



MODTRONIX
ENGINEERING

Modular Electronic Solutions



Modtronix Tcp/Ip stack

Table of Contents

1 Introduction.....	3
2 Defines.....	3
3 DYNAMIC HTTP PAGE GENERATION.....	4
3.1 HTTPGetVar.....	4
4 HTTP CGI.....	6
4.1 HTTPExecCmd.....	7
5 MICROCHIP FILE SYSTEM (MPFS).....	9
5.1 Format.....	9
6 MPFS Image Builder.....	9
6.1 MPFS Access Library.....	10

1 Introduction

The Modtronix TCP/IP stack is a modified version of the original Microchip TCP/IP stack. See file headers for licence agreements. Large parts of this document have been copied from Microchip's Application note AN833 by Nilesh Rajbharti.

2 Defines

The following defines have been added to the stack. They are in addition to the original defines.

<i>Define</i>	<i>Values</i>	<i>Used By</i>	<i>Description</i>
APP_USE_LCD	N/A	websrvr.c	Defines if the LCD display is used or not.
BRD_XXX	BRD_SBC44EC BRD_SBC45EC BRD_SBC65EC	Mac.c	Defines the board type.

3 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 '%nxx', where the '%' character serves as a control code, the 'n' character represents the variable group and 'xx' represents a two-digit variable value (in upper case hex format). The variable value has a range of 00-FF (Which translates to 0-255 decimal), and must use upper case characters! The variable group ('n' character) can be any alpha numeric character ('0-9', 'a-z', 'A-Z'), giving a total of $10+26+26 = 62$ groups. Each group can have 256 variable values. This gives a total of 15872 possible variables.

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%%".

3.1 HTTPGetVar

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

Syntax

```
WORD HTTPGetVar(HTTP_VAR httpVar, WORD ref, BYTE* val)
```

Parameters

httpVar [in]

Variable identifier whose status is to be returned.

ref [in]

Call Reference. This reference value indicates if this is a very 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 a 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 bytes, the main application can use *ref* as an index to data array 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:

<i>Value</i>	<i>Meaning</i>
HTTP_START_OF_VAR	This is the very first callback for 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 current variable.
For all others	Main application specific value.

val [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 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:

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

Pre-Condition

None

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. What 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 a 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=%a04</td><td>D6=%d06</td><td>D5=%d05</td>
...
```

During processing of this file, HTTP encounters the '%a04' string. After parsing it, HTTP makes a callback `HTTPGetVar` (`a4`, `HTTP_START_OF_VAR`, `&value`). The main user application implements `HTTPGetVar` as follows:

```
WORD HTTPGetVar(HTTP_VAR httpVar, WORD ref, BYTE *val)
{
    //Identify variable group
    if (httpVar.group = 'a')
    {
        // Identify variable value.
        // Is it RA4 ?
        if ( httpVar.val == 4 )
        {
            // We will simply return '1' if RA4 is high, or '0' if low.
            if ( PORTBbits.RA4 )
                *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 values...
            ...
    }
}
```

For more detail, refer to "webpages*.cgi" files and the corresponding callback in the "httpexec.c" source file.

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=%sF0</td>
```

While processing this file, HTTP encounters the ‘%sF0’ string. After parsing it, HTTP makes a callback

HTTPGetVar(httpVar, HTTP_START_OF_VAR, &value). The main application implements HTTPGetVar as follows:

```
WORD HTTPGetVar(HTTP_VAR httpVar, WORD ref, BYTE* val)
{
    //Identify variable group
    if (httpVar.group = 'a')
    {
        // Identify variable value.
        // Is it s05 ?
        if ( httpVar.val == 0xf0)
        {
            // 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...
        }
    }
    ...
}
```

For more detail, refer to “Webpages*.cgi” files and the corresponding callback in the “httpexec.c” source file.

4 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 handle this method, the main application must implement a callback function with an appropriate code. The Modtronix HTTP Server does not perform “URL decoding”. This means that if any of the form field text contains certain special non-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”. See "The Microchip HTTP Server" (page 77) for a complete list of characters. A file that contains HTML form, must have “.cgi” as its file extension.

4.1 HTTPExecGetCmd

This function is a callback from HTTP. When the HTTP server receives a GET method with more than one parameter, it calls this function. This callback function is implemented by the main application. This function must repetively call HTTPGetParam() function until all name-value parameters sent with the GET command have been obtained. It must than decode the name-value parameter and take appropriate actions. Such actions may include supplying new Web page name to be returned and/or performing an I/O task.

Syntax

```
BYTE HTTPGetParam(TCP_SOCKET s, BYTE* param, BYTE* paramLen)
```

Parameters

s [in]

Socket that is currently receiving this HTTP command.

param[out]

The param buffer where the null terminated name and value of this parameter will be written to.

Return Values

Main application may need to modify *rqstRes* with a valid web page name to be used as command result.

Pre-Condition

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.

This function must repetively call HTTPGetParam() function until all name-value parameters sent with the GET command have been obtained.

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=%p07</td></tr>
<tr><td>Low Power Setting:</td>
<td><input type=text size=2 maxlength=1 name=L value=%p08</td></tr>
<tr><td>High Power Setting:</td>
<td><input type=text size=2 maxlength=1 name=H value=%p09</td></tr>
<tr><td><input type=submit name=B value=Apply></td></tr>
</table>
</form>
</body></html>
```

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 “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 Power Level, Low Power Setting and High Power Setting text boxes respectively, then clicks on the “Apply” button. 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 HTTPExecGetCmd.

The main application implements HTTPExecGetCmd as below:

```
void HTTPExecGetCmd(TCP_SOCKET s, BYTE* rqstRes)
{
    BYTE param[15];    //This buffer will hold the name-value parameters

    //Get next name-value parameter until we have retrieved them all
    do {
        valueIdx = (BYTE)sizeof(param);    //Input parameter is size of param buffer

        //Get name-value parameters. Returns true if there are more
        //name-value parameters to follow
        //- Pointer to Name parameter = &param[0]
        //- Pointer to Value parameter = &param[valueIdx]
        moreParams = HTTPGetParam(s, param, &valueIdx);

        //Identify parameter - is the name part
        if ( param[0] == 'P' ) // Is this power level?
        {
            //The value is the Power level value.
            PowerLevel = atoi(&param[valueIdx]);
        }
        else if ( param[0] == 'L' ) // Is this Low Power Setting?
            LowPowerSetting = atoi(&param[valueIdx]);
        else if ( param[0] == 'H' ) // Is this High Power Setting?
            HighPowerSetting = atoi(&param[valueIdx]);
    } while (moreParams);

    // If another page is to be displayed as a result of this command, copy
    // its upper case name into rqstRes
    // strcpy(rqstRes, "RESULTS.CGI");
}
```

Note: For more detail, refer to the “webpages*.cgi” files and the corresponding callback in the “httpexec.c” source file.

4.2 HTTPGetParam

This function is called withing the HTTPExecGetCmd to retrieve all parameters passed via the HTTP GET function. Ensure that the param buffer passed to this function is large enough to hold the null terminated name and value string. If not, they will be truncated.

Syntax

```
extern void HTTPExecGetCmd(TCP_SOCKET s, BYTE* rqstRes)
```

Parameters

s [in]
Socket that is currently receiving this HTTP command. This is the socket that is to be used for consequent HTTPGetParam() calls.

rqstRes[in]
Name of the Requested resource - GET command's action.

Return Values

Main application may need to modify *rqstRes* with a valid web page name to be used as command result.

Pre-Condition

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.

This function must repetively call HTTPGetParam() function until all name-value parameters sent with the GET command have been obtained.

Example

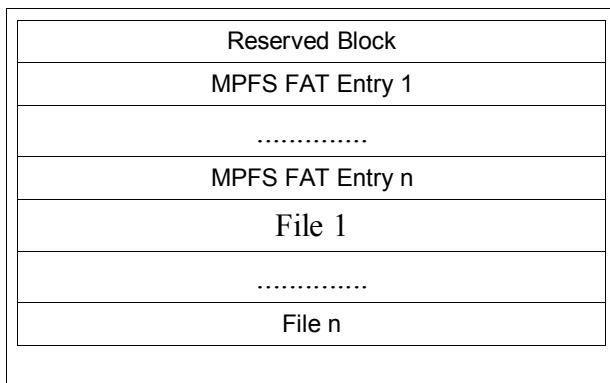
Consider the HTML page "power.cgi", as displayed by a remote browser:

5 MICROCHIP FILE SYSTEM (MPFS)

5.1 Format

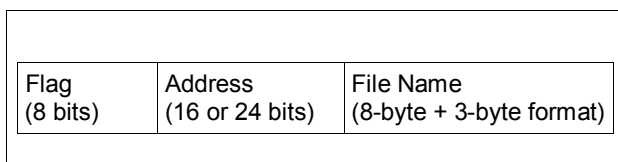
The Microchip HTTP Server uses a simple file system (the Microchip File System, or "MPFS") to store Web pages. The MPFS image can be stored in on-chip program memory or an external serial EEPROM. MPFS follows a special format to store multiple files in given storage media, which is summarized in Figure 1.

Figure 1 - MPFS IMAGE FORMAT



The length of "Reserved Block" is defined by `MPFS_RESERVE_BLOCK`. The reserved block can be used by the main application to store simple configuration values. MPFS storage begins with one or more MPFS FAT (File Allocation Table) entries, followed by one or more file data. The FAT entry describes the file name, location and its status. The format for the FAT entry is shown in Figure 2.

Figure 2 - MPFS FAT ENTRY FORMAT



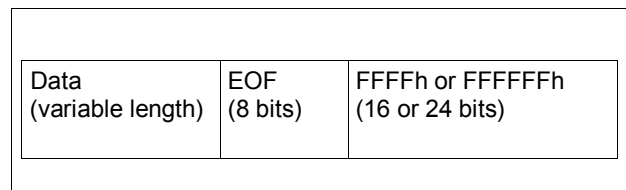
The *Flag* indicates whether the current entry is in use, deleted, or at the end of the FAT.

Each FAT entry contains either a 16-bit or 24-bit address value. The address length is determined by the type of memory used, as well as the memory size model. If internal program memory is used, and the Microchip TCP/IP Stack project is compiled with a small memory model, 16-bit addresses are used. If internal program memory and the large memory model are selected, a 24-bit address is used. The 16-bit addressing scheme is always used for external EEPROM devices, regardless of the memory size model. This implies a maximum MPFS image size of 64 Kbytes for these devices.

MPFS uses "short" file names of the "8 + 3" format (8 bytes for the actual file name and 3 bytes for the extension, or `NNNNNNNN.EEE`). The 16-bit address gives the start of the first file data block. All file names are stored in upper case to make file name comparisons easier.

The address in each FAT entry points in turn to a data block that contains the actual file data. The data block format is shown in Figure 3. The block is terminated with a special 8-bit flag called EOF (End Of File), followed by `FFFFh` (for 16-bit addressing), or `FFFFFFh` (24-bit addressing). If the data portion of the block contains an EOF character, it is stuffed with the special escape character, DLE (Data Link Escape). Any occurrence of DLE itself is also stuffed with DLE.

Figure 3 - MPFS DATA BLOCK FORMAT



6 MPFS Image Builder

This application note includes a special PC-based program (`MPFS.exe`) that can be used to build MPFS image from a set of files. Depending on where the MPFS will ultimately be stored, the user has the option to generate either a 'C' data file or binary file representing the MPFS image.

The complete command line syntax for `MPFS.exe` is `mpfs [/?] [/c] [/b] [/r<Block>] <InputDir> <OutputFile>`

where:

- `/?` displays command line help
- `/c` generates a 'C' data file
- `/b` generates a binary data file (default output)
- `/r` reserves a block of memory at beginning of the file (valid only in Binary Output mode, with a default value of 32 bytes)

`<InputDir>` is the directory containing the files for creating the image

`<OutputFile>` is the output file name

For example, the command `mpfs /c <Your WebPage Dir> mypages.c` generates the MPFS image as a 'C' data file, `mypages.c`, from the contents of the directory "Your Web Pages".

In contrast, the command `mpfs <Your WebPage Dir> mypages.bin` generates a binary file of the image with a 32-byte reserved block (both binary format and the 32-byte block are defaults), while

```
mpfs /r128 <Your WebPage Dir> mypages.bin
```

generates the same file with a 128-byte reserved block.

Note: Using a reserve block size other than the default of 32 bytes requires a change to the compiler define `MPFS_RESERVE_BLOCK` in the header file "StackTsk.h".

all of the Web pages and related files and save in a single directory. If the file extension is ".htm", the Image Builder strips all carriage return and linefeed characters to reduce the overall size of the MPFS image. If the MPFS image is to be stored in internal program memory, the generated 'C' data file must be linked with the "webservr" project. If the image is to be stored in an external serial EEPROM, the binary file must be downloaded there. For more information, refer to "The Microchip FTP Server" (page 85).

The MPFS Image Builder does not check for size limitations. If the binary data format is selected, verify that the total image size does not exceed the available MPFS storage space.

6.1 MPFS Access Library

The source file "MPFS.c" implements the routines required to access MPFS storage. Users do not need to understand the details of MPFS in order to use HTTP. All access to MPFS is performed by HTTP, without any help from the main application.

The current version of the MPFS library does not allow for the addition or deletion of individual files to an existing image; all of the files comprising a single MPFS image are added at one time. Any changes require the creation of a new image file.

If internal program memory is used for MPFS storage, `MPFS_USE_PGRM` must be defined. Similarly, if external data EEPROM is used for MPFS storage, `MPFS_USE_EEPROM` must be defined. Only one of these definitions can be present

in "StackTsk.h"; a compile-time check makes certain that only one option is selected.

Depending on the type of memory device used, its page buffer size will vary. The default setting of the page buffer size (as defined by `MPFS_WRITE_PAGE_SIZE` in the header file "MPFS.h") is 64 bytes. If a different buffer size is required, change the value of `MPFS_WRITE_PAGE_SIZE` appropriately.

Note: This version of the MPFS access library uses the file "xeeprom.c" for access to external data EEPROMs. When a file is being read or written, MPFS exclusively controls the I2C bus and will not allow any other I2C slave or master device to communicate. Users creating applications with multiple I2C devices need to bear this in mind.