AN1107

An HTTP Server Using BSD Socket API

Author: Sean Justice
Microchip Technology Inc.

INTRODUCTION

An embedded HTTP (Hyper Text Transfer Protocol) server, or web server (as it is commonly called), is an excellent addition to any network-enabled device. HTTP server capability allows an embedded device to be monitored and controlled remotely using any standard, off-the-shelf Internet browser. Owing to the ubiquitous deployment of Internet browsers, a web-enabled device can be accessed from almost any computer – desktop or mobile.

This Microchip HTTP server application note and the included FAT16 module, supplemented by the TCP/IP application note AN1108, "*Microchip TCP/IP Stack with BSD Socket API*", provide an HTTP Server module that can be integrated with almost any application on a Microchip 32-bit microcontroller product.

The TCP/IP application note and the FAT16 module are required to compile and run the HTTP server module. All notes and files mentioned in this document are available for download from www.microchip.com.

The software in the installation files includes a sample application that demonstrates all of the features offered by this HTTP server module.

Questions and answers about the HTTP server module are provided at the end of this document in **"Answers to Common Questions"** on page 35.

ASSUMPTION

The author assumes that the reader is familiar with the following Microchip development tools: MPLAB® IDE and MPLAB® REAL ICE™ in-circuit emulator. It is also assumed that the reader is familiar with C programming language, as well as TCP/IP stack, FAT16 file system, and HTTP server concepts. Terminology from these technologies is used in this document, and only brief overviews of the concepts are provided. Advanced users are encouraged to read the associated specifications.

FEATURES

The HTTP server provided here does not implement all HTTP functionality; it is a minimal server targeted for embedded systems. The user can easily add new functionality as required.

The HTTP server presented here incorporates the following features:

- Provides portability across the 32-bit family of PIC® microcontrollers
- HTTP Server APIs compatible with PIC18/PIC24 Microchip HTTP Server APIs
- Supports multiple HTTP connections
- Automatic interaction with the FAT16 file system
- Supports the HTTP methods: `GET`, `HEAD`, `POST`, and `PUT`
- Supports "continue" response that may be requested by the client.
- Supports a modified Common Gateway Interface (CGI) to invoke predefined functions from within the remote browser
- Supports dynamic web page content generation
- Supports HTTP web page authentication

LIMITATIONS

While the HTTP server supports the passing variables and values using the `GET` and `POST` method, the size of the data is limited. The size of the variable and value are limited to `HTTP_VAR_LEN` and `HTTP_VALUE_LEN`, respectively, to allow for the variable/value pair to be contained in the receive buffer. Accommodation for this limitation should be made in the web page design. It is prudent to use the smallest possible variable and value length to save memory length allocations.

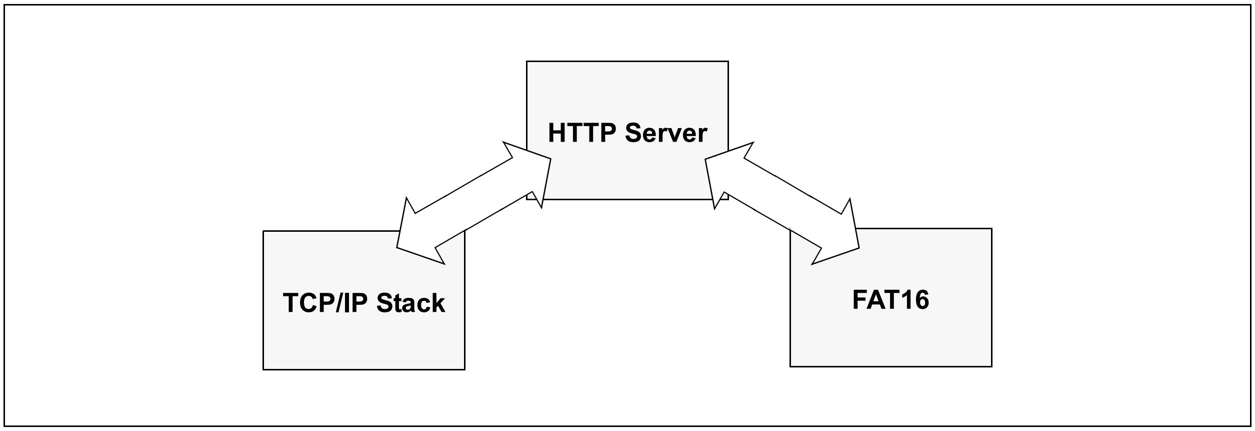
Because the server is running on an embedded system, large web pages with graphics could take longer to download to the web browser. The use of pages containing frames that are updated independently will allow for faster refreshing of data.

The HTTP server is designed to handle a finite amount of simultaneous connections. The number of connections that the server can handle depends on the amount of data that is able to be allocated on the memory heap. An analysis of a worst-case scenario can be used to insure that the size of the memory heap is sufficient to support the maximum number of simultaneous connections.

TYPICAL HARDWARE

The HTTP server demonstration application requires the use of AN1108, "*Microchip TCP/IP Stack with BSD Socket API*", and FAT16 hardware and software.

FIGURE 1: HTTP SERVER DEPENDENCIES



RESOURCE REQUIREMENTS

Program memory required by the HTTP server is stated in the following table.

TABLE 1: MEMORY REQUIREMENTS

Resource	Memory
`HTTP.c` module	5492 bytes
RAM memory required by the HTTP server	30 bytes

Note: Since the HTTP server requires the use of TCP/IP and FAT16, it will also inherit their program memory and RAM requirements.

The compiler used for the memory requirements was Microchip C32 version 1.00. The optimization was not used. Note that the use of compilers and optimization settings may increase or decrease the memory requirements.

TABLE 2: RAM MEMORY REQUIRED BY THE HTTP SERVER

Resource	Memory
Private data	6 bytes
Per connection data	44 bytes
Per passed parameter	32 bytes (user-defined)
HTTP file name	32 bytes (user-defined)
Receive buffer	80 bytes (user-defined)
Transfer buffer	80 bytes (user-defined)

Note: A typical application running 10 HTTP connections with the default size defines will use 2548 bytes of RAM.

INSTALLING SOURCE FILES

The complete source code for the Microchip HTTP Server is available for download from the Microchip web site (see **"Source Code for the HTTP Server"** on page 37).

The source code is distributed in a single Windows® installation file:
`pic32mx_bsd_tcp_ip_v1_00_00.exe`.

Perform the following steps to complete the installation:

1. Execute the file. A Windows installation wizard will guide you through the installation process.
2. Click **I Accept** to consent to the software license agreement.
3. After the installation process is completed, the **HTTP Server Using BSD Socket API** item is available under the **Microchip** program group.

   The complete source files are copied to the following directory, in your choice of installation path:
   `\pic32_solutions\microchip\`
   `bsd_tcp_ip\source\bsd_http_server`

   The "include" files are copied to the following directory:
   `\pic32_solutions\microchip\`
   `include\bsd_tcp_ip`

   The demonstration application for the BSD HTTP server is located in the following directory:
   `\pic32_solutions\`
   `bsd_http_server_dhcp_demo`
4. For the latest version-specific features and limitations, refer to the version HTML page, which can be accessed through `index.html` .

SOURCE FILE ORGANIZATION

The HTTP Server consists of multiple files. These files are organized in multiple directories.

Table 3 shows the directory structure.

Table 4 lists the server-related source files.

TABLE 3: SOURCE FILE DIRECTORY STRUCTURE

Directory	Description
pic32_solutions\bsd_http_server_dhcp_demo	HTTP server demo project files and source files
pic32_solutions\microchip\bsd_tcp_ip\source\bsd_http_server	HTTP server source files
pic32_solutions\microchip\include\bsd_tcp_ip	HTTP server “include” files
pic32_solutions\microchip\bsd_tcp_ip\source	TCP/IP source files
pic32_solutions\microchip\include\bsd_tcp_ip\templates	HTTP server and TCP/IP template header files
pic32_solutions\microchip\bsd_tcp_ip\templates	HTTP server template source files
pic32_solutions\microchip\fat16\source	File I/O source files
pic32_solutions\microchip\include\fat16	File I/O header files
pic32_solutions\microchip\include\fat16\template	File I/O template header files

TABLE 4: SOURCE FILES

File	Directory	Description
bsd_http_server_demo.mcp	pic32_solutions\bsd_http_server_dhcp_demo	MPLAB HTTP server demo project
bsd_http_server_demo.mcw	pic32_solutions\bsd_http_server_dhcp_demo	MPLAB HTTP server demo workspace
httpex.c	pic32_solutions\bsd_http_server_dhcp_demo\source	User-modifiable HTTP server callback functions
main.c	pic32_solutions\bsd_http_server_dhcp_demo\source	Main demo file
httpex.h	pic32_solutions\bsd_http_server_dhcp_demo\source	User-modifiable HTTP server callback functions header
eTCP.def	pic32_solutions\bsd_http_server_dhcp_demo\source	User-modifiable TCP/IP defines
fat.def	pic32_solutions\bsd_http_server_dhcp_demo\source	User-modifiable FAT16 defines
http.c	pic32_solutions\microchip\bsd_tcp_ip\source\bsd_http_server	HTTP server library source
http_private.h	pic32_solutions\microchip\bsd_tcp_ip\source\bsd_http_server	HTTP server private defines
httpex.tmpl	pic32_solutions\microchip\bsd_tcp_ip\templates	User-modifiable HTTP server callback functions template
http.h	pic32_solutions\microchip\include\bsd_tcp_ip	HTTP ”include”
http.tmpl	pic32_solutions\microchip\include\bsd_tcp_ip\templates	User-modifiable HTTP defines, template
httpex.tmpl	pic32_solutions\microchip\include\bsd_tcp_ip\templates	User-modifiable HTTP server callback functions header, template

DEMO APPLICATION

Included with the Microchip HTTP Server is a complete working application to demonstrate the HTTP server running on the Microchip BSD TCP/IP stack. This application (the web server) is designed to run on Microchip's Explorer 16 demonstration board. However, it can be easily modified to support any board.

Demo HTTP Server Application Features

Version 1.0 of the demo HTTP Server application implements the following features. Refer to version log, through `index.html`, for version specific feature additions or improvements.

- Downloads files to a HTTP client
- Downloads dynamic files to a HTTP client
- Using the GET method to pass information, will toggle LEDs on the Explorer16 board
- Using the POST method to pass information, will display messages on the Explorer16 board's LCD.
- Displays the web authentication.

The main source file for this application is `main.c`. Users should refer to the source code as a starting point for creating their own applications, utilizing different aspects of the Microchip HTTP server.

In order to compile the project you must have the source code from AN1108, "*Microchip TCP/IP Stack with BSD Socket API*".

Programming Demo Application

To program a target board with the demo application, you must have access to a PIC® microcontroller programmer. The following procedure assumes that you will be using MPLAB REAL ICE in-circuit emulator as a programmer. If not, refer to the instructions for your specific programmer.

1. Connect MPLAB REAL ICE in-circuit emulator to the Explorer 16 board or to your target board.
2. Apply power to the target board.
3. Launch the MPLAB IDE.
4. Select the PIC device of your choice (this step is required only if you are importing a hex file that was previously built).
5. Enable the MPLAB REAL ICE in-circuit emulator as your programming tool.
6. If you want to use a previously built hex file, import the following file:
7. `pic32_solutions\bsd_http_server_dhcp_demo\release\bsd_http_server_dhcp_demo.hex`

   If you are rebuilding the hex file, open the project file: `pic32_solutions\bsd_http_server_dhcp_demo\bsd_http_server_dhcp_demo.mcp`, and follow the build procedure to create the application hex file.
7. The demo application contains necessary configuration options required for the Explorer 16 board. If you are programming another type of board, make sure that you select the appropriate oscillator mode from the MPLAB configuration settings menu.
8. Select the Program menu option from the MPLAB REAL ICE in-circuit emulator menu to begin programming the target.
9. After a few seconds, you should see the message "Programming successful". If not, check your board and your MPLAB REAL ICE connection. Click Help on the menu bar for further assistance.
10. Remove power from the board and disconnect the MPLAB cable from the target board.

Setting Demo Application Hardware

In order to run the HTTP demo correctly, you must set up the hardware on the Explorer16 board to use the TCP/IP stack and FAT16. Refer to AN1108, "*Microchip TCP/IP Stack with BSD Socket API*" for proper hardware setup.

The demo requires that the TCP/IP connection (Microchip Part Number AC164123) uses SPI 1 and the FAT16-type media storage device (Microchip Part Number AC164122) uses SPI 2.

The HTTP Server demo application provides DCHP server support. If no DCHP server is present, the default address that has been provided by `http.def` will be used.

Sample web pages are provided with this demo and have been saved, by default, to the following location: `pic32_solutions\bsd_http_server_dhcp_demo\webpages`. Copy the sample web pages to the SD media card so that the web page content will display properly.

Executing Demo Application

When the programmed microcontroller is installed on the Explorer 16 demo board and powered up, the system LED should blink to indicate that the application is running. The two-line LCD display will show the following information:

```
PIC32 BSD HTTP
<Current IP address>
```

When configured correctly and using the provided FAT16 module, the demo HTTP server will serve web pages. The sample web pages demonstrate which HTTP methods are supported. An authentication page is also included. The sample pages included with the Microchip stack source archive illustrate a modified form of CGI (a remote method invocation) and dynamic page generation (variable substitution).

Building the Demo HTTP Server

The demo HTTP server application included in this application note can be built using Microchip's MPLAB C32 C Compiler. However, you can port the source to whichever compiler that you routinely use with Microchip microcontroller products.

This application note includes a predefined HTTP server project file for use with Microchip MPLAB IDE: `bsd_http_server_dhcp_demo.mcp`. This project was created using a PIC32 device. If you are using a different device, you must select the appropriate device by using MPLAB menu command. In addition, the demo application project uses additional include paths as defined in the **Build Options** of MPLAB IDE.

```
.\source
..\microchip\include
```

Table 5 lists the source files that are needed to build the demo HTTP server application, and their respective locations.

The following instructions describe a high-level procedure for building the demo application. This procedure assumes that you are familiar with MPLAB IDE and will be using MPLAB IDE to build the application. If not, refer to the instructions for your in-circuit development environment to create and build the project.

1. Make sure that source files for the Microchip HTTP server are installed. If not, refer to **"Installing Source Files"**on page 3.
2. Launch MPLAB IDE and open the project file, `bsd_http_server_dhcp_demo.mcp`
3. Use the appropriate MPLAB IDE menu commands to build the project. Note that the demo project was created to compile properly when the source files are located in the directory structure that is suggested by the installation wizard. If you installed the source files to other locations, you must recreate or modify existing project settings to accomplish the build.
4. The build process should finish successfully.

   If not, make sure that your MPLAB IDE and compiler are set up correctly.

TABLE 5: DEMO HTTP SERVER APPLICATION PROJECT FILES

File	Directory
main.c	\pic32_solutions\bsd_http_server_dhcp_demo\source
httpex.c	\pic32_solutions\bsd_http_server_dhcp_demo\source
eTCP.def	\pic32_solutions\bsd_http_server_dhcp_demo\source
fat.def	\pic32_solutions\bsd_http_server_dhcp_demo\source
http.c	\pic32_solutions\microchip\bsd_tcp_ip\source\bsd_http_server
dhcp.c	\pic32_solutions\microchip\bsd_tcp_ip\source\bsd_dhcp_client
block_mdr.c	\pic32_solutions\microchip\bsd_tcp_ip\source
earp.c	\pic32_solutions\microchip\bsd_tcp_ip\source
eicmp.c	\pic32_solutions\microchip\bsd_tcp_ip\source
eip.c	\pic32_solutions\microchip\bsd_tcp_ip\source
ENC28J60.c	\pic32_solutions\microchip\bsd_tcp_ip\source
etcp.c	\pic32_solutions\microchip\bsd_tcp_ip\source
ether.c	\pic32_solutions\microchip\bsd_tcp_ip\source
eudp.c	\pic32_solutions\microchip\bsd_tcp_ip\source
gpfunc.c	\pic32_solutions\microchip\bsd_tcp_ip\source
pkt_queue.c	\pic32_solutions\microchip\bsd_tcp_ip\source
route.c	\pic32_solutions\microchip\bsd_tcp_ip\source
socket.c	\pic32_solutions\microchip\bsd_tcp_ip\source
tick.c	\pic32_solutions\microchip\bsd_tcp_ip\source
fat.c	\pic32_solutions\microchip\fat16\source
fileio.c	\pic32_solutions\microchip\fat16\source
mediasd.c	\pic32_solutions\microchip\fat16\source
mstimer.c	\pic32_solutions\microchip\common
exlcd.c	\pic32_solutions\microchip\common

USING THE HTTP SERVER

The installation files that accompany this application note contain the full source code for the Microchip HTTP server (see **“Source Code for the HTTP Server”** on page 37).

All applications based on the Microchip HTTP server must be written in a cooperative multitasking manner. Cooperative multitasking architecture consists of a number of tasks executing in sequence. A cooperative task would quickly perform its required operation and return so that the next task would be able to execute.

Because of this requirement, a task that needs to wait for some external input, or performs a long operation, should be broken down into subtasks using a state machine approach. Further discussion of cooperative multitasking and state machine programming is beyond the scope of this document. You should refer to software engineering literature for more detail.

To simplify file management and application development, all source files are located in subdirectories under the `source` directory. See **“Source File Organization”** page 3 for more information.

When you develop your application for the Microchip HTTP server, the directory structure of the demo as a reference to create your own application-specific subdirectory.

The following steps are typical of those you would use to develop an application based on the Microchip Stack. Note that these steps assume that you are using MPLAB IDE and are familiar with the interface.

1. Install the Microchip Stack source as described in **“Installing Source Files”** on page 3.
2. Create your application-specific directory in the `pic32_solutions` directory.
3. Use MPLAB IDE to create your application project and add the Stack source files as per your HTTP node functionality.
4. Use the MPLAB **Build Option** dialog box to set two additional include search paths:
   ```
   .\source
   ..\microchip\include
   ```
5. Add your application specific source files. Now your application project is ready for the build.

Integrating Your Application

The HTTP server included with this application note is implemented as a cooperative task that co-exists with the Microchip BSD TCP/IP Stack and your main application. The server, itself, is implemented in the source file, `http.c`, with a user application implementing seven callback functions. The demo application source file, `httpex.c`, should be used as a template application to create the necessary callback functions.

The main component of the server consists of the FTP server task..

In order to integrate the HTTP server into a user application, do the following:

1. Set the desired `MAX_HTTP_CONNECTIONS` value in the `http.def` header file.
2. Include the files `http.c` in the project.
3. Include files to support the TCP/IP stack and FAT16.
4. Modify the `main()` function of the application to include the HTTP server.

The HTTP server uses the file `index.htm` as its default web page. If a remote client (browser) accesses the HTTP server by its IP address or domain name only, `index.htm` is the default page served. This requires that all applications include a file named `index.htm`. If necessary, the name of this default file can be changed by modifying the compiler definition `HTTP_DEFAULT_FILE_STR` in the `http.def` file.

For HTTP authentication, the server will display the file `blocked.htm` if the user has entered an incorrect user name or password. If you wish to change the file that is displayed, modify compiler definition file: `HTTP_UNATHORIZED_FILE` located in `http.def`.

As a default, the authentication encryption method is BASIC64 for transmitting the user name and password to the server. To change the encryption method, modify the following compiler definition file `HTTP_AUTHENTICATION_METHOD` (it is located in `http.def).`

All HTTP filenames should be in FAT16 format: file names cannot be longer than eight characters, cannot contain spaces, and the extensions cannot be longer than three characters.

It is very important to make sure that *none* of the web page file names contain any of the following special alphanumeric characters:

- single or double quotes (‘ and “)
- left or right angle brackets (< and >)
- the pound sign (#)
- the percent sign (%)
- left or right brackets or braces ([, {,] and })
- the “pipe” symbol (|)
- the backslash (\ )
- the caret (^)
- the tilde (~)

If a file does contain any of these characters, the corresponding web page will become inaccessible. No prior warning will be given.

The HTTP server also maintains a list of file types that it supports. It uses this information to advise a remote browser on how to interpret a particular file, based on the file’s three-letter extension. By default, the Microchip HTTP server supports “`.txt`”, “`.htm`”, “`.gif`”, “`.cgi`”, “`.jpg`”, “`.cla`”, “`.wav`”, “`.js`” and “`.css`” files. If an application uses file types that are not included in this list, the user may modify the table “`_httpExtTbl`” in the file “`http.c`” to allow them.

HTTP SERVER TASK

The HTTP server task contains three functions the main application can call to set up, run, and stop HTTP protocol. They initialize the TCP/IP stack and create a listening socket. The functions also control incoming connections and requests, as well as closing all of the connections correctly.

- `HTTPInit`
- `HTTPServer`
- `HTTPCloseAll`

HTTPInit

HTTPInit will initialize the TCP/IP connection to handle incoming connections by an HTTP client.

Syntax

```
BOOL HTTPInit(void)
```

Parameters

None.

Precondition

The TCP/IP socket must be initialized.

Side Effects

None.

Remarks

HTTPInit should be called after FAT16 and TCP/IP have been initialized.

Example

```
// Initialize the TCP/IP
   TCPIPSetDefaultAddr();
   InitStackMgr();
   TickInit();

   // Initialize the FAT16 library.
   if ( !FSInit() )
   {
      // If failed to initialize the FAT16, set an error LED
      // Primary reasons for failure would be no SD card detected
      // Or badly formatted SD card.
      return -1;
   }

   if(!HTTPInit())
      return -1;
```

HTTPServer

HTTPServer controls the incoming connections and processes HTTP client requests.

Syntax

```
void HTTPServer(void)
```

Parameters

None.

Return Values

None.

Precondition

HTTPInit must have been called before HTTPServer, to work correctly. The TCP/IP service routine must be called before, or after, HTTPInit.

Example

```
// TCP/IP, HTTP and DHCP have all been initialized

   while(1)
   {
     StackMgrProcess();
     HTTPServer();
     DHCPTask();
   }
```

HTTPCloseAll

HTTPCloseAll is used to close all of the TCP/IP connections and deallocate any memory.

Syntax

```
void HTTPCloseAll(void)
```

Parameters

None.

Return Values

None.

Precondition

None.

Side Effects

The HTTP server will not accept or process any requests from the HTTP clients.

Remarks

This function will close all connections to the HTTP clients. In order to re-establish the HTTP server, the main application must call HTTPInit and then HTTPServer to handle the incoming HTTP client requests.

Example

```
// close all HTTP connects
  HTTPCloseAll();
```

HTTP SERVER MEMORY USAGE

When the HTTP server accepts an incoming client connection, the server allocates memory on the stack to manage the incoming requests. To limit the number of connections, modify the `HTTP_MAX_CONNECTIONS` in `http.def` file.

Memory usage is set as shown below:

- 44 bytes of memory that is unadjustable
- length of the file name (`HTTP_FILENAME_LEN`)
- length of the receive buffer (`HTTP_HTML_CMD_LEN + 1`)
- length of the transmit buffer (`HTTP_SEND_BUF_LEN + 1`)
- length of the number of `GET` parameters (`HTTP_MAX_ARGS * HTTP_VAR_LEN`)

The topical memory allocated for each connection would be 366 bytes. The user can modify the “upon” defines to increase or decrease the amount of memory allocated in `http.def`.

DYNAMIC HTTP PAGE GENERATION

The HTTP server can dynamically alter pages and substitute real-time information, such as input/output status. To incorporate this real-time information, the corresponding CGI file (`*.cgi`) must contain a text string `%xx`, where the ‘`%`’ character serves as a control code and ‘`xx`’ represents a two-digit variable identifier. The variable value has a range of 00-99. When the HTTP server encounters this text string, it removes the ‘`%`’ character and calls the `HTTPGetVar` function. If the page requires ‘`%`’ as a display character, it should be preceded by another ‘`%`’ character. For example, to display “23%” in a page, put “`23%%`”.

HTTPGetVar

HTTPGetVar is a callback from HTTP. When the HTTP server encounters a string '%xx' in a CGI page that it is serving, it calls this function. HTTPGetVar is implemented by the main user application and is used to transfer application specific variable status to HTTP.

Syntax

```
WORD HTTPGetVar(BYTE var, WORD ref, BYTE *val)
```

Parameters

var [in]

Variable identifier whose status is to be returned.

ref [in]

Call Reference. This reference value indicates whether this is a first call. After first call, this value is strictly maintained by the main application. HTTP uses the return value of this function to determine whether to call this function again for more data. Given that only one byte is transferred at one time with this callback, the reference value allows the main application to keep track of its data transfer. If a variable status requires more than single bytes, the main application can use ref as an index to data array that is to be returned. Every time a byte is sent, the updated value of ref is returned as a return value; the same value is passed on next callback. In the end, when the last byte is sent, the application must return HTTP_END_OF_VAR as a return value. HTTP will keep calling this function until it receives HTTP_END_OF_VAR as a return value.

Possible values for this parameter are:

Value	Meaning
HTTP_START_OF_VAR	This is the very first callback for a given variable for the current instance. If a multi-byte data transfer is required, this value should be used to conditionally initialize index to the multi-byte array that will be transferred for the current variable.
For all others	Main application specific value.

data [out]

One byte of data that is to be transferred.

Return Values

New reference value as determined by main application. If this value is other than HTTP_END_OF_VAR, HTTP will call this function again with return value from the previous call.

If HTTP_END_OF_VAR is returned, HTTP will not call this function and assumes that variable value transfer is finished.

Possible values for this parameter are:

Value	Meaning
HTTP_END_OF_VAR	This is a last data byte for a given variable. HTTP will not call this function until another variable value is needed.
For all others	Main application specific value.

Precondition

None

HTTPGetVar (Continued)

Side Effects

None

Remarks

Although this function requests a variable value from the main application, the application does not have to return a value. The actual variable value could be an array of bytes that may or may not be the variable value. Which information to return is completely dependent on the main application and the associated web page. For example, the variable ‘50’ may mean a JPEG frame of 120 x 120 pixels. In that case, the main application can use the reference as an index to the JPEG frame and return one byte at one time to HTTP. HTTP will continue to call this function until it receives `HTTP_END_OF_VAR` as a return value of this function.

Given that this function has a return value of 16 bits, up to 64 Kbytes of data can be transferred as one variable value. If more length is needed, two or more variables can be placed side-by-side to create a larger data transfer array.

Example 1

Consider the page “`status.cgi`” that is being served by HTTP.

`status.cgi` contains following HTML line:

```
…
<td>S3=%04</td><td>D6=%01</td><td>D5=%00</td>
…
```

During processing of this file, HTTP encounters the `%04` string. After parsing it, HTTP makes a callback `HTTPGetVar(4, HTTP_START_OF_VAR, &value)`. The main user application implements `HTTPGetVar` as follows:

```
WORD HTTPGetVar(BYTE var, WORD ref, BYTE *data)
{
    // Identify variable.
    // Is it RB5 ?
    if ( var == 4 )
    {
    // We will simply return ‘1’ if RB5 is high,
    // or ‘0’ if low.
        if ( PORTBbits.RB5 )
            *val = ‘1’;
        else
            *val = ‘0;
        // Tell HTTP that this is last byte of
        // variable value.
        return HTTP_END_OF_VAR;
    }
    else
    // Check for other variables...
    ...
}
```

HTTPGetVar (Continued)

Example 2

Assume that the page “status.cgi” needs to display the serial number of the HTTP web server device.

The page “status.cgi” being served by HTTP contains the following HTML line:

```
…
<td>Serial Number=%05</td>
…
```

While processing this file, HTTP encounters the ‘%05’ string. After parsing it, HTTP makes a callback HTTPGetVar(4, HTTP_START_OF_VAR, &value). The main application implements HTTPGetVar as follows:

```
WORD HTTPGetVar(BYTE var, WORD ref, BYTE *data)
{
     // Identify variable.
     // Is it RB5 ?.
     // If yes, handle RB5 value - will be similar to Example 1.
     // Is it “SerialNumber” variable ?
     if ( var == 5 )
     {
             // Serial Number is a NULL terminated string.
             // First of all determine, if this is very first call.
             if ( ref == HTTP_START_OF_VAR )
             {
                     // This is the first call.  Initialize index to SerialNumber
                     // string. We are using ref as our index.
                     ref = (BYTE)0;
             }
             // Now access byte at current index and save it in buffer.
             *val = SerialNumberStr[(BYTE)ref];
             // Did we reach end of string?
             if ( *val == ‘\0’ )
             {
                     // Yes, we are done transferring the string.
                     // Return with HTTP_END_OF_VAR to notify HTTP server that we
                     // are finished transferring the value.
                     return HTTP_END_OF_VAR;
             }
             // Or else, increment array index and return it to HTTP server.
             (BYTE)ref++;
             // Since value of ref is not HTTP_END_OF_VAR, HTTP server will call
             // us again for rest of the value.
             return ref;
     else
     // Check for other variables...
     ...
}
```

HTTP CGI

The HTTP server implements a modified version of CGI. With this interface, the HTTP client can invoke a function within HTTP and receive results in the form of a web page. A remote client invokes a function by HTML `GET` method with more than one parameter. Refer to "*RFC1866*" (the HTML 2.0 language specification) for more information.

When a remote browser executes a `GET` method with more than one parameter, the HTTP server parses it and calls the main application with the actual method code and its parameter. In order to manage this method, the main application must implement a callback function with an appropriate code.

The Microchip HTTP server does not perform "URL decoding". This means that if any of the form field text contains certain special alphanumeric characters (such as <, >, ", #, %, etc.), the actual parameter value would contain `%xx` (`xx` being the two-digit hexadecimal value of the ASCII character) instead of the actual character. For example, an entry of `<Name>` would return `%3CName%3C`.

A file that contains HTML form must have `.cgi` as its file extension.

HTTPExecCmd

HTTPExecCmd is a callback from HTTP. When the HTTP server receives a GET method with more than one parameter, it calls this function. HTTPExecCmd is implemented by the main application. This function must decode the given method code and take appropriate actions. Such actions may include supplying a new web-page name to be returned, and/or performing an I/O task.

Syntax

```
void HTTPExecCmd(BYTE **argv, BYTE argc)
```

Parameters

argv [in]

List of command string arguments. The first string (argv[0]) represents the form action, while the rest (argv[1..n]) are command parameters.

argc [in]

Total number of parameters, including form action.

Return Values

Main application may need to modify argv[0] with a valid web page name to be used as command result.

Precondition

None.

Side Effects

None.

Remarks

This is a callback from HTTP to the main application as a result of a remote invocation. There could be simultaneous (one after another) invocation of a given method. Main application must resolve these simultaneous calls and act accordingly.

By default, the number of arguments (or form fields) and total of argument string lengths (or form URL string) is limited to 5 and 80, respectively. The form fields limit includes the form action name. If an application requires a form with more than four fields and/or total URL string of more than 80 characters, the corresponding definitions of MAX_HTTP_ARGS and MAX_HTML_CMD_LEN (defined in http.def) must be increased.

Example

Consider the HTML page “Power.cgi”, as displayed by a remote browser:

```
<html>
<body><center>
<FORM METHOD=GET action=Power.cgi>
<table>
<tr><td>Power Level:</td>
<td><input type=text size=2 maxlength=1 name=P value=%07></td></tr>
<tr><td>Low Power Setting:</td>
<td><input type=text size=2 maxlength=1 name=L value=%08></td></tr>
<tr><td>High Power Setting:</td>
<td><input type=text size=2 maxlength=1 name=H value=%09></td></tr>
<tr><td><input type=submit name=B value=Apply></td></tr>
</table>
</form>
</body></html>
```

HTTPExecCmd (Continued)

This page displays a table with labels in the first column and text box values in the second column. The first row, first column cell contains the string Power Level; the second column is a text box to display and modify the power level value. The last row contains a button labelled **Apply**. A user viewing this page has the ability to modify the value in the **Power Level** text box and click on the **Apply** button to submit the new power level value to the Microchip stack.

Assume that a user enters values of '5', '1', and '9' in the **Power Level**, **Low Power Setting**, and **High Power Setting** text boxes, respectively, then clicks **Apply**. The browser would create a HTTP request string Power.cgi?P=5&L=1&H=9 and send it to the HTTP server. The server in turn calls HTTPExecCmd with the following parameters:

argv[0] = "Power.cgi", *argv[1]* = "P", *argv[2]* = "5", *argv[3]*="L", *argv[4]*="1", *argv[5]*="H", *argv[6]*="9"
a*rgc* = 7

The main application implements HTTPExecCmd as below:

```
void HTTPExecCmd(BYTE *argv, BYTE argc)
{
        BYTE i;
        // Go through each parameter for current form command.
        // We are skipping form action name, so i starts at 1...
        for (i = 1; i < argc; i++ )
        {
                // Identify parameter.
                if ( argv[i][0] == 'P' )        // Is this power level?
                {
                        // Next parameter is the Power level value.
                        PowerLevel = atoi(argv[++i]);
                }
                else if ( argv[i][0] == 'L' )   // Is this Low Power Setting?
                        LowPowerSetting = atoi(argv[++i]);
                else if ( argv[i][0] == 'H' )   // Is this High Power Setting?
                        HighPowerSetting = atoi(argv[++i]);
        }
        // If another page is to be displayed as a result of this command, copy
        // its upper case name into argv[0]
        // strcpy(argv[0], "RESULTS.CGI");
}
```

Note: For this example, the total number of arguments exceeds the default limit of 5. In order for this example to function properly, the value of MAX_HTTP_ARGS (located in http.def) must be set to at least 7.

HTTP POST

The HTTP server supports the POST method. The POST method sends data from the client to the server. This method is usually preferred over the GET method because it does not limit the amount of data that is transferred, while “hiding” the data from the user. When using the GET method, the data that was transferred to the server is displayed in the URL, where the POST method will not. The GET method also limits the number of characters that can be transmitted.

HTTP POST methods are commonly used in forms that might have many fields or comment dialogs. The data from these forms is sent in the body of the HTTP client request. The server will process the information and may choose to display a confirmation page notifying the user that the data was received.

HTTPSendVar

HTTPSendVar is a callback from HTTP. When the HTTP server receives a POST request, it will call this function to post the variable and value. The main application will implement this function. The function must decode the variable and its value to perform the desired method. The file name of the action making the POST request is passed to provide the method with proper decoding. The actions taken by this function may include updating a database file, I/O, or sending an e-mail. The function will send back an HTTP status code that will inform the server regarding the status of the request.

Syntax

```
HTTP_STATUS_CODES HTTPSendVar(BYTE *filename, BYTE *var, BYTE *value, BOOL end)
```

Parameters

filename [in]

Name of the file that was requested by the client.

var [in]

ASCII variable from the POST request.

value [in]

ASCII value from the POST request.

end [in]

A flag indicating that there is no more information to be passed.

Return Values

The main application will return an HTTP_STATUS_CODES enumeration, based on the actions that were performed.

HTTP_STATUS_OK – function was able to perform required task.

HTTP_STATUS_CREATED – a new file was created, based on the posted data.

HTTP_STATUS_ACCEPTED – the change requested by the data was accepted by the server.

HTTP_STATUS_NOT_MODIFIED – based on the data that was past, no modification was performed.

HTTP_STATUS_BAD_REQUEST – server was not able to process the request.

HTTP_STATUS_NOT_IMPLEMENTED – server was not able to process the request, due to the fact that the parameters passed have not been implemented.

Precondition

None.

Side Effects

None.

Remarks

This is a callback from HTTP to the main application. The variable and value are passed in ASCII format and any special characters that have already been formatted. The variable and value lengths are assigned in http.def and have a default of 80 characters. If the user would like to change the variable or value length, they need to modify HTTP_VAR_LEN and HTTP_VALUE_LEN, which are found in http.def, respectively.

Example

Consider the following HTML code that uses the POST method to send data to the server:

```
<form action=lcdwrite.cgi method=POST>
Line 1 <input type=text name=line1 size=18 maxlength=16><br>
Line 2 <input type=text name=line2 size=18 maxlength=16><br>
<input type=submit>
<input type=reset>
```

HTTPSendVar (Continued)

The HTML page will display two input boxes into which the user can type text. There will also be two buttons: **Reset** and **Submit**. **Reset** will clear the text boxes and **Submit** will send the information in the text box to the server.

The data that is sent will look like the following line, assuming that the user entered `Testing` on the first line and `1 2 3` on the second line:

```
line1=Testing&line2=1+2+3
```

The HTTP server will call the callback function twice with the following data:

```
....
HTTPSendVar(“lcdwrite.cgi”, “line1”, “Testing”, FALSE);
....
HTTPSendVar(“lcdwrite.cgi”, “line2”, “1 2 3”, TRUE);
....
```

The main application implements `HTTPSendVar` as shown below:

```
HTTP_STATUS_CODES HTTPSendVar(BYTE *filename, BYTE *var, BYTE *value, BOOL end)
{
    BYTE    i;
    BYTE    k;

    for(i = 0; i < HTTP_EX_POST_SIZE; i++)
    {
        if(!strcmp(filename, actionFile[i].filename))
            break;
    }

    if(i >= HTTP_EX_POST_SIZE)
        return HTTP_STATUS_NOT_FOUND;

    switch(i)
    {
        case 0:
            for(k = 0; k < HTTP_EX_POST_LCD_WRITE_SIZE; k++)
            {
                if(!strcmp(var, lcdwriteVar[k].filename))
                    break;
            }

            if(k >= HTTP_EX_POST_LCD_WRITE_SIZE)
                return HTTP_STATUS_NOT_IMPLEMENTED;

            strcpy(lcdLine[k].filename, value);

            {
             BYTE size;

             size =  (BYTE)strlen(lcdLine[k].filename);

             if(size < 16)
             {
                BYTE j;
                BYTE*ptr;

                ptr = lcdLine[k].filename;

                for(j = size; j < 16; j++)
                   ptr[j] = ' ';

                ptr[j] = '\0';
```

HTTPSendVar (Continued)

```
                }

              MyLcdWriteLine(k + 1, lcdLine[k].filename);
            }
              break;

    return HTTP_STATUS_OK;
}
```

Note: This function uses a constant look-up table to check support of the POST request. Also, the user may wish to use this method instead of HTTPExecCmd for processing parameters that are passed with the GET request. It they do so, define _HTTP_EXTEND_SET in http.def to allow the HTTP to use this callback when managing the GET request.

HTTP `PUT`

To upload files onto the server, the client will use the `PUT` method. The `PUT` method contains the file body for the server to save on the server. The method will either create or update the file on the server.

HTTPPUTUpload

HTTPPUTUpload is a callback from the HTTP. When the client sends a PUT request, the HTTP calls this function to check whether the file is allowed to be uploaded.

Syntax

```
BOOL HTTPPUTUpload(BYTE *filename)
```

Parameters

filename [in]

The file that has been requested to be uploaded on the server.

Return Values

If the file is allowed to be uploaded, the function will return TRUE, else FALSE.

Remarks

This callback function is used to protect the client from uploading files that could overwrite "protected" files. There could be four ways in which the main application accepts or rejects the file:

1. The main application checks for a specific file name that it will allow to be uploaded.
2. The main application allows only certain file extension to be uploaded.
3. The main application checks the file against a list of protected files or protected file extensions.
4. The main application does not allow certain file extensions to be uploaded.

It is recommended that the main application block the uploading of *.htm files, or at least have some sort of authentication requirement to access the web page that would allow such an action.

Example

The application can implement HTTPPUTUpload in any one of these combinations.

1. The main application could check for a specific file to upload:

```
BOOL HTTPPUTUpload(BYTE *filename)
{
     if(!strcmp(filename, "somefile.txt"))
        return TRUE;

    return FALSE;
}
```

2. The main application may allow for certain file extensions to be uploaded:

```
BOOL HTTPPUTUpload(BYTE *filename)
{
     while(*filename != '.' || *filename != '\0')
        filename++;

    if(*filename == '\0')
        return FALSE;

    if(!strcmp(filename, ".txt"))
        return TRUE;

    return FALSE;
}
```

HTTPPUTUpload (Continued)

3. The main application could restrict which file(s) can be uploaded:

```
BOOL HTTPPUTUpload(BYTE *filename)
{
   if(!strcmp(filename, "somefile.txt"))
      return FALSE;

   return TRUE;
}
```

4. The main application can restrict which types of files are uploaded by file extension:

```
BOOL HTTPPUTUpload(BYTE *filename)
{
    while(*filename != '.' || *filename != '\0')
      filename++;

   if(*filename == '\0')
      return FALSE;

   if(!strcmp(filename, ".txt"))
      return FALSE;

   return TRUE;
}
```

HTTPPUTSendFile

HTTPPUTSendFile is a callback from the HTTP. When the HTTP has uploaded a file, the main application might want to display a page to the client indicating the status of the upload. Most likely the file that is loaded by the client will be a dynamic file type (CGI).

Syntax

```
BOOL HTTPPUTSendFile(BYTE *filename, HTTP_STATUS_CODES status)
```

Parameters

filename [out]

The name of the file that will be downloaded to the client following the upload.

status [in]

The current status of the upload process. This parameter indicates the status of the uploading file. The main application might want to download a different file when the upload failed vs. when it is successful.

HTTP_STATUS_CREATED – file was successfully uploaded on the server.

HTTP_STATUS_INTERNAL_SERVER_ERR – error in uploaded file.

Return Values

If the function returns TRUE, the file will be downloaded to the client.

Precondition

None.

Side Effects

None.

Remarks

The callback from HTTP to the main application is for the client to display notification of the status of the PUT method. It is recommended that the main application download a page indicating success or failure to the client.

Example

The main application implements HTTPPUTSendFile as shown below:

```
BOOL HTTPPUTSendFile(BYTE *filename, HTTP_STATUS_CODES status)
{
    if(status == HTTP_STATUS_CREATED)
        strcpy(filename, "put_ok.cgi");
   else if(status == HTTP_STATUS_INTERNAL_SERVER_ERR)
        strcpy(filename, "put_err.htm");
   else
        return FALSE;

    return TRUE;
}
```

HTTP Authentication

The web developer may want to limit public access to certain pages or files on the server. HTTP has authentication to allow for this limited access. When a client tries to access a restricted page or file, the server will prompt the client for a user name and password. The client will try again to access the page, this time passing an encrypted user name and password. The main application is responsible for decrypting and verifying the user name and password. The HTTP server will pass the type of encryption method that the client will use to encrypt the user name and password. Since the type of encryption is made public when the server passes it to the client, it is recommended that the client double encrypt the data. That is, to take the user name and password and initially encrypt it using a Java Script, and then let the client encrypt it. This means that the server will also have to double decrypt the user name and password. It is also important to note that HTTP does not have a log out. This means that once the client has access to the pages; as long as the browser is open, it will resend the user name and password when accessing pages or files.

HTTPAuthorizationRequired

Like the callback function HTTPPUTUpload, HTTPAuthorizationRequired is used to decide whether the file needs to have authorization. Also like HTTPPUTUpload, this function can be implemented in more than one way. The HTTP server can require authorization on a file-by-file basis or based on the file extension. It is recommend that you use authorization on pages that will allow users to upload files onto the HTTP server.

Syntax

```
BOOL HTTPAuthorizationRequired(BYTE *filename)
```

Parameters

filename [in]

Name of the file requested by the client.

Return Values

If the file requires authorization to download, the function will return TRUE, else FALSE.

Precondition

None.

Side Effects

None.

Remarks

None.

Example

The main application implements HTTPAuthorizationRequired as shown below:

```
BOOL HTTPAuthorizationRequired(BYTE *filename)
{

    if(!strcmp(filename, "somefile.htm"))
        return TRUE;

    return FALSE;
}
```

HTTPChkAuthorization

A callback function, HTTPChkAuthorization will decrypt the user name and password sent by the client. The type of encryption will also be passed, so the callback function will use the correct decrypting algorithm. The decrypted user name and password will be in the form username:password.

Syntax

```
BOOL HTTPChkAuthorization(BYTE *type, BYTE *text)
```

Parameters

type [in]

The type of encryption used by the client.

text [in]

The encrypted user name and password sent by the client

Return Values

If the decrypted user name and password are authenticated, then the callback function will return a TRUE and allow the client to receive the requested file.

Precondition

None.

Side Effects

None.

Remarks

The encryption method that is most commonly used is BASIC64. It is not a secure method for encryption because it is so common and the method is passed along with the data in the HTTP header sent from the client. It is recommend that the main application have a decrypting algorithm instead of compares. For this example, a simple comparing of user name and password will be used. An encryption algorithm is not included as part of this app note.

Example

Note: The user name and password, admin:password, is YWRtaW46cGFzc3dvcmQ= when using BASIC64 to encrypt.

If the following data were passed, the main application would implement the function, as shown below:

```
HTTPChkAuthorization(“BASIC”, “YWRtaW46cGFzc3dvcmQ=”)
BOOL HTTPChkAuthorization(BYTE *type, BYTE *text)
{
    type = strupr(type);

    if(strcmp(type, "BASIC"))
        return FALSE;

    if(strcmp(text, "YWRtaW46cGFzc3dvcmQ="))
        return FALSE;

    return TRUE;
}
```

HTTP HEAD

The HTTP client may request the HEAD method. This method will not need any callback function by the main application. The HEAD request is nothing more than a sort of “are you there?” request. On receiving the request, the server will send the appropriate response.

HTTP Continue

To avoid sending large amounts of data that could not be processed by the server, the HTTP client can send a Continue request to the server. The client will only sent the header of a POST or PUT request, asking the server to send a Continue response. If the server wishes to process the body of the request, it will respond with the Continue response. The client will then send the remainder of the data. This is useful if data is large and could be rejected by the server. An example might be the PUT request that will upload a large 300K file. The client will send the header of the PUT request to make sure that the server is going to accept the uploading of this large file before sending it. Like the HEAD request, the main function will not need to implement any callbacks for this feature.

USER-MODIFIABLE DEFINES

`HTTP_MAX_CONNECTIONS`

Location

`http.def`

Recommended Values

1 - 10

Remarks

The number of connections is limited by this value. One should consider the amount of memory that will be allocated when assigning a value to this define.

`HTTP_HTML_CMD_LEN`

Location

`http.def`

Recommended Values

64-126

Remarks

This define is the buffer size of the TCP/IP receive. For each connection, a receive buffer for it is allocated on the heap. Decreasing the define will also decrease the size of the data that you are able to process. An example would be if you define `HTTP_HTML_CMD_LEN` to be 64 characters, yet you have a value that could be 80 characters, it will not be processed correctly. It is recommended that web pages that pass information to the server have a limit on the amount of characters that it can pass.

`HTTP_VAR_LEN`

Location

`http.def`

Recommended Values

At least half of `HTTP_HTML_CMD_LEN`. or a maximum of `HTTP_HTML_CMD_LEN - (HTTP_VALUE_LEN + 3)`

Remarks

This define is a limit on the size of the variable identifier that is passed in the `GET` or `POST` request. The developer should consider this when designing the web page, to protect against incorrect processing.

`HTTP_VALUE_LEN`

Location

`hhtp.def`

Recommended Values

At least half of `HTTP_HTML_CMD_LEN` or a maximum of `HTTP_CMD_LEN - (HTTP_VAR_LEN + 3)`

Remarks

This define is a limit on the size of the value that is passed with the variable in the `GET` or `POST` request. The developer should consider this when designing the web page, by putting maximum lengths on the input that the client may enter and send to the server.

`HTTP_FILENAME_LEN`

Location

`http.def`

Recommended Values

13 - 32

Remarks

This define is for the requested web page from the client. The value should be at least 13 characters to hold an 8-character filename plus the '.' and 3-character extension. If the user is using a file system that supports long file names, the value can be greater. One should remember that the each connection will have a file name buffer.

`HTTP_SEND_BUF_LEN`

Location

`http.def`

Recommend Values

64-1536

Remarks

This define is the buffer length that is used to store data that is sent via TCP/IP. The buffer should be as small as possible, because each connection will have a transmit buffer.

`HTTP_MAX_ARGS`

Location

`http.def`

Recommended Values

Odd numbers, starting with a minimum value of 3.

Remarks

This should be an odd number because the first argument will be the file name. One should remember that all of the parameters that are passed need to fit in the receive buffer.

`HTTP_DEFAULT_FILE_STR`

Location

`http.def`

Recommended Value

`Index.htm`

Remarks

This define is the file name that is used when the client does not send any file to download. This is usually seen on the first `GET` request. The user needs to make sure that the file name corresponds with a file in the file system.

```
HTTP_AUTHENTICATION_METHOD
```

Location

```
http.def
```

Recommend Value

```
Basic\r\n
```

Remarks

This define is which type of encryption the client will use when encrypting the user name and password for a web page that requires authentication. There MUST be a `return` and `newline` character after the encryption scheme (`\r\n`).

```
HTTP_UNATHORIZED_FILE
```

Location

```
http.def
```

Recommended Value

```
blocked.htm
```

Remarks

This define is the file name that the browsers will display when the user has tried to go into a unauthorized web page. The file name MUST have a corresponding file in the file system.

ANSWERS TO COMMON QUESTIONS

Q: Why am I not able to serve all of the connections?

A: There could be two reasons that the HTTP server does not properly serve all of the connections. The first reason could be that the TCP/IP stack is already handling the maximum number of connections. To correct this, see AN1108, "*Microchip TCP/IP Stack with BSD Socket API*" to learn how to increase the number of supported connections.

If the number of TCP/IP connections is not the cause, the user may need to change the `HTTP_MX_CONNECTIONS` in `http.def`. Increasing this define will increase the amount of memory that the server allocates when handling connections. It is recommended that the user calculate the amount of memory that would be allocated in a worst case scenario.

Q: Why are my pages downloading so slowly?

A: The problem could be that you are trying to download a large page. Remember, this is an embedded system and its resources are not the same as those in a PC. You might want to change clock speeds to increase processing power, or modify the page to download in "frames." Using frames will allow the smaller parts of the page to be downloaded.

Increasing the transmit buffer length will increase the amount of data sent in a TCP/IP packet. However, it is recommend that you make this length as small as possible, because the server allocates this buffer on the heap for each connection that is made on the server .

Q: Why am I not able to view any web pages?

A: Make sure that the link name for that page is correct. It is recommended that you use page names that are eight characters in length and have extensions that are three characters. Also make sure that the format of the page (extension) is supported by the HTTP server. The table that has page formats and file extensions is in `http.c`, `_httpExtTbl`. The user may modify the table, but they must remember that they are responsible for also updating the `HTTP_FILE_EXT_ENUM` enumeration located in `http.h`.

Make sure that the IP address that the client is using to request web pages is correct. You may do this by checking the `eTCP.def` file in the TCP/IP stack source code (AN1108).

Check that the file system is configured properly.

Make sure that the device is properly configured for the target hardware.

Q: Why is the server not processing the parameters passed by the HTTP client?

A: If the callback function is not handling the passed parameters correctly, you might have two problems. Make sure that the callback function is implemented correctly. You might not be recognizing the passed file correctly, or that the variable is a valid one. Check the web page source code to make sure that the variable name matches the variable name that you are expecting. Make sure that the variable-plus-value character length is not greater than the receive buffer length.

A good way to verify that your data is being passed, is to use LEDs or a dynamically loaded page that may echo the data that was passed.

Q: Why is a web page that needs authorization being displayed without prompting the user for authentication?

A: HTTP does not log out the user when it loads new pages. The authorization will be sent every time with requesting pages. If the user has already entered a correct user name and password and the browser is open, they will be able to load that page.

Make sure that the callback function that tells the HTTP server that the requested page needs authorization is implemented correctly. If this function is not properly identifying the pages that require authentication, then a “protected” page could be displayed.

The callback function that decrypts the incoming user name and password needs to correctly decode and evaluate that the passed user name and password is valid. Make sure that the encryption method used does not allow incorrect information to be decrypted and mistaken for valid information.

Q: Why is a web page that requires authorization not displaying, even though I am using the correct user name and password?

A: The encryption method that you are using could be case sensitive, make sure that the data entered is in the same format that it was registered in. It is recommended that the user name and password not have any special characters. Try to use 0-9, a-z, and A-Z in the user name and password. Limit the size of the password and user name to fit in the receive buffer.

Make sure that the encryption method that you are passing to the client is the same method that you are using to decrypt the data. Check to make sure that the user name and password that you are using are valid for the page that you are trying to display. You may have different user name and passwords to display different pages. The client will only send the most recent user name and password to the server.

Q: Why won’t the server save files that I upload?

A: Check the callback function that determines whether the file is allowed to be uploaded on the server. Using the `HTTPPutSendFile` callback function, have the client display a CGI file that may display the name of the file that was saved; or an error page, if the upload fails, with a reason code.

Make sure that the file system hardware and software that you are using is working correctly.

The storage device that you are using could be filled and not allowing more files to be uploaded.

Q: How do I guard against people uploading web pages on my server?

A: It is highly recommended that you require authorization to access web pages that will allow you to upload any type of file, especially if it would allow a client to upload web pages. Another provision that should be taken is to use the callback function `HTTPPutUpload` to guard against uploading web pages. This can easily done by not allowing “`.htm`” and “`.cgi`” files to be uploaded.

CONCLUSION

The HTTP server presented here provides another protocol option for the BSD Socket TCP/IP Stack. Together with the stack and the user’s application, it provides a compact and efficient web page provider that can run on any of the PIC32MX 32-bit microcontrollers. Its ability to run independently of an RTOS or application makes it versatile.

PARTIAL LIST OF RFC DOCUMENTS

RFC Document	Description
RFC 826	Ethernet Address Resolution Protocol (ARP)
RFC 791	Internet Protocol (IP)
RFC 792	Internet Control Message Protocol (ICMP)
RFC 793	Transmission Control Protocol (TCP)
RFC 768	User Datagram Protocol (UDP)
RFC 821	Simple Mail Transfer Protocol (SMTP)
RFC 1055	Serial Line Internet Protocol (SLIP)
RFC 1866	Hypertext Markup Language (HTML 2.0)
RFC 2616	Hypertext Transfer Protocol (HTTP) 1.1
RFC 1541	Dynamic Host Configuration Protocol (DHCP)
RFC 1533	DHCP Options
RFC 959	File Transfer Protocol (FTP)

The complete list of Internet RFCs and the associated documents is available on many Internet web sites. Interested readers are referred to **www.faqs.org/rfcs** and **www.rfc-editor.org** as starting points.

APPENDIX A: SOURCE CODE FOR THE HTTP SERVER

The complete source code for the HTTP Server for the Microchip BSD TCP/IP Stack, including the demo applications and necessary support files, is offered under a no-cost license agreement. It is available for download as a single archive file from the Microchip corporate web site, at:

www.microchip.com.

After downloading the archive, always check the file `version.log` for the current revision level and a history of changes to the software.

REVISION HISTORY

Revision A (10/2007)

This is the initial released version of this document.

Revision B (03/2008)

Revised “Installing Source Files” section; Revised Tables 3, 4 and 5.

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE**.** Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

Trademarks

The Microchip name and logo, the Microchip logo, Accuron, dsPIC, KEELOQ, KEELOQ logo, MPLAB, PIC, PICmicro, PICSTART, PRO MATE, rfPIC and SmartShunt are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

FilterLab, Linear Active Thermistor, MXDEV, MXLAB, SEEVAL, SmartSensor and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Analog-for-the-Digital Age, Application Maestro, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICworks, dsSPEAK, ECAN, ECONOMONITOR, FanSense, In-Circuit Serial Programming, ICSP, ICEPIC, Mindi, MiWi, MPASM, MPLAB Certified logo, MPLIB, MPLINK, mTouch, PICkit, PICDEM, PICDEM.net, PICtail, PIC^{32} logo, PowerCal, PowerInfo, PowerMate, PowerTool, REAL ICE, rfLAB, Select Mode, Total Endurance, UNI/O, WiperLock and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

All other trademarks mentioned herein are property of their respective companies.

Printed on recycled paper.

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-572-9526
Fax: 886-3-572-6459

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

01/02/08