIN - WEBS_PROGRESS_STATUS_METHOD_URI_VER_PARSED - WEBS_PROGRESS_STATUS_HEADER_PARSED - WEBS_PROGRESS_STATUS_END

10.14 Structure WEBS_REQUEST_NOTIFY_HOOK

Description

Used to get notified of incoming requests to the Web server.

Prototype

```
typedef struct WEBS_REQUEST_NOTIFY_HOOK {
  struct WEBS_REQUEST_NOTIFY_HOOK* pNext;
         IP_WEBS_pfRequestNotify   pf;
} WEBS_REQUEST_NOTIFY_HOOK;
```

Member	Description
pNext	Pointer to the previously registered element of WEBS_REQUEST_NOTIFY_HOOK.
pf	Pointer to callback handling the requested method of type Callback IP_WEBS_pfRequestNotify.

10.15 Structure WEBS_REQUEST_NOTIFY_INFO

Description

Used to pass information to a callback registered to be notified upon a request to the Web server.

Prototype

```
typedef struct WEBS_REQUEST_NOTIFY_INFO {
  const char*        sUri;
        WEBS_OUTPUT* pOutput;
        U8           Method;
} WEBS_REQUEST_NOTIFY_INFO;
```

Member	Description
sUri	Pointer to string containing the requested location.
pOutput	Connection context.
Method	HTTP METHOD used in the request: - METHOD_GET - METHOD_HEAD - METHOD_POST

10.16 Structure WEBS_AUTH_DIGEST_APP_API

Description

Used to interact with the application when Digest Authentication is used.

Prototype

```
typedef struct {
  void (*pfStoreNonce)(WEBS_OUTPUT* pOutput,
                       WEBS_AUTH_DIGEST_OUTPUT* pDigestOutput,
                       void (*pfStore)(
                               WEBS_OUTPUT*             pOutput,
                               WEBS_AUTH_DIGEST_OUTPUT* pDigestOutput,
                         const char*                    pNonce,
                               unsigned                 NonceLen),
                       int GenerateNew);
} WEBS_AUTH_DIGEST_APP_API;
```

Member	Description
pOutput	Connection context.
pDigestOutput	Digest Authentication context.
pfStore	Callback executed from the pfStoreNonce callback to store a nonce known by the application.
- pOutput	Connection context.
- pDigestOutput	Digest Authentication context.
- pNonce	Pointer to buffer containing the nonce to store.
- NonceLen	Length of nonce to store without string termination.
GenerateNew	= 0: Use old nonce or lookup nonce in an user implemented nonce cache. Nonce timeouts can be implemented by simply generating anew nonce once a timeout is reached. ≠ 0: Force generating a new nonce for a 401 response to send to the client.

10.17 Structure IP_WEBS_WEBSOCKET_API

Description

Used for interaction between webserver and a WebSocket implementation.

Prototype

```
typedef struct {
  int (*pfGenerateAcceptKey)( WEBS_OUTPUT* pOutput,
                              void*        pSecWebSocketKey,
                              int          SecWebSocketKeyLen,
                              void*        pBuffer,
                              int          BufferSize );
  void (*pfDispatchConnection)( WEBS_OUTPUT* pOutput,
                                void*        pConnection );
} IP_WEBS_WEBSOCKET_API;
```

Member	Description
pfGenerateAcceptKey	Callback used to generate the accept key that needs to be sent back for a WebSocket connection to be established. IP_WEBSOCKET_GenerateAcceptKey() can be used by the callback. Input/output buffer is the same. If this is not supported by your calculation callback the input buffer needs to be saved by your callback.
- pOutput	Webserver connection context.
- pSecWebSocketKey	Location of the received WebSocket key.
- SecWebSocketKeyLen	Length of the received key.
- pBuffer	Output buffer for the generated response key.
- BufferSize	Size of the output buffer for the response key.
pfDispatchConnection	Callback used to dispatch the connection handle from the Web server to another resource. Once the callback returns the transport connection (e.g. socket handle) no longer belongs to the webserver.
- pOutput	Webserver connection context.
- pConnection	Connection handle of the transport medium (e.g. socket handle).

10.18 Callback IP_WEBS_pfMethod

Description

Used to extend the usage of METHODs in the Web server for a given URI.

Prototype

```
typedef int (*IP_WEBS_pfMethod) (       void*        pContext,
                                        WEBS_OUTPUT* pOutput,
                                  const char*        sMethod,
                                  const char*        sAccept,
                                  const char*        sContentType,
                                  const char*        sResource,
                                        U32          ContentLen );
```

Member	Description
pContext	METHOD context for incoming data used with IP_WEBS_METHOD_* routines.
pOutput	Output context for IP_WEBS_* routines.
sMethod	String containing METHOD requested by client.
sAccept	String containing value of "Accept" field of header sent by client. May be NULL in case there was no such field.
sContentType	String containing value of "Content-Type" field of header sent by client. May be NULL in case there was no such field.
sResource	String containing URI that was accessed.
ContentLen	Length of data submitted by client that can be read. 0 in case no data was sent by client.

Note

Strings located at sAccept and sContentType need to be copied to another location before calling any other Web Server API as they will be overwritten.

10.19 Callback IP_WEBS_pfPreContentOutput

Description

Called before content is generated and sent by the Web server in regular cases like VFiles and files from a filesystem.

Prototype

```
typedef void (*IP_WEBS_pfPreContentOutput)( WEBS_OUTPUT* pOutput );
```

Member	Description
pOutput	Output context for IP_WEBS_* routines.

10.20 Callback IP_WEBS_pfRequestNotify

Description

Called upon a request to the Web server.

Prototype

```
typedef void (*IP_WEBS_pfRequestNotify) ( WEBS_REQUEST_NOTIFY_INFO* pInfo );
```

Member	Description
pInfo	Pointer to structure containing information about the current request being handled of type Structure WEBS_REQUEST_NOTIFY_INFO.

Note

Data located in pInfo needs to be copied to another location if they shall be used outside of the callback as they will be overwritten.

