InterNiche's embHTTP User's Guide for PIC32MX/MZ (MPLABX Tools)

51 E. Campbell Ave
Suite 160
Campbell, CA. 95008

InterNiche Technologies Inc.
email: Sales@iNiche.com
support: http://www.iniche.com/support

Revised: November 6, 2013

Table of Contents

Introduction

This Technical reference is provided with the InterNiche **embHTTP** library. The purpose of this document is to provide enough information so that a moderately experienced "C" programmer with a reasonable understanding of TCP/IP protocols can develop client and/or server applications using MPLABX development tools.

Note: Throughout this document, the term "web application" is used to refer to the server applications that the user develops for use over embHTTP.

The primary features of this library are:

- Small footprint
- Pre-ported to the FreeRTOS Operating System (source code included)
- "Device Locked" to PIC32MX/MZ **Important Note:** The software described in this document will not run on any component other than the PIC32MX/MZ. For support of another controller, contact InterNiche Sales: Sales@iNiche.com
- Sample Applications
- Menu System and Command Line Interface
- DEBUG and Non-DEBUG versions of the library are provided.

A Note About this Document

Unless specifically mentioned otherwise, the term **embTCP** is intended to apply to both the embTCP and embDUAL embedded library products.

Product Requirements

Installation

Before you start using this product, it is important that you have successfully built, downloaded and executed some small program using InterNiche's TCP/IP Library to your PIC32MX/MZ based board using MPLABX development tools. This is so that you have some end-to-end experience with your entire development environment and that you have confidence that your Ethernet and TCP/IP stack works.

Product Registration

As provided, embHTTP contains license information that will only allow it to operate for a finite period of time before halting. Registration is accomplished by visiting www.TCPIPStack.com, submitting a simple form and checking your email for a `http_license.o` file that should be used instead of `http_unregistered.o`

Project Integration

1. Begin with a working embTCP or embDUAL project ('debug' mode)
2. Unzip the package in to the same directory that contains the embsrc, emblibs and emb_h directories
3. Add embHTTP to your project:
    1. Edit `tcpdata.c` file and make the following modifications:
        - Add the following lines beneath the other `net_module` externs near the top of the file:
          `extern struct net_module http_module;`
          `extern struct net_module wbs_module;`
        - Find the in_modules array and add the following lines:
          `&http_module,`
          `&wbs_module,`
        - Verify that the file "`inmain.c`" contains a call to "`example_init()`" between the call to `nichestack_init()` and `TK_START_OS()`.
    2. Add `emblibs/libembHTTP-debug.a` to your project
    3. Add the file `httpdata.c` to your existing project in the same directory as `tcpdata.c`. For example1, the variables in httpdata.c should not be changed from their original default values, e.g,

```
char *webapp_root_path = "/";
char *dft_webpage = "index.iws";
```

    4. Include the file `http_license.o` ** into your existing project

** NOTE: If you have not yet registered your product, use `http_unregistered.o`. Registration will enable full use of the library and is accomplished by visiting www.TCPIPStack.com.

This will create an operating system task for embHTTP and will integrate it with the protocol stack.

An Important Note Regarding Stack Sizes

It is important to recognize that the task stack size requirements must be set appropriately for the unique requirements of your application and the requirements of your final product. Failure to properly tune the stacks will result in either wasted memory or nearly impossible to diagnose runtime errors.

The size of embHTTP's task stacks are specified in the `httpdata.c` file. Please refer to FreeRTOS.org for information regarding stack sizing and the debugging of stack overflow conditions.

Example1.c

`Example1` is a very simple application the uses embHTTP. It is designed to show that embHTTP is functioning and that a browser can connect to it and display a web page. Once this is done, it should be removed.

`Example1` assumes that it is linked with a DEBUG version of `embTCP` and that the system has a console.

In order to avoid any dependence on system/platform architecture, `Example1` carries its own in-memory file system and populates it with the web application files used by `Example1`.

Installing the Example1 Demo Application

1. Add the files from the `http_examples/example1` directory to your project
2. Compile and link `example1` and `embHTTP-debug.a` with an embTCP application, compiled as "DEBUG".
3. Start your application. Once the system begins to execute, it will display on your configured output device a message similar to the following:

```
embHTTP - Licensed for CHIPNAME. 0000-0000-0000-0000
Licensed to: NAME, user@example.com, COMPANYNAME
For PRODUCTNAME on CHIPNAME
```

4. From another system open a browser and browse to the IP address shown in the initialization message on the console. The browser should display a webpage that says:

```
"Congratulations. You have connected to embHTTP Webserver"
```

   It will also display a graphic and other information
5. Once you have successfully received the webpage, remove example1.c from your project.

Sample Application Walkthrough

Example1

`Example1.c` starts with two arrays. The first, "`unsigned char banner_02gif[ ]`", consists of a hexadecimal representation of each byte in banner_02.gif. The second, "char *indexiws", consists of a single character string representation of the index.iws file.

The main function of example1 is the `htmlinit()` routine which is always called at the end of embHTTP's initialization. This routine is unique to each web application and so **must be provided by you** (see the htmlinit section in this document for more information). For `example1`, the function simply calls `vfmakefile` to create the `banner_02.gif` and index.iws files in memory, and then it returns.

Once this initialization has completed, then when a browser connects to the IP address of the system where embHTTP is running, the browser will display the web page provided by index.iws.

Example2

`Example2` was designed as an example for the use of many of the basic capabilities of embHTTP. Example2 depends on the existence of a working file system. It makes calls to the basic file system APIs (vfopen(), vfread(), etc.), and you must have working implementations of these API function before example2 can be used.

Example2 provides examples of:

- Integrating JavaScript and the Ajax (the XMLHttpRequest object) with HTML
- Processing and displaying input from forms
- File upload
- Calling a CLI menu command over HTTP

As was done for the first example, `example2` carries its own in-memory file system in order to avoid any dependence on your system/platform architecture. Each file is represented by an array in `filestrings.c`. This method will not be needed for your web application unless your platform also uses an in-memory file system.

The subdirectory "`example_webfiles`" contains the original files that were used to create this demo application.

Debug vs Non-Debug Libraries

embHTTP includes two versions of the library: Debug, which is intended for use during initial application development; and Non-Debug, which is appropriate for use in your final product.

Overview

embHTTP processes requests from a web browser. It calls the appropriate internal routines and web applications to perform the requested functions or deliver the requested pages.

The embHTTP server handles two request types, `GET` and `POST`. There are many similarities between the two. With embHTTP, everything that can be done with a `GET` could be done with a `POST`. However, the reverse is not true; there are many tasks that can be done with a `POST` that cannot be done with a `GET`. A `GET` is designed for simple requests that take only a few simple parameters. A `POST` is normally used to send the name/value pairs entered into a form and should be used to send requests

with more than a simple set of parameters.

The browser sends requests to:

1. Obtain a file (Normally done with `GET`)
2. Request that one or more functions be performed on the server (`GET` or `POST` depending on whether form data is sent)
3. Upload a file to the server (`POST` only)

Embedded include files and commands

If the browser requests a simple file, embHTTP will generate an HTTP response header containing the length of the file that will be returned. It will then read the file and return it to the browser by writing to the socket used for the Internet. There will be no change on the server.

In embHTTP, a Web application file with the extension "`.iws`" or "`.iwx`" is called a "script" file. When a script file is requested, the Webserver will parse the file looking for escape sequences that contain embedded commands or that list one or more other files to include. An escape sequence within a file is always begins with the string, "`<#`", and ends with the string, "`#>`".

When embHTTP encounters this escape sequence, it will:

1. Send the file data prior to the escape sequence,
2. Parse out the include file or embedded command, including any parameters,
3. Execute the embedded command or read the include file,
4. Send any data output as a result of the command or include file,
5. Continue reading and outputting the file named in the request until it finds another escape sequence or the end of the file.

If the file extension is "`.iws`", embHTTP will set the file type parameter in the response header to "`text/html`". If the file extension is "`.iwx`", it will set the file type parameter to "text/xml".

A single file may contain any number of embedded commands or include files, and each include file (assuming it has an "`.iws`" or "`.iwx`" extension) could also contain embedded commands and include files. The nesting of include files is only limited by configuration parameters or available memory.

In order to include one file within another, the included file name must be within an escape sequence and the file name must be preceded by the word "include" and followed by a semicolon. For example: `<# include footer.htm; #>`. Note that there is only white space (space, horizontal tab, new-line, vertical tab, and form-feed) between the word "include" and the file name. There are no quotes in the escape sequence.

As an example of how this might be used, a set of Web pages could all include, at the appropriate spots, a .gif file to display the company logo, a file containing HTML to display the page header, and another file with HTML for the page footer.

A single escape sequence could contain one or more commands and/or include files. Each command or include file must be followed by a semicolon. The semicolon separates multiple commands, but it must always be used, even if there is only one command or include file in the escape sequence.

An embedded command must be either a:

1. a menu command known to the menu system,
2. a built-in command ("`include`" or "`echo`"),
3. the name of a user defined CGI routine registered with `reg_cgi_func()` (see Registering a CGI Function)
4. a script-a set of one or more commands, include files, or names of CGI routines within a single escape sequence.

Commands can be embedded within a file simply by putting the command within an escape sequence at the point within the file where you want the output from the command to be displayed. For example:

```
<# cticks; #>.
```

Note again that the command is always followed by a semicolon with no quotes around the command name.

An embedded command can include parameters. A command begins with the first non-white space character either after a begin-escape sequence or after the last semicolon within an escape sequence. A command ends with a semicolon. embHTTP will call the appropriate routine to handle the command, passing the entire string from the beginning of the command to the last character before the semicolon.

White space characters are ignored (space, horizontal tab, new-line, vertical tab, and form-feed) are used to separate the elements within an escape sequence. For readability, any number of white space characters may be used anywhere within the escape sequence. For example:

```
<#
currtime;
mycommand  -x -f filename -n nvalue;
include myfile.htm;
#>
```

Built-in Commands

Currently there are two build-in commands: "include" and "echo".

Include: The "include" command is simply the "include" part of

```
<# include filename; #>
```

Include: The "include" command is treated almost exactly like any other command. The difference is that it is implemented within embHTTP.

Echo: The "echo" command is used to send text to the browser. The text to be sent is delineated by either single or double quotation marks (whichever is found first following the echo command.) The text within the quotation marks can contain any characters or escape sequences acceptable to the browser, except quotation marks. In particular, the text can contain semicolons. This permits the use of special HTML escape sequences such as " ", which is used to code a space. For example:

```
<# echo "The current time is "; currtime; echo " GMT";  #>
```

The echo routine simple forwards all characters between the beginning quotation mark and the ending quotation mark. In the above example, the browser will display something like:

```
The current time is 10:14:11 GMT.
```

CLI

Menu Commands

embTCP includes a Command Line Interface (CLI) which is common to all InterNiche Embedded Libraries and provides access to its menu system. During initialization, most InterNiche modules use the CLI interface to provide a set of commands (a Menu) that can be used to control features in that module or to obtain information about the module. Each module's Menu becomes a submenu with the InterNiche menu system. Each menu entry represents a routine that will be executed when the entry is selected. (See the embTCP Reference Manual for more information).

Assuming the proper permissions, commands can be executed locally from the command prompt or remotely via telnet.

It is also possible for the porting engineer to set up specific menu commands for execution via the Internet. A simple way to do this would be to embed the command with an escape sequence in a script file. The script file could contain just enough HTML or XML surrounding the command to properly display its output as a web page.

---

embHTTP Menu Commands

embhttp htconfig - display or modify http server configuration

---

Name

`embhttp htconfig - display or modify http server configuration`

Syntax

`http htconfig [-d | -e] [-i <timeout>] [-n <timeout>] [-p <http_root_path>] [-r <bufferlength>] [-t <bufferlength>] [-w <webpage>]`

Parameters

(none)	Command without arguments displays the current state of http configuration parameters
-d	Disables Webserver. Terminates any existing connections and will not accept new connections.
-e	Enables the Webserver.

`-i`	Integer: Idle timeout in seconds for persistent connections
`-n`	Integer: Idle timeout in seconds for non-persistent connections
`-p`	Argument of type `string` sets the HTTP root path.
`-r`	Integer: HTTP's receive buffer size
`-t`	Integer: HTTP's transmit buffer size
`-w`	Argument of type `string` sets the default web page.

Description

This command displays or sets http configuration parameters

Notes/Status

- For the `-i` and `-n` options, an argument of '0' represents an infinite timeout.

http htnetstat - display HTTP Server statistics and status

Name

`http htnetstat - display HTTP Server statistics and status`

Syntax

`http htnetstat`

Parameters

This command takes no arguments

Description

This command displays status information for the embhttp module.

CGI Commands

The term CGI refers to the name of a routine or command that can be directly executed via a `GET` or `POST` request. As described above, both InterNiche provided Menu commands, as well as user provided commands, can be executed remotely via the Webserver by setting up script files (files that contain embedded commands and include files.) The CGI mechanism allows a `GET` or a `POST` to directly request the execution of a command. With the exception of the two built-in commands, "include" and "echo", the porting engineer must provide the routines that will enable the commands to be directly executed via the Webserver.

Like Menu commands, a CGI routine may also be executed from an escape sequence in a `.iws` or `.iwx` file. For example, a `mycgi.iws` file might contain:

```
<p>Some header text</p>
<# my_cgi_routine; #>
```

The advantage of this approach is that the webserver will build and send the required http headers in front of the output of "`my_cgi_routine`" and it will put the last chunk and trailer sections after the CGI output. Otherwise, the CGI routine itself would have to deal with the HTTP headers and trailers.

In the most common use of a CGI, the end-user on a browser first requests an HTML page from the server, which contains a form. The user fills in the form data and submits the form. The code that displayed the form also specifies the name of the routine that should be executed when the form is submitted. Normally, a `POST` request is used, and the form data is appended to the file name as a string of uuencoded text. It is also possible to call a CGI with a `GET` message. In this case, any parameters are attached to the request as a question mark separated list.

When the embHTTP receives a request for one of the web applications CGI functions, it parses all of the name/value pairs and stores them in the form structure and calls the web applications CGI routine. The web application can then use the various HTML library functions to retrieve this form information.

Web application developers can use this routine to do whatever they want with the passed data. The routine can change parameters on the server, write any desired HTML text directly over the socket, or it can ask embHTTP to return a file. This returned file could be a script file, which embHTTP will parse and handle just as if it had receive the file request from `GET` or a `POST`.

Authentication and Security

The HTTP 1.0 specification specified a user/password authentication scheme, which is generally referred to as "Basic" authentication. Basic authentication provides only minimal security. RCF 2617 specifies a much more secure authentication scheme that is generally called "Digest" authentication.

Access to each file and CGI command can be controlled separately, because each is associated with its own entry in the `ht_fileopts` file which contains a field indicating the authorization type required to access that file or command. (See Specifying Options for Web Application Files)

Basic Authentication

The advantages of Basic authentication are that it is simple, it uses few system resources, and it is supported by all popular web browsers. However, because the user name and password are sent in the clear, it is rarely used on publicly accessible web sites. It is still used on small, private, embedded systems or systems where only minimal security is sufficient

When a browser attempts access to a protected page, the browser asks them to enter a user name and password. The correct information must be entered before the server will provide the secure page.

Internally, several steps happen when a user first tries to access a protected page. First, the browser does a standard HTTP `GET` or `POST` request for the HTML page. The server notes that the page is protected and scans the HTTP header for an appropriate Authorization field. If this field is missing, as it usually is on the first request, the server sends an authorization failure error back to the browser. The browser then displays a popup window to the user asking for the name and password. The user types this in and the browser will encode the username/password into a sequence of base-64 characters (This process is called, "uuencoding"). The browser re-sends the HTTP `GET` request with the Authentication field (the uuencoded password info) appended to the HTTP request header. The server converts the uuencoded string back into plain text, checks the name and password, and, if it is OK, returns the protected HTML page.

It should be noted that uuencoding is NOT done to provide security. Passwords may contain special characters that could break the HTTP protocol. Uuencoding prevents this by substituting other characters in place of those that might cause problems. There is no secret involved in converting uuencoding back into plain text.

Digest Authentication

The embHTTP Server also provides a more sophisticated authorization scheme, Digest Authentication. Digest Authentication also uses the user/password scheme and most of what is described above for Basic Authentication also applies to Digest Authentication. The difference is that the username and password are now encrypted and several additional fields are added to improve security.

If a requested file is marked for Digest Authentication, embHTTP will send a response that includes the authentication options it supports and a special token, called the `nonce`. The client's response indicates the options it has selected. The response is encrypted in a manner that proves the client or system knows the password. If the server is satisfied with the response, it provides the requested file(s).

It should be noted that while Digest Authentication is secure enough for most real-world applications, it has vulnerabilities. It is not as good as state of the art public and private key encryption systems.

When a browser first requests an MD5 authorized file/page, the request will fail, but the browser will be told the type of authorization needed and given a nonce to include in subsequent requests for that file/page. Once fully authorized, the nonce is valid for that user for a server-determined period of time starting from when the nonce was created.

The nonce is a key factor in the strength of MD5. The nonce is a random number passed with the server's challenge. It must be encrypted and included with the other encrypted data returned in the browser's response to the challenge. The nonce ensures that the browser is responding to the specific challenge and is not simply repeating the response overheard from a previous response to a challenge. It is important that the nonce be highly random so that it cannot be predicted in advance.

The sample `get_nonce( )` routine supplied with the embHTTP delivery is contained within `httpdata.c` and creates the nonce directly from CTICKS. This method provides only minimum security, and it should be replaced with a stronger algorithm.

RFC 2617 suggests:

```
time-stamp H(time-stamp ":" ETag ":" private-key)
```

The nonce is saved in a table where each entry is a struct ht_svnonce.

```
struct ht_svnonce {
   char nonce[20];
   char name[FILENAMEMAX];
   int  nametype;
```

```
    u_long exptime;
    u_long lastused;
};
```

where:

nonce	Computed nonce value
name	User name or file name that is being authenticated
nametype	username or file name (MD5_USER or MD5_FILE)
exptime	Time when nonce will expire
lastused	Used to determine which entry in the table to reuse if no entries are available for a new request.

In the initial request, the browser may not know to send a user name, but because we return a nonce, we temporarily use the name of the requested file/page to identify for the nonce entry in the noncetab. When subsequently we get a request with this nonce and a user name, we substitute the user name into noncetab entry.

A noncetab entry is considered temporary until the user succeeds with digest authentication. At that point, the lastused field for the noncetab entry is set. The lastused field will be updated whenever the user accesses an MD5-authorized file/page.

Configuration

htmlinit function

Your web application must supply a function named `htmlinit()` with the following prototype:

```
int htmlinit(void);
```

As embHTTP finishes initialization, it will call `htmlinit()`. While the function must exist, embHTTP has no expectations for what it will do. The web application can do any initialization that it needs in this function.

Note: The return value from `htmlinit()` is critical. If `htmlinit()` returns a negative value, it will be assumed that the web application failed to initialize and embHTTP and embTCP will halt. If htmlinit returns 0 or a positive value, it will be assumed what the web application was successfully initialized.

Tunable Parameters

The `httpdata.c` file contains a set of parameters that may be tuned for the specific implementation. These are shown in the following table.

Parameter	Type	Default Value	Description
uint16_t	http_stksize	3072	Default size in bytes of the HTTP task's OS stack
uint16_t	wbs_stksize	4096	Default size in bytes of the Web Server task's OS stack
ht_enabled	int	TRUE	If TRUE, the HTTP server will be enabled at init time
webapp_root_path	char *	"/"	Path to web application files. May be an absolute path or relative to the executable's directory. The path must end with a forward slash
dft_webpage	char *	index.iws	The top-level web page that will be served if the user does not specify a specific file.
ht_txbufsize	int	2048 bytes	Size of HTTP transmit buffer for a connection. Range 400 to 16384 bytes. **NOTE**: When .iws and iwx files include other files, a single connection may use more than one transmit buffer
ht_rxbufsize	int	1024 bytes	Size of HTTP receive buffer for a connection. Range 700 to 16384 bytes. One per connection. **Note: The entirety of a non-multipart POST request, including all form data, must fit into a buffer whose size is specified by this value.**
pers_idletmo	uint32_t	180 * TPS	Idle timeout for persistent connections
non_pers_idletimeout	uint32_t	120 * TPS	Idle timeout for non-persistent connections
per_cons	int	TRUE	Enables persistent connections

The parameters below cannot be changed dynamically.

Parameter	Type	Default Value	Description
ht_port	int	80	HTTP listening port. Normally this will remain as 80, the "well-known" port number for HTTP. This

			parameter cannot be changed via the menu system.
ht_fileopts	char *	./ht_fileopts	Full path to a text file that contains a list of web application files that require special options: authorization type or user-specified headers to add when serving the file. NOTE: The path to `ht_fileopts` is NOT relative to `webapp_root_path`.

With the exception of the `webapp_root_path` and `dft_webpage`, the default values are suitable for most implementations. The parameter values found in `httpdata.c` will be set at compile time. The values for the top set of parameters may also be changed at run time via the "`htconfig`" menu command. They should not be altered in any other manner.

Normally, run time changes to the parameter values should be made at init time by calling the htconfig command. If htconfig is called before any HTTP connections have occurred, there is no need to disable and re-enable HTTP. Values changed at run time will not be retained across a reboot. While the run-time changes should normally be made at init time, this is not strictly necessary-that is, the system will continue to function, but the changes could cause some confusion for the browsers or users that have open connections.

Specifying options for web application files

The text file specified by the "`ht_fileopts`" variable can be used to specify options for specific web application files. The table below describes the three file options:

Option	Parameter	Definition
-b	None	Require Basic Authentication for this page/file
-d	None	Require Digest (MD5) Authentication for this page/file
-u	index list	Each number in the list represents a 1-based index into the user_headers table. If more than one user header applies to the specified file, the values in the list must be separated by one or more spaces.

Basic and Digest authentication are described in the "Authentication and Security" section of this document. User headers are described in the "Adding additional headers" section below.

The path/filename for `ht_fileopts` is one of the tunable parameters described in the "Tunable Parameters" section above. Note:

- The path to `ht_fileopts` is not relative to the `ht_root_path`, and can be an absolute path or a path relative to the executable.
- For files listed within `ht_fileopts`, the paths are either absolute or relative to `http_root_path`.

`ht_fileopts` is read once during initialization. Any subsequent changes to `ht_fileopts` will have no effect until the system is rebooted.

Entries in `ht_fileopts` have the format:

```
path/filename options
```

where each filename is listed on a separate line. Files that do not require options should not be listed in `ht_fileopts`. The file line may include both an authentication option and a list of user headers. Options 'b' and 'd' are mutually exclusive.

The authentication option applies to all files that are served with the specified file. For example, if the top-level `index.iws` file includes various `gif` files, xml scripts, and directly included files the authentication for the page (including none) will apply to all of these. It will also apply to links from the page, unless those links are specified with a different authorization level in ht_fileopts. For example, if you have logged into a page that requires BASIC authentication, you can follow all links from that page without further authorization, unless the linked file requires Digest authentication.

The following is an example of entries in `ht_fileopts`:

```
example/formtest.iwx  -b
example/info.iws -d -u 1 3
```

These entries specify that access to `formtest.iwx` requires Basic Authentication, and access to info.iws requires Digest authentication. The first and third headers in the `user_headers` table will be included when `info.iws` is served.

File Types and HTTP Headers

Built-in filetype-header associations

embHTTP has a default set of associations between common file extensions and the message headers (MIME types) used to describe the format of the file. These are shown in the following table.

Message Header (MIME type)	Associated file extensions
text/html	htm html iws
image/gif	gif
image/jpeg	jpeg jpg jpe
image/png	png
text/plain	bat txt c h cpp
test/css	css
application/x-zip-compressed	zip
application/octet-stream	ico ext com
text/xml	xml iwx

User defined filetype-header associations

The table "`userftypes[ ]`" in `httpdata.c` can be used to associate additional file types with additional MIME headers. Each entry in usrftypes[ ] is USERFTYPE structure:

```
typedef struct userftype {
   char *fextension;    /* File extension */
   char *hdrstr;        /* MIME header string */
} USERFTYPE;
```

In the following example, the header "`application/xyz`" will be added at the end of the other HTTP headers for each file that ends with the ".xyz" extension, and the header "`application/abc`" will be added for each file that ends with the "`xxx`" extension.

```
USERFTYPE userftypes[] = {
   /* Insert additional file types here */
   { "xyz", "application/xyz" },
   { "xxx", "application/abc" },
   { NULL, NULL },  /*  MUST be last element of array. Do NOT remove */
};
```

Because `userftypes` defines a compilable array of C structures, the formatting must be exactly as shown. All of the brackets and commas are required. The `userftypes` array must exist. The last element must be:

```
   { NULL, NULL }
```

This is required whether or not there are any user defined file types.

Adding additional headers

It is possible to specify that additional headers should be added for specific files. These headers will be added in addition to those added based on the specific file extension. Specifying additional headers is a two-step process.

First additional headers can be defined in the "`user_headers`" array in `httpdata.c`. "`user_headers`" is simply an array of strings where each string is a header that the implementation may want to have added for specific files. The following example lists three headers:

```
   char *user_headers[] = {
  /* Insert  user headers here */
  "Access-Control-Allow-Origin: *\r\n",
```

```
  "Any header you want to add *\r\n",
  "One more header *\r\n",
   NULL,  /* MUST be last element of array. Do NOT remove */
};
```

NOTE: The `user_headers` array must exist. The last element must be `NULL` whether or not there are user headers.

Second, in order to cause one or more of these headers to be added each time a specific file is served, the file must be listed in the text file, "file_opts" An entry consists of the pathname, followed by the "-u" option. The parameter for the u option is a list of one or more indexes into the user_headers table. In the following example, the first and third headers from user_headers[ ] will be added when serving "../example/file.xyz:

```
example/file.xyz -u 1 3
```

API

---

embHTTP API

CGI Functions

The term "CGI Function" refers to a routine or command that can be directly executed via a GET or POST request. The InterNiche embedded products provide a rich set of build-in menu commands and functions that can be called via CGI functions. Most web applications will define additional CGI functions. The interface that calls CGI functions was designed assuming that most CGI functions will process the results of an HTML form. However CGI functions are not limited to this.

CGI Function Prototype

CGI functions have the following typedef:

```
typedef int (*HT_CGIFUNC)(char *requri,
                          int reqtype,
                          char **filetext,
                          void *si,
                          void *form,
                          void *ctx );
```

Where:

requri	Base URI sent by the browser, not including any parameters
reqtype	Request type or method (e.g., GET, POST)
filetext	Used by the application to return the name of a file whose contents should be displayed at this point by the browser
si	void pointer to be passed to routines that require it
form	void pointer to be passed to routines that require it
ctx	void pointer used with calling CLI commands and functions

The return from a CGI function should consist of one of the following values:

FP_ERR	Internal (code) error
FP_FILE	fileptr is valid
FP_DONE	CGI did everything, just clean up
EHT_BADREQ	Bad request from the browser

If the return is not one of the above values, or if it is FP_ERR, the Webserver will return an HTTP 500 (Server error) to the client browser.

If the application wants to send a text message in response to the request, it should format the response, send it via `embhttp_sendbuf`, and then return FP_DONE. This tells embHTTP to clean up the connection. The text message could either be a success response, or an error response. If the application returns EHT_BADREQ, then embHTTP will send an HTTP 400 (bad request) response to the browser with the text, "Form Parse Error".

One of the parameters in the call to the CGI function was the address of a pointer variable (char **fileptr). The application can set this to point to a file and return FP_FILE to embHTTP, which will then send the file as a response to the request. The file may be a simple file or a file containing embedded commands and/or include files.

When non-HTML data is sent in the middle of a data stream that contains HTML data, then substitutions must be made for certain characters in order to prevent the browser from interpreting them as HTML control characters. This is called "cooking" the output. The application should call `setcookedflag(si)` macro whenever the data should be cooked, it should call `clearcookedflag(si)` at the end of that data.

Registering a CGI Function

For each CGI function defined by the web application, the application must call `reg_cgifunc()` before the CGI can be invoked because of a browser request.

The following is the prototype for reg_cgi_func()

```
int reg_cgifunc(char *cmdname, HT_CGIFUNC cgifunc, int security);
```

cmdname	name of the command that will be passed in the request from the browser.
cgifunc	name of the HT_CGIFUNC that the Webserver will call for the request
security	= 0 (none) , HT_AUTHBASIC, or HT_AUTHDIGEST

This function returns either SUCCESS or an errcode (EINVAL or ENOMEM)

NOTE: The security parameter will be ignored if the CGI call is embedded in a webpage for which the user is authorized.

Registering C Variables

InterNiche embedded libraries menu commands provide access to the current value of a large number of system variables. The web application can display these variables via the following steps

1. Register the variable with the reg_cvars function below.
2. Creating a CGI routine that calls a menu command whose output contains the variable of interest.
3. Parsing the variable from the output
4. Pass the variable to the appropriate htmllib function for output on the web page.

The "`reg_cvars()`" function can be used by the web application to register C variables that the Webserver should serve when it receives a request for that variable. It has the following prototype:

```
int reg_cvars(char *webname, void *varaddr, int type)
```

webname	Name seen by the browser for the variable
varaddr	Address of the variable
type	Variable type

This function returns either ENOMEM or 0 for SUCCESS

The following table shows the variable types recognized by the Webserver and the htmllib function that will be called to display the variable (See "htmllib functions" section)

Variable Type Name	Meaning	htmllib function to be called
WBS_INTEGER	short/long/signed/unsigned value	wbs_putlong()
WBS_IP4ADDR	Display specified IP4 Address	wbs_putip4addr()
WBS_CTICKS	Display current CTICKS	wbs_putlong()

General Functions

Name

`setcookedflag()`

Syntax

```
void setcookedflag(void *si);
```

Parameters

si	passed to the application

Description

Sets the `HF_COOKOUTPUT` flag, which tells `embhttp_sendbuf` that it must encode characters to prevent them from being interpreted as special HTML characters.

Returns

Nothing

Name

`clearcookedflag()`

Syntax

```
void clearcookedflag(void *si);
```

Parameters

si	passed to the application

Description

Clears the `HF_COOKOUTPUT` flag.

Returns

Nothing

Name

`embhttp_sendbuf()`

Syntax

```
int embhttp_sendbuf(void *si, char *buf, int length);
```

Parameters

si	The si pointer passed to the CGI routine
buf	address of the data buffer to be sent
len	length in bytes of 'buf'

Description

This routine is called whenever data needs to be sent by the web application. If the `HF_COOKOUTPUT` flag is set, the data is first passed through a routine that encodes characters that might be misinterpreted by the browser. `embhttp_sendbuf` will block until length bytes have been sent or there is an error

If data chunking is being used in reply to a HTTP request, then `embhttp_sendbuf()` does the appropriate formatting. That is, it sends the length fields, followed by

CRLF, followed by data, followed by CRLF. If buff is NULL and length is 0, then it means that all data has been sent and `embhttp_sendbuf()` will send a Last-Chunk message (0 followed by CRLF CRL).

Returns

SOCKET_ERROR or bytes written. The extra bytes sent for chunking are not reported in the return value. That is, the maximum return value is equal to the "`length`" parameter.

Functions for Reading and Writing Form Fields

The following tables summarize the functions available to reading and write form fields. Each of these functions is described in more detail below.

Function Name	Purpose
General functions for reading form fields. Returns the value for the name/value pair matching the specified name and index.	
ht_get_form_str	Return pointer to string value
ht_get_form_int	Write integer value to address of value argument
ht_get_form_bool	Write the TRUE/FALSE value of CHECKBOX
ht_get_form_ip4addr	Write binary IP4 address to ipptr argument.
ht_get_form_ip6addr	Write a binary IPv6 address to the ip6adr argument
Simplified functions for reading form fields. Returns value for the first matching name	
get_form_str	Return pointer to string value
get_form_int	Write integer value to address of value argument
get_form_bool	Write the TRUE/FALSE value of CHECKBOX
get_form_ip4addr	Write binary IP4 address to ipptr argument.
get_form_ip6addr	Write a binary IPv6 address to the ip6adr argument
Functions for writing to a form. Writes specified string to the socket specified by the CTX argument	
wbs_putstring	Writes specified string
wbs_putlong	Formats long into an ASCII string before writing
wbs_putshort	Formats short into an ASCII string before writing
wbs_putip4addr	Formats binary IP4 address to ASCII string in dotted notation
wbs_putip6addr	Formats a binary IP6 address in an ASCII string in colon notation

Getting Information from Forms

There are two versions of each of the get_XXX functions in this API. The first versions are more general and will be listed first. These must be used when the form may contain multiple name/value pairs matching the specified name.

Name

`ht_get_form_str()`

Syntax

```
int ht_get_form_str(void *form, char *name, char *buf, int buflen, int index);
```

Parameters

form	The form pointer passed to the CGI routine
name	Name of name/value pair to search for
buf	Buffer large enough to hold the expected strings
buflen	length of passed buffer
index	If there are multiple matches to name, the index indicates which one to return

Description

Search a form for name/value pairs matching the specified name. If multiple matching name/value pairs are found, the index argument is used to determine which to return. The index argument is "1 based". If found, the null terminated string will be copied into the provided buffer. If the string is too long for the buffer, it will be truncated with a null-terminator at the end. If no matches are found, the contents of the buffer will be unchanged.

Returns

Number of name/value pairs matching the specified name.

Name

`ht_get_form_int()`

Syntax

```
int ht_get_form_int(void *form, char *name, uint32_t *value, int index);
```

Parameters

form	The form pointer passed to the CGI routine
name	Name of name/value pair to search for
value	Where to write the integer value of the form pair
index	If there are multiple matches to name, the index indicates which one to return

Description

Search a form for name/value pairs matching the specified name. Convert the value to an integer and write it to the address of the "value" argument. If multiple matching name/value pairs are found, the index argument (1-based) is used to determine which to return.

Returns

Number of name/value pairs matching the specified name.

Name

`ht_get_form_bool()`

Syntax

```
int ht_get_form_bool(void *form, char *name, int *value, int index);
```

Parameters

form	The form pointer passed to the CGI routine
name	Name of name/value pair to search for
value	Where to write the resulting TRUE/FALSE

index	If there are multiple matches to name, the index indicates which one to return

Description

Get the TRUE/FALSE state of an INPUT type= CHECKBOX in a form. Note that unchecked CHECKBOXes generally don't even appear in the form. If multiple matching name/value pairs are found, the index argument (1-based) is used to determine which to return. If the selected value is TRUE, then an integer one is written to the address of the "value" argument; if FALSE, a 0 value is written.

Returns

Number of name/value pairs matching the specified name.

Name

`ht_get_form_ip4addr()`

Syntax

```
int ht_get_form_ip4addr(void *form, char *name, ip_addr *ipptr, int index);
```

Parameters

form	The form pointer passed to the CGI routine
name	Name of name/value pair to search for
ipptr	Where to write the binary IP4 address value of the form pair
index	If there are multiple matches to name, the index indicates which one to return

Description

Search a form for name/value pairs matching the specified name. Convert the value to a binary IP4 address and write it to the address of the "ipptr" argument. If multiple matching name/value pairs are found, the index argument (1-based) is used to determine which to return.

Returns

Number of name/value pairs matching the specified name.

Name

`ht_get_form_ip6addr()`

Syntax

```
int ht_get_form_ip6addr(void *form, char *name, ip6_addr *ip6adr, int index);
```

Parameters

form	The form pointer passed to the CGI routine
name	Name of name/value pair to search for
ip6_addr *ip6adr	A pointer to an IP6 address structure where the binary IP6 address value of the form pair will be written
index	If there are multiple matches to name, the index indicates which one to return

Description

Search a form for name/value pairs matching the specified name. Convert the value to a binary IPv6 address and write it into the passed ip6_addr structure

Returns

Number of name/value pairs matching the specified name. If no matching name is found, an ipv6 address of all 0's is written into the passed buffer.

Simplified Functions for Retrieving Form Information

The following are simpler versions of the above APIs. They should only be used if it is known that only one name/value pair will match the specified name. The functions will return the value of the first matching name/value pair found in the form.

Name

`get_form_str()`

Syntax

```
char *get_form_str(void *form, char *name, char *buf, int buflen);
```

Parameters

form	The form pointer passed to the CGI routine
name	Name of name/value pair to search for
buf	Buffer large enough to hold the expected strings
buflen	length of passed buffer

Description

Search a form for the name/value pair matching the name passed. If found, the null terminated string will be copied into the provided buffer. If the string is too long for the buffer, it will be truncated with a null-terminator at the end. If no matches are found, the contents of the buffer will be unchanged.

Returns

-1 or the number of bytes copied into the buffer.

Name

`get_form_int ()`

Syntax

```
int get_form_int(void*form , char *name, uint32_t *value);
```

Parameters

form	The form pointer passed to the CGI routine
name	Name of name/value pair to search for
value	Where to write the integer value of the form pair

Description

Search a form for name/value pairs matching the specified name. Convert the value to an integer and write it to the address of the "value" argument

Returns

0 if OK, else -1 if there's a problem, in which case the passed variable is not altered.

Name

`get_form_bool ()`

Syntax

```
int get_form_bool(void *form, char *name);
```

Parameters

form	The form pointer passed to the CGI routine
name	Name of name/value pair to search for

Description

Get the TRUE/FALSE state of an INPUT type= CHECKBOX in a form. Note that unchecked CHECKBOXes generally don't even appear in the form.

Returns

TRUE if the named variable was found and set to TRUE; Otherwise it returns FALSE.

Name

`get_form_ip4addr ()`

Syntax

```
char *get_form_ip4addr(void *form, char *name, ip_addr *ipptr);
```

Parameters

form	The form pointer passed to the CGI routine
name	Name of name/value pair to search for
ipptr	Where to write the binary IP4 address value of the form pair

Description

Search a form for name/value pairs matching the specified name. Convert the value to a binary IP4 address and write it to the address of the "ipptr" argument

Returns

NULL if OK, else pointer to a string describing a problem, in which case the contents of ipptr are not altered.

Name

`get_form_ip6addr ()`

Syntax

```
char *get_form_ip6addr(void *form, char *name, ip6_addr *ip6adr);
```

Parameters

form	The form pointer passed to the CGI routine
name	Name of name/value pair to search for

ip6adr	A pointer to an IP6 address structure where the binary IP6 address value of the form pair will be written

Description

Search a form for name/value pairs matching the specified name. Convert the value to a binary IPv6 address and write it into the passed ip6_addr structure

Returns

NULL if OK, else a pointer to a string describing a problem, in which case an ipv6 address of all 0's is written into the passed buffer.

Writing to a Form

The routines below format a data type into ASCII for display on a web page and then send the formatted string to the socket passed.

Name

`wbs_putstring ()`

Syntax

```
int wbs_putstring(void *si, char *string);
```

Parameters

si	The si pointer passed to the CGI routine
string	The string to be sent over the socket

Description

Writes the specified string to the socket specified by "`si`"

Returns

-1 for a socket error; otherwise, the number data bytes written, not including the extra bytes sent for chunking. The maximum return value is equal to the length argument.

Name

`wbs_putlong()`

Syntax

```
int wbs_putlong(void *si, uint32_t lng);
```

Parameters

si	The si pointer passed to the CGI routine
lng	The integer to be sent over the socket

Description

Formats the specified long into an ASCII string for display on a web page and sends the formatted string to the browser.

Returns

-1 for a socket error; otherwise, the number data bytes written, not including the extra bytes sent for chunking.

Name

`wbs_short()`

Syntax

```
int wbs_putshort (void *si, uint16_t shrt);
```

Parameters

si	The si pointer passed to the CGI routine
shrt	The short integer to be sent over the socket

Description

Formats the specified short into an ASCII string for display on a web page and sends the formatted string to the browser.

Returns

-1 for a socket error; otherwise, the number data bytes written, not including the extra bytes sent for chunking.

Name

`wbs_putip4addr()`

Syntax

```
int wbs_putip4addr(void *si, ip_addr ipaddr);
```

Parameters

si	The si pointer passed to the CGI routine
ipaddr	The binary value of the IPv4 address to be sent over the socket

Description

Formats the passed binary IPv4 address into an ASCII string in the standard dotted notion for an IP4 address. It sends the string to the browser.

Returns

-1 for a socket error; otherwise, the number data bytes written, not including the extra bytes sent for chunking.

Name

`wbs_putip6addr()`

Syntax

```
int wbs_putip6addr(void *si, ip6_addr *ip6adr);
```

Parameters

si	The si pointer passed to the CGI routine
ip6adr	Pointer to a binary IPv6 address

Description

Formats the passed IPv6 binary address into an ASCII string in the standard colon notion for an IPv6 address. It sends the string to the browser.

Returns

-1 for a socket error; otherwise, the number data bytes written, not including the extra bytes sent for chunking.

The File System API

embHTTP requires a file system. It must have standard interfaces into the file system regardless of whether the file system is implemented on a disk, in flash, in memory, or by any other type of hardware. The header file "vf_file.h" gives the prototypes of the required functions used by embHTTP. For each of these functions, the system designer needs to provide a implementation appropriate for their specific platform. The file API functions are in "`httpdata.c`". Each function contains a location where specific implementation code should be added.

For many platforms, the implementation code could simply be a call to the appropriate C library function.

Note: The API functions "`vfmakefile()`" and "`vfputc()`" are not used internally by embHTTP. If you are sure your application will never need these two functions, then you do not have to add implementation code within those functions..

The following brief list of the file system API functions. Each will be described in more detail below:

API	Purpose
HT_VF_LOC	Lock file system
HT_VF_UNLOCK	Free file system lock
vfmakefile	Create a file in the file system
vfopen	Open a file
vfclose	Close a file
vfread	Read data from a file
vfwrite	Write data to a file
vfseek	Move to new position within a file
vftell	Obtain the current position within a file
vfgetc	Read one character from a file
vputc	Write one character to a file

Name

`HT_VF_LOCK`

Syntax

```
HT_VF_LOCK();
```

Parameters

None

Description

Write an implementation for this define if it is necessary to lock your file system before calling one or more of the file system APIs. The default version:

```
#define HT_VF_LOCK()
```

will compile to nothing.

Returns

Nothing

Name

`HT_VF_UNLOCK`

Syntax

```
HT_VF_UNLOCK();
```

Parameters

None.

Description

Write an implementation for this define, if it is necessary to lock your file system before calling one or more of the file system APIs. The default version:

```
#define HT_VF_UNLOCK()
```

will compile to nothing.

Returns

Nothing.

Name

`vfmakefile()`

Syntax

```
int vfmakefile(char *name, char *data, unsigned size, uint16_t flags);
```

Parameters

data	character array
size	character array size
flags	file flags

Description

Converts a character array into a VFS file

Returns

0 on SUCCESS or an error code

Name

`vfopen()`

Syntax

```
void *vfopen(char *filename, char *mode);
```

Parameters

filename	file name
mode	Mode in which file is to be opened (read, write, etc.)

Description

Opens the a file and returns a void file pointer, which should be used for subsequent file APIs for the file

Returns

NULL or a pointer that should be used for subsequent file APIs involving this file

---

Name

`vfclose()`

Syntax

```
void *vfclose(void *fptr);
```

Parameters

fptr	void pointer of an open file structure

Description

Closes a file and frees the file structure. The implementation should cast fptr to the type needed.

Returns

Nothing.

Name

`vfread()`

Syntax

```
int vfread(char *buf, unsigned size, unsigned count, void *fptr);
```

Parameters

buf	Buffer where the data will be stored
size	Size in bytes of each data item
count	Number of data items to be read
fptr	Pointer to open void structure

Description

Reads from the current position in the file, the specified number of data items, each of the given size, into the specified buffer. The implementation should cast fptr to the type needed.

Returns

Number of items read

Name

`vfwrite()`

Syntax

```
int vfwrite(char *buf, unsigned size, unsigned count, void *fptr);
```

Parameters

buf	Buffer containing data to be written
size	Size in bytes for each data item
count	Number of data items to be written
fptr	Pointer to open void structure

Description

Writes the specified number of data items, each of the given size, into the specified file starting at the current position in the file. The implementation should cast fptr to the type needed.

Returns

Number of items written

Name

`vfseek()`

Syntax

```
int vfseek(void *fptr, long offset, int origin);
```

Parameters

fptr	Pointer to open void structure
offset	Bytes from origin to place position indicator
origin	One of: SEEK_SET, SEEK_CUR, SEEK_END

Description

Move to a new position in an open file. The implementation should cast fptr to the type needed.

Returns

0 on success, or a non-zero error value

Name

`vftell()`

Syntax

```
long vftell(void *fptr);
```

Parameters

fptr	Pointer to open void structure

Description

Returns the current offset in bytes from the beginning of the file. The implementation should cast fptr to the type needed.

Returns

The current offset in the file or -1 if there was an error

Name

`vfgetc()`

Syntax

```
int vfgetc(void *fptr);
```

Parameters

fptr	Pointer to open void structure

Description

Reads a character from the current position of the file pointer. Otherwise it returns EOF and sets an error code. The caller can check the error code to see if there was an error or if it just reached the end of the file.

Returns

Returns the character read or EOF.

Name

`vfputc()`

Syntax

```
int vfputc(int ch, void *fptr);
```

Parameters

ch	The character to be written
fptr	Pointer to open void structure

Description

Writes the character given in the integer argument to the current location of the file pointer.

Returns

Returns the character written on success; otherwise, it returns EOF and sets an error code.

Application Developer Supplied Security Functions

`httpdata.c` contains two authentication functions that need to be implemented by the application developer. The existing implementations were designed only to allow embHTTP to function, and they should be replaced. See the Authentication and Security section of this manual for a more complete explanation.

Name

`ht_check_permit()`

Syntax

```
int ht_check_permit(char *username, char *password, int module, void *fname);
```

Parameters

username	user name supplied in the HTTP request
password	password supplied in the HTTP request
module	integer identifier of calling module. Always HTTP_USER for embHTTP
fname	file name for which permission is requested

Description

Application supplied function to grant or refuse access permission. This function is called for any request that references a file listed in file_opts with `a`, `b` or `d` option.

Returns

TRUE if access is granted; otherwise FALSE

Name

`get_nonce()`

Syntax

```
struct ht_svnonce *get_nonce(char *username, char *fname);
```

Parameters

username	user name supplied in the request
fname	file name for which permission is requested

Description

A nonce is a random value that is passed back with the authentication challenge. The nonce must be encoded as part of the browsers response. The method for computing the nonce value is a large factor in the strength of Digest Authentication. See the make_nonce section of Digest Authentication in the embHTTP User's Guide for a complete description

Returns

One of the entries in noncetab

Overview

Related Products

This product was derived from a portable, flexible and more full-featured product available from InterNiche Technologies, Inc. For more information about this **SOURCE CODE PRODUCT**, please visit www.iNiche.com or email Sales@iNiche.com.

For Additional Information ...

- InterNiche Support Site
- FreeRTOS web site