



TCP/IP Networking, Part 3: Advanced Web-Based Control

Microchip TCP/IP Stack HTTP2 Module

© 2007 Microchip Technology Incorporated. All Rights Reserved.

Building Embedded Web Applications

Slide 1

Welcome to the next web seminar in Microchip's series on embedded web applications. We'll be discussing the HTTP2 web server module included with Microchip's free TCP/IP Stack, and more specifically, how you can use web-based control to accept input to your embedded application.

My name is Elliott Wood, and I'm an Applications Engineer at Microchip's headquarters in Chandler, AZ. I am one of the main developers of Microchip's TCP/IP Stack, and I'll be explaining how to build a simple web-based control system for an embedded product.

Project Overview

- **Web Enabled Vending Machine**
 - **Online Features:**
 - Update product names and prices
- www.microchip.com/tcpip
- **Scope:**
 - Vending machine and web pages are written
 - Just need to add the HTTP2 callbacks

In previous seminars, we added web-based monitoring capabilities to a simple vending machine. We also added web-based control for the lights. If you haven't watched these seminars, you may want to do so before continuing. This presentation will assume a basic understanding of the HTTP2 web server module, and how to use dynamic variables.

In this seminar we will extend that application. When the seminar is complete, you'll also be able to modify the product names and prices from any web browser.

If you're just viewing out of curiosity, or to see what's possible, all you need is a general understanding of embedded design. The demonstrations presented today assume either a PICDEM.net 2 development board, or an Explorer 16 with the Ethernet PICTail Plus daughterboard.

If you want to follow along with the examples, you will need to have downloaded and installed the full TCP/IP Stack distribution from the web page shown on the slide. We will be using the TCPIP WebVend App project that is installed with the stack. You should also have at least completed the Getting Started section in the TCPIP User's Guide for one of the previously mentioned boards, and know how to use the MPFS2 utility.

Just as in previous seminars, we will assume that a web developer has already designed the web pages for us. We will start with the working vending machine application from the last session, plus an additional new web page. Our task will be to add the required hooks between the form on the web page, and our embedded application. If you're following along, you should copy the base "WebPages2" folder and "CustomHTTPApp.c" source file from the "03 – Control with POST" directory inside the "Labs" folder.

Agenda

- **Web Forms Review**
- **Processing POST Method Forms**
- **Demonstration**

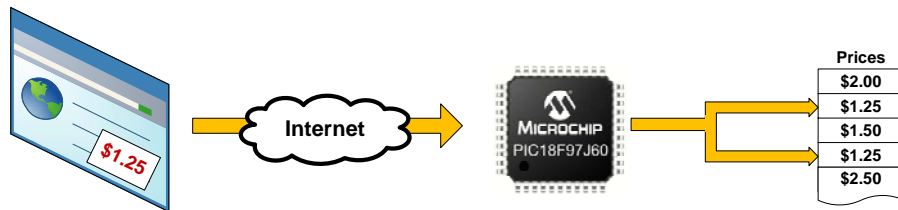
We will start with a quick review of web forms and the different methods. We will then learn how to process data submitted via POST. We will walk through an example, then show a demonstration at the end.

Let's get started.

Web Forms Review

[no narrative]

Web Forms Review



- **Accept input through web pages**
- **Control system outputs and data online**

As you hopefully remember, web forms allow the web server module to accept data from users through a network. This data can be used to control system memory or outputs. In the simplified example on the slide, the microcontroller accepts new prices from the web page, and responds by updating the pricing table in its memory.

Form Methods

- **GET**

`<form method="get" ...>`

- Data appended to URL
- Length limited to ~100 bytes
- Easiest to process

- **POST**

`<form method="post" ...>`

- Data sent as request body
- Length is unlimited
- More difficult to process

Recall that there are two different methods for submitting data to a server. The first, “GET”, was covered in the previous seminar. It was simple to process, but could only accept limited amounts of data.

This webinar will cover the “POST” method. This method sends the data as the body of the request, following the headers. On the Microchip platform, this data is more difficult to process because the application assumes responsibility for the TCP buffer. However, this additional complexity allows unlimited amounts of data to be received.

The POST Method

Let's take a look at how that would work.


The POST Method

- **Data sent as request body**
- **Full control of the buffer**
 - Unlimited data length
 - Data arrives as space is available
- **More difficult to process**
 - May arrive in different order or in pieces
 - Application manages arrival and decoding

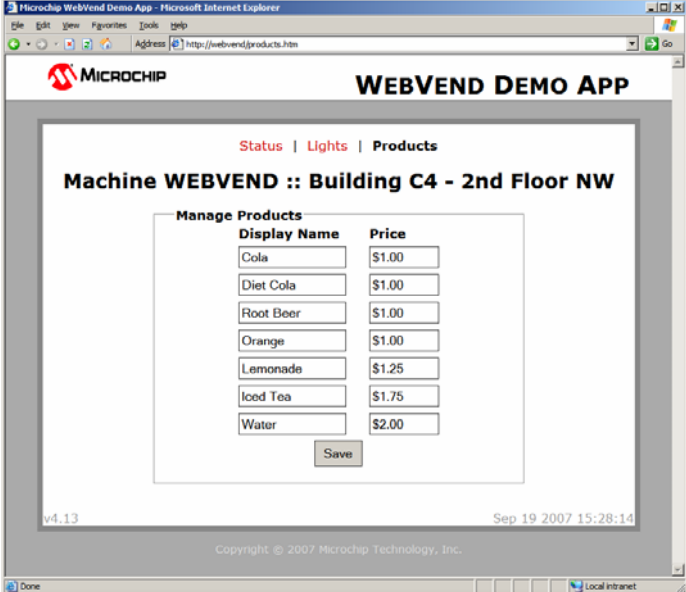
The POST method differs from the GET method in that data is not appended to the URL, but rather sent as the request body and after all the headers. It is left in the TCP buffer, and your application therefore gains full control of the data as it arrives. This allows unlimited lengths of data to be processed, with each new chunk arriving as space is available.

This additional control however brings additional complexity. Since data will arrive in pieces, form fields that depend on others may be more difficult to process as you may not know all values at once. Your application will be responsible for retrieving data as it is available, and determining what can safely be discarded to make room for more.

Still, this is the best method for forms that will be receiving lots of data at once.



POST Method Example



Display Name	Price
Cola	\$1.00
Diet Cola	\$1.00
Root Beer	\$1.00
Orange	\$1.00
Lemonade	\$1.25
Iced Tea	\$1.75
Water	\$2.00

© 2007 Microchip Technology Incorporated. All Rights Reserved. Building Embedded Web Applications Slide 9

As an example, we will work with the Products page, “products.htm”. This page accepts new names and prices for each of the seven products offered in the vending machine.

POST Method Example

- **GOALS:**

- Modify product names and prices



Display Name	Price
Cola	\$1.00
Diet Cola	\$1.00
Root Beer	\$1.00
Orange	\$1.00
Lemonade	\$1.25
Iced Tea	\$1.75
Water	\$2.00

- **SOLUTION:**

- Create form to accept all data at once
- More than 100 bytes, so use POST
- Process one variable at a time

Our form will accept 10 bytes for each name, plus 5 for each price. That's already close to 100 bytes before we take into account the overhead associated with naming and delimiting each field. Looks like POST is the best option for receiving this data. We will retrieve one field at a time from the TCP buffer, process it, and immediately discard that data to make room for more.

POST Example: products.htm

```
<td>
<input type="text" name="name[0]"
  value="~name(0)~" maxlength="9"
  size="15" /></td>

<td>
<input type="text" name="price[0]"
  value="~price(0)~" maxlength="5"
  size="8" /></td>
```

Once again, the web page has already been provided. We'll just open it up to take a brief look so we know what to expect. We see that there are seven blocks of code, each with two `<input>` tags. One of those blocks is shown on this slide.

Highlighted in blue is the name of each field. We want to note these so that we know what to expect when we write our callback function. The fields are named as arrays, so they'll be easy to process in our callback. We'll just read which variable is being received, then read which item number in the array to assign the value.

New in the input tags of this form is the value parameter. This parameter controls the default value of each text field. To populate these fields, the "name" dynamic variable is being reused from the first seminar. You'll also notice that a new one called "price" has been added. The callback for this variable has already been written and placed in CustomHTTPApp.c. As a brief note, our vending machine only accepts quarters, so the price for each product is stored as the number of quarters required for purchase. The maximum price is \$5.

POST Example: Callback

- **Handled in `HTTPExecutePost()`**
- **Data stored in TCP buffer**
 - Must manage data retrieval
 - Must track in `curHTTP.byteCount`
- **Procedure is to loop over:**
 - Find data
 - Retrieve data
 - Process data and remove

Now that we've looked at the code for the web page, let's move on to the callback. This time, we'll be modifying `HTTPExecutePost()`. Recall that for this method, the data is stored in the TCP buffer, so we must manage the data retrieval. The value of `curHTTP.byteCount` will indicate how many bytes to expect, and we must track how many of these have been read. This will be our only indication that no more data will be arriving.

The overall procedure will be to look for the next complete field in the buffer, retrieve that field name and its data, then process it and throw it away.

POST: Managing the Buffer

- **HTTP2 uses `curHTTP.byteCount`**
 - Must update whenever bytes are read
- **Locate data with:**
 - `TCPFind()`
 - `TCPFindArray()` and `TCPFindROMArray()`
 - `TCPFindArrayEx()` & `TCPFindROMArrayEx()`
- **Retrieve data with:**
 - `TCPGet()` and `TCPGetArray()`

Before we begin, let's discuss this TCP buffer. Before invoking your callback, the HTTP2 server knows how many bytes will be received as form data. It places this value in the `byteCount` field of the `curHTTP` structure. Your application should reduce this number each time bytes are read so that you know how many more bytes remain. There is no "end-of-data" delimiter, so this is the only indication your application will have to know when to stop asking for more data and consider your task complete.

Since data is not in RAM, we cannot use the `HTTPGetArg()` functions that we used in the prior example. This time, we're dealing directly with the TCP buffer. As such, we will need to use the `TCPFind()` functions to locate data. These functions are described with the rest of the TCP documentation, but in general, they provide great flexibility to find single characters or entire strings, and can limit the scope of the search to be case-insensitive or to only look through specific bytes in the buffer.

Once we've found what we're looking for, the `TCPGet()` functions can be used to retrieve this data from the TCP buffer into our part's main RAM. In doing so, we will free up space in the TCP buffer. This space will eventually be filled with the next packet of data arriving from the web browser, and the process will continue.

HTTPExecutePost()

```
HTTP_IO_RESULT HTTPExecutePost(void)
{
    BYTE name[20], item, *ptr;
    WORD len;

    // Load the file name
    MPFSGetFilename(curHTTP.file, name, 20);

    // Make sure it's the products.htm page
    if(strcmppgm2ram((char*)name,
        (ROM char*)"products.htm") != 0)
        return HTTP_IO_DONE;
```

Let's take a look at an example to explain this. The POST example starts off in the same manner, by first checking which form is being submitted. If an unrecognized form is submitted, we will ignore the data and move on. Otherwise, we continue with the rest of our processing.

HTTPExecutePost()

```
// Loop while data remains
while(curHTTP.byteCount)
{
    // Check for a complete variable
    len = TCPFind(sktHTTP, '&', 0, FALSE);
    if(len == 0xffff)
    {
        // Check if this is the last one
        if(TCPIsGetReady(sktHTTP) ==
            curHTTP.byteCount)
            len = curHTTP.byteCount - 1;
        else // Wait for more data
            return HTTP_IO_NEED_DATA;
    }
}
```

For this form, we will loop over all the variables as long as more bytes are expected. First, we check to see if a complete variable is in the TCP buffer. We know that fields are terminated by the ampersand (&) symbol, so we look for one of those. If we don't find one, then hex "FFFF" will be returned and one of two conditions exist. The last field has no termination, so if the amount of data in the buffer equals the amount of data we still expect, then this is the last field. Otherwise, our TCP buffer has been depleted and we need to wait for more data. In that case we'll return HTTP_IO_NEED_DATA, and the HTTP server will invoke our callback again later when more data is available.

HTTPExecutePost()

```
if(len > HTTP_MAX_DATA_LEN - 2)
{ // Make sure we don't overflow
  curHTTP.byteCount -=
    TCPGetArray(sktHTTP, NULL, len+1);
  continue;
}
len = TCPGetArray(sktHTTP,
                  curHTTP.data, len+1);
curHTTP.byteCount -= len;
curHTTP.data[len] = '\0';
HTTPURLDecode(curHTTP.data);
```

When a variable is found, we will move it into RAM and process it. First, we want to make sure we won't overflow our buffer, so if the variable is too long we remove it and skip to the next one. This should never happen in normal operation, but prevents a skilled attacker from exploiting a buffer overflow flaw.

The next step is to actually read the field into memory. We can use the data array associated with our HTTP connection as temporary storage. Once the data is read, we must remember to update the remaining byte count, terminate the string, and decode the data that was received.

HTTPExecutePost()

```
// Figure out which variable it is
if(memcmp(pgm2ram(curHTTP.data,
                (ROM void*)"name", 4) == 0)
{ // A name was found
    item = curHTTP.data[5] - '0';
    if(item > MAX_PRODUCTS)
        continue;
    memcpy((void*)Products[item].name,
          (void*)&curHTTP.data[8], 10);
}
```

At this point, we have two null-terminated strings in our data array: one name and one value. The next step is to determine what type of variable was received. First, we'll see if it was a product name. If so, then we should determine the item number, and as long as it's valid, copy the new name into that item's name string. We use memcpy() here to copy 10 bytes. If the string is shorter, a null terminator will be copied and the rest will be known to be garbage. If it's longer than 10 characters, this function will truncate the string.

HTTPExecutePost()

```
else if(memcmp(pgm2ram(curHTTP.data,  
    (ROM void*)"price", 5) == 0)  
{// A price was found  
    item = curHTTP.data[6] - '0';  
    if(item > MAX_PRODUCTS)  
        continue;  
    ptr = curHTTP.data + 9;  
    // Skip the $ if entered  
    if(*ptr == '$') ptr++;  
    // Read the dollars value  
    Products[item].price = (*ptr++ - '0') * 4;
```

If we didn't read a name, then we check if a price was encountered. If so, we will load the item number and then check if we need to skip a currency symbol. We will read the next character as the dollar amount and assign it to the price.

HTTPExecutePost()

```
// Read in the cents value
if(strcmppgm2ram((char*)ptr,
    (ROM char*)".87") > 0)
    Products[item].price += 4;
else if(strcmppgm2ram((char*)ptr,
    (ROM char*)".62") > 0)
    Products[item].price += 3;
else if(strcmppgm2ram((char*)ptr,
    (ROM char*)".37") > 0)
    Products[item].price += 2;
else if(strcmppgm2ram((char*)ptr,
    (ROM char*)".12") > 0)
    Products[item].price += 1;
```

We will continue with the remainder of the string to determine the rest of the price. This code implements a simplified rounding function. Since our machines only accept quarters, we will use this code to determine how many extra quarters should be added to the price.

HTTPExecutePost()

```
// Make sure price isn't over max
if(Products[item].price > 20)
    Products[item].price = 20;
}

// Update LCD and AppConfig and return
WriteLCDMenu();
SaveAppConfig();
return HTTP_IO_DONE;
}
```

Finally, we make sure that the price isn't over our \$5 maximum. We are now done processing this variable, and we'll close the loop. As long as data remains in the buffer, this loop will look for the next name/value pair.

Once the loop exits, we know that no more data is expected. At this point, we must rewrite the LCD menu on the screen, and save the configuration to EEPROM. Finally, we return HTTP_IO_DONE to indicate that our callback has completed.

POST Method Demonstration

I'll demonstrate the functionality of this page now.

First, I think everyone would be much happier if we stopped charging for drinks around here. I'll go ahead and set all the prices to zero. I also heard that the Orange Soda, Lemonade, and Iced Tea weren't selling too well, so I'll replace those with a Pilsner, a Stout, and a Brown Ale. Unlikely, but you never know when management might be taking notes from these webinars!

I'll click Save and we'll see my changes get populated to the system. If I click back to the Status page, we'll see that the bar graph has automatically been updated with my new products. My new drink policies are sure to be a hit at the office. And this online vending machine is certainly much easier to manage!

POST Method Summary

- **Data arrives in pieces as request body**
- **Full control of the buffer**
 - Unlimited data length
 - More difficult to process
 - Must manage `curHTTP.byteCount`
- **Handled in `HTTPExecutePost()`**
 - Find, Retrieve, Process, Repeat

You just saw how data is processed when the POST method is used. Data arrives in pieces as space is available, and your application gains full control of the buffer. This method affords you unlimited data length, but at the expense of additional complexity when parsing the input.

Your custom callback function `HTTPExecutePost()` is responsible for parsing data submitted in this method. It will generally follow a pattern of finding the next string of data, retrieving it, and then processing it. This pattern repeats as long as more data is expected, but may have to temporarily return control to the main application when the buffer is depleted.

Web Forms: Summary

- **Allows for input to application**
- **Use GET for short inputs**
 - Data is in `curHTTP.data`
- **Use POST for longer inputs**
 - Data is in TCP buffer

Today's seminar explained how add web-based command and control to a simple embedded application. We demonstrated the POST method, for processing longer inputs. In this case, data is left in the TCP buffer. This makes processing slightly more complex, but makes a much wider range of applications possible.

Conclusion

- **Use forms to add web-based control**
- **Full status and control available online**
- **Second of several TCP/IP seminars**
 - Authentication, E-mail, and more
- **Please provide feedback!**

You now have all the tools necessary to add web-based control to your own application. Complete online status and control systems can be added to any embedded device!

As mentioned at the beginning, this is one of several TCP/IP seminars. From that recent demonstration, it's clear that we need to add some sort of user authentication to prevent people from giving away free drinks. Either that, or we're going to need automatic e-mail updates when stock runs low, because free beer isn't going to last very long. Fortunately, future seminars will cover both of these topics, and many more network-related applications.

Before you go, please remember to provide feedback using the link on this seminar's web page. Comments both good and bad will be appreciated so we can better tailor future seminars to your needs.

For More Information

- **TCPIP Stack User's Guide**
 - Installed with the TCP/IP Stack
 - Accessible via Start menu
- **TCPIP Demo App**
 - *WebPages2*
 - *CustomHTTPApp.c*

www.microchip.com/tcpip

For more information on the topics presented here, please refer to the TCPIP Stack User's Guide. This file is installed with the TCP/IP Stack, and can be accessed via the Start menu.

More examples and labs are available within the TCPIP Demo App that is installed with the stack. The demo application has its own WebPages2 folder and CustomHTTPApp.c source file that explain in more detail how everything works.

Finally, our TCPIP and Ethernet design center online has many more resources to get started with your own project.

Thanks for watching!