Pico W Web Server

Ted Rossin 7/20/2023

Contents

Intro

The Raspberry Pi Pico W added WiFi support to the Pico. The documentation for the Pico W pushes users to micro Python for an example to create a web page on the fly. The C/C++ SDK seems limited to providing some low level Light Weight Internet Protocol (LwIP) functions for moving packets of data around but no info for creating an IOT type web interface. There seem to only be examples of moving packets between a server and client. These LwIP functions and callbacks are tricky to use. Failure to use them correctly results in panics or random hangs. I've put a wrapper around the requirements and placed this code into a class named WebServer to allow web pages to be created on the fly with simple printf type functions and callback functions when buttons are pushed or images are requested. All code and samples can be found here:

sites.google.com/site/tedrossin/home/electronics/raspberry-pi-pico

There is an http library in the pico-sdk/lib/lwip/src/apps/http directory but I was only able to find an example that used it here:

github.com/krzmaz/pico-w-webserver-example

While this example shows how to control the on board LED and read the processor temperature via a web interface, the web page is mostly static and is not created on the fly.

The WebServer class allows pages to be created on the fly by manually sending each line of an html file while embedding device state. I've also supplied HtmlHelper and JavaScript classes which sit on top of the basic WebServer class to handle parsing requests (which button was pushed, what form changed ..) and implementing JavaScript Canvas graphics functions in addition to AJAX (periodic auto info updates that do not redraw the entire page) support.

There are many guides and books on HTML 5 and JavaScript that can be found in stores and on the web. Also, HTML files can be created with a simple editor and saved as learn.html. Then all that is needed is to double click the file and it will be loaded into your default browser. Change the file and hit the reload button on your browser to see how the change affects the display. This makes prototyping pretty fast. Once you are happy with the page, put it in code. Just remember to convert any quotes from " to \" in the strings. Any aspects that are variable can be put into the code when generating strings to be sent to the browser.

WebServer Class Methods

The following are the methods (functions) of the WebServer class that are used to generate a web page. Once a connection is made to the WiFi network via Init, the WebServer class will listen for connections, page requests and file requests (images). These events are placed into a message queue for the application to read and respond to via ReadMessage or the less flexible but easier to use ProcessMessages. In response to a page request, the application can generate the web page using SendText, Printf, SendImage or Write.

Init

```
int Init(const char *WifiSsid,const char *WifiPassword,int RetryForever=1)
```

Call this method first to connect to a desired WiFi network. Init will return 0 if the connection is made and -1 if it was unable to connect. Omitting the last parameter will allow Init to retry until a connection is made. This is useful when recovering from a power failure where it takes the router much longer to power up than the Pico. The default behavior can be overridden by giving the third parameter the value INIT_SINGLE_TIMEOUT. In this case the default timeout of 30 seconds is used.

Example:

```
if(Ws.Init("MyRouter","PicoWrules!",INIT_SINGLE_TIMEOUT)){ printf("Can't connect\n"); while(1); }
```

ResetWifi

```
void ResetWifi(void)
```

Call this method to shut down the WiFi connection and reconnect to the router using the information from the last call to Init. It may take 120 seconds for this function to finish making the connection and return due to the way LwIP handles the binding of the port number.

FetchAddr

```
char *FetchAddr(void)
```

Call this method to display the IP address of the Pico W if Init returned 0.

Example:

```
printf("Connected to %s\n",Ws.FetchAddr())
```

ReadMessage

```
WebMsg_t ReadMessage(void)
```

Read the current message at the tail of the message queue and remove it. The method returns this structure:

```
typedef struct{
    unsigned int Id;
    int Type;
    const char *Data;
```

```
} WebMsg_t;
```

The message Types are:

MSG_EMPTY:

No messages to process so the application is free to do some useful work or sleep. Web page response will be sluggish if too much time elapses before reading the queue again.

MSG_REQUEST:

If the client (browser) requests a page or file, this message will be sent. The Data member of the structure points to the string that was sent to the server. The application needs to send a valid response or the browser will display an error message.

MSG_CLOSED:

When all of the requested data has been sent to the client from the server, this message will be sent to the application. This might be useful for deallocating resources that were needed to handle the request or just to know that the client has all the data.

MSG_CANCELED

If the client closes the connection before all the data has been sent, this message is sent. For simplicity, CLOSED and CANCELED can be treated the same or just ignored.

Example: Handle message requests until there are no more then sleep for 50ms before checking again.

```
while(1){
    do{
        Msg = Ws.ReadMessage();
        switch(Msg.Type){
            case MSG_REQUEST:
                ProcessRequest(Msg.Data);
                break;
            case MSG_CLOSED:
                break;
            case MSG_CANCELED:
                break;
        }
    }while(Msg.Type != MSG_EMPTY);
    sleep_ms(50);
}
```

ProcessMessages

```
void ProcessMessages(unsigned int IdleSleepTime_ms,void (*Cb)(const char *Request));
```

This just puts the loop in the example for ReadMessage into a function that uses a call back to call the ProcessRequest function.

Example: Same as ReadMessage example with less typing.

```
Ws.ProcessMessages(50,ProcessRequest);
```

SendText

```
int SendText(const char *Data);
```

Sends constant data to the client in response to a request message. The string will most likely come from flash memory if the complier is working correctly.

Example: Send and HTML header back to the client

```
Ws.SendText(
```

```
"HTTP/1.1 200 OK\n"
"Content-Type: text/html\n"
"\n"
"<!DOCTYPE html>\n"
"<html>\n"
"<body>\n");
```

Printf

```
int Printf(const char *Format, ...);
```

Same as SendText but allows formatted data to be sent just like using printf. This method needs to copy and process the string so SendText uses less resources if sending non-formatted strings.

Example: Display the visit number and a line number on the web page.

```
Ws.Printf("<br>Visit #%d.  Stay awake %d\n",State.SendCount,i);
```

SendImage

```
int SendImage(const char *Type,int Size,const char *Data);
```

Send a jpeg, gif or other type of image back to the client in response to an image file request. This method will send the header based on the Type specified along with the raw data for the image. See WebImage for an example of how to use this method.

Example:

```
void SendAnvil(void)
{
     Ws.SendImage("gif",ANVIL_SIZE,Anvil);
}
```

Write

```
int Write(const char *Data,int Size);
```

Send raw data to the client. This method is used by SendImage to send the image data but can also be used to send custom data. No header is sent.

Example:

```
Ws.Write(MyData,95758);
```

Examples

The following examples can be used for starting points for projects. All of these examples run on a bare Pico W. No external hardware is required.

- **WebSimple:** A simple example to show how to display text with a portion that is variable
- **WebButton:** Interactive buttons that change color based on the button state, and other buttons that control the on board LED plus a text input box to allow the title of the page to be changed.
- **WebImage:** An example that shows how to make a web page that has images
- **WebGraphics:** A JavaScript example that draws some basic graphic elements
- **WebAjax1:** A simple clock that updates the time faster than twice a second without requiring the page to be reloaded using JavaScript.
- **WebAjax2:** A more complex JavaScript example that displays an historic auto-ranging graph of the processor temperature that is updated every second with the latest temperature.

The details of these examples are explained in the following sections.

WebSimple

WebSimple creates a simple web page with a link to an external web site and 10 lines of text which display the number of visits along with the line number. The source starts out by including some header files and then creating the WebServer class as well as defining the server state as follows:

```
#include <string.h>
#include <stdlib.h>
#include "pico/stdlib.h"
#include "WebServer.h"
#include "WifiInfo.h"  // Defines WIFI_SSID and WIFI_PASSWORD

WebServer Ws;

static struct{
    int SendCount;
} State;
```

The router name and password are defined as strings in WifiInfo.h. This two line file is not included with the examples as it needs to be created based on the network being used.

The main function is defined at the end of the file as follows:

```
int main()
{
    State.SendCount = 0;

    stdio_init_all();
    if(Ws.Init(WIFI_SSID,WIFI_PASSWORD)){
        printf("Failed to connect\n");  while(1);  // Hang
    }
    printf("Connected to %s\n",Ws.FetchAddr());
        // Handle messages from web server. (IdleSleepTime_ms,RequestCallback)
    Ws.ProcessMessages(50,ProcessRequest);
        // Never get here
    return 0;
}
```

The number of visits is set to zero and the USB serial port is set up first. Then the WebServer class is initialized and attempts to connect to the network. If successful, the IP address is printed out to allow a browser to connect. After this, the ProcessMessages method is called which listens for connections and page requests from web browsers (clients). If there are no messages, the method will sleep for 50ms then check again. If there is a page request, the function ProcessRequest is called with a pointer to a string that holds the request. The ProcessMessages method will not return but will just make calls to ProcessRequest as page requests arrive. This function is shown below:

```
void ProcessRequest(const char *Req)
{
    char *Ptr,Info[100];
    int i;

    strncpy(Info,Req,100);
    Ptr = strstr(Info,"\n");  if(Ptr) *Ptr = 0;  else Info[40] = 0;
    printf("    Got Request: %s\n",Info);

    State.SendCount++;
    Ws.SendText(
        "HTTP/1.1 200 OK\n"
        "Content-Type: text/html\n"
        "\n" //  This is needed (for Chrome, Edge and Safari).  I don't know why.
        "<!DOCTYPE html>\n"
        "<html>\n"
        "<body>\n");
    Ws.SendText(
       "<a href=\"https://sites.google.com/site/tedrossin/\">Visit Ted's Site</a><br>");
    for(i=0;i<10;i++){
```

```
            Ws.Printf("<br>Visit #%d.  Stay awake %d\n",State.SendCount,i);
        }
        Ws.SendText("</body>\n"
                    "</html>\n");
    }
```

The top few lines simply print out the first line of the request to stdout for monitoring what is received by the server. This code is not necessary but is useful for learning what browsers send to servers. The visit counter is advanced before the HTML header is sent with Ws.SendText. Next, the link to an external site is sent to the browser followed by 10 lines of text (
 is HTML speak for a new line/line BReak). The function ends by sending the HTML footer and returns.

WebButton

This example uses the HtmlHelper class to create interactive buttons that change color based on the button state, and other buttons that control the on board LED plus a text input box to allow the title of the page to be changed. The HtmlHelper class is also used to parse requests from browsers so that the web server state can be modified and the page displayed based on the new state.

The header file for HtmlHelper needs to be included and the class created as follows:

```
#include "HtmlHelper.h
HtmlHelper Hh;
```

The main function is similar to the WebSimple example except there is more state to initialize and callbacks are set up for the parsing of requests:

```
State.LampMode = MODE_GREEN;
State.Led = LED_OFF;
State.UpdateTitle = 0;
strcpy(State.Title,"Buttons are Fun");

 // Call the function LedButtonPushed if a web request with the Name/Value name set to LED
Hh.SetCallBack("LED",LedButtonPushed);  // Requires Hh.Parse to be called to work.
Hh.SetCallBack("MODE",ColorButtonPushed);
Hh.SetCallBack("TitleInfo",TimeInfoUpdate);
Hh.SetCallBack("Update",UpdateButtonPushed);
```

This example shows how to manually read from the message queue so that more info can be sent to stdout or if other work needs to be done. The result is the same as ProcessMessages technique used in the WebSimple example.

```
while(1){
    do{
        Msg = Ws.ReadMessage();
        switch(Msg.Type){
            case MSG_REQUEST:
                printf("Request received\n");
                ProcessRequest(Msg.Data);
                break;
            case MSG_CLOSED:
                printf("Connection closed\n");
                break;
            case MSG_CANCELED:
                printf("Connection CANCELED\n");
                break;
        }
    }while(Msg.Type != MSG_EMPTY);
    sleep_ms(50);
}
```

The callback functions for the buttons are passed the value for the name/value pair sent from the browser. For example, to control the on board LED:

```
void LedButtonPushed(const char *Value)
{
    if(!strcmp(Value,"OFF")) State.Led = LED_OFF;  else State.Led = LED_ON;
    cyw43_arch_gpio_put(CYW43_WL_GPIO_LED_PIN, State.Led);
}
```

When the two buttons are created for controlling the LED, one is set to OFF and the other is set to ON. These strings are sent to the server by the browser based on which button has been pushed.

The ProcessRequest function starts out with a call to ParseRequest which is as follows:

```
void ParseRequest(const char *Req)
{
    char *Ptr,Info[100];

        // Just for learning what comes back from client (web browser)
    strncpy(Info,Req,100);
    Ptr = strstr(Info,"\n");   if(Ptr) *Ptr = 0;  else Info[50] = 0;
    printf("    Got Request: %s\n",Info);

    State.UpdateTitle = 0;
    Hh.Parse(Req);
    if(State.UpdateTitle){
        strncpy(State.Title,TmpTitle,TITLE_SIZE);
    }
}
```

Again, the first few lines are just to see what happens when a button is pushed or text is modified. After this, the state variable UpdateTitle is set to 0. If the title is changed (via the Update Title button), this variable will be set to one in a callback function. After this, the Hh.Parse method is called to parse the string sent by the browser. This method looks for LED, MODE, TitleInfo or Update. For each one found, the corresponding callback that was defined with `Hh.SetCallBack` is called. If the state variable UpdateTitle is set to 1, the new title is copied to the state variable so that when the page data is returned, the new title can be used. The function then returns control back to ProcessRequest to generate and return the page information to the browser.

```
void ProcessRequest(const char *Req)
{
    ParseRequest(Req);

    Hh.SendHeader();
    Ws.Printf("<h1>%s</h1>\n",State.Title);
    Ws.SendText("<h2>Push the Buttons</h2>\n");

    if(State.LampMode==MODE_RED)
        Hh.CreateButton("MODE=RED","Red",0xff,0x1c,0x10);
    else
        Hh.CreateButton("MODE=RED","Red");
    if(State.LampMode==MODE_GREEN)
        Hh.CreateButton("MODE=GREEN","Green",0x4c,0xaf,0x50);
    else
        Hh.CreateButton("MODE=GREEN","Green");
    if(State.LampMode==MODE_BLUE)
        Hh.CreateButton("MODE=BLUE","Blue",0x4c,0x50,0xaf);
    else
        Hh.CreateButton("MODE=BLUE","Blue");
```

Hh.SendHeader will send the same header that WebSimple sent with Ws.SendText. After that, the page title is created with Ws.Printf using the state variable Title followed by header text to tell the user to push the buttons. Based on the state variable LampMode, the code will generate three buttons where the button that corresponds to the LampMode will be colored that color and the other two will be the default color. This way when the button with the text Green is pushed, the browser will send the string MODE=GREEN to the server when requesting the page and the server will parse this string and set the LampMode to MODE_GREEN and then deliver the page back with only the button labeled Green painted green and the other two the default color of the browser.

The LED On and LED Off buttons are handled the same way except that the LED callback function will set the on-board LED to the value of the Led state variable. The code to create the text box to change the title and close out the page is below:

```
        Ws.SendText("<h2>Change the Title</h2>\n");
        Ws.SendText("<form action=\"/action_page.php\">\n");
                // (Characters,PixelWidth,Value,Name,Label)
            Hh.CreateWidthInputField(TITLE_SIZE,700,State.Title,"TitleInfo","The Title");
            Ws.SendText("<br>\n");
            Hh.CreateSubmitButton("Update","1","Update Title");
        Ws.SendText("</form>\n");

        Hh.SendFooter();
    }
```

This is just some HTML 5 code to create a form using some helper methods to make a text box and submit button. The current page title is displayed in the text box so that it is easy to modify if there is just a simple change. The page is completed with the helper method SendFooter.

WebImage

The WebImage example demonstrates how to add images to a web page using precompiled images. When a web page is returned to the web browser that includes images (), the images names are saved by the browser and once the entire page is received, the web browser will ask for each image with a separate request to the server. These image requests differ from normal HTML requests in that they will have a line (after the first line with the GET) that has "Accept: image/" in the request. The HtmlHelper method Parse looks for this string and will then look to see if any image callbacks have been defined. If it finds a match, it will call the callback function. If not, it will send a not found message back to the client. Either way, if the "Accept: image/" string is found, the Parse method will return 1 otherwise it will return 0. This result is then used by the application to determine if the page should be sent to the web browser or the image with its header has already been sent via the callback or the not found message has been sent.

In the directory for this example is a Tools directory which has a C program and a Python program to convert an image file to a header file that can be included into the application. In this example, three images are included. Anvil and SpinTed2 are both animated GIFs while Veg is just a static JPEG image of vegetables. The first few bytes of Anvil.h are as follows:

```
#define ANVIL_SIZE    14223
const char Anvil[] = {
     0x47,0x49,0x46,0x38,0x39
```

The ANVIL_SIZE is used to tell the WebServer SendImage method how many bytes to send. The const qualifier lets the compiler know to place the image array into flash memory (ROM) which there is more of than RAM. The WebImage.cpp file starts out the same as the WebButton example except the three images are included:

```
#include "Anvil.h"
#include "SpinTed2.h"
#include "Veg.h"
```

The main function is also similar to the WebButton example except it uses the SetImageCallBack of HtmlHelper to set the callbacks for when the images are requested. Three buttons are created in the example to select between the three images so a callback is registered for ImageButtonPushed. The SetImageCallBack calls register functions to be called when the web browser requests any of the three images as follows:

```
        Hh.SetCallBack("IMAGE",ImageButtonPushed);

        Hh.SetImageCallBack("anvil.gif",SendAnvil);
        Hh.SetImageCallBack("SpinTed2.gif",SendSpinTed);
        Hh.SetImageCallBack("Veg.jpg",SendVeg);
```

The SendAnvil function is shown below which makes a call to the WebServer method SendImage telling the method the type of image, the size and where to fetch the data from.

```
        void SendAnvil(void)
        {
             Ws.SendImage("gif",ANVIL_SIZE,Anvil);
        }
```

The SendImage method will send an image header with the type GIF then send ANVIL_SIZE bytes pointed to by Anvil.

The ProcessRequest function starts out a little different from the previous examples in that it checks the return value from ParseRequest and simply returns if it is non-zero.

```
int ParseRequest(char *Req)
{
    printf("    Got Request: %s\n",Req);

    return(Hh.Parse(Req));  // Returns 1 if no more data needs to be returned.
}

void ProcessRequest(char *Req)
{
    if(ParseRequest(Req)) return;  // Request has been satisfied via callbacks (images sent)

    Hh.SendHeader();

        …  Code to display 3 buttons to select which image to display

    switch(State.ImageId){
            case IMAGE_ANVIL:
                Ws.SendText("<img src=\"anvil.gif\" alt=\"Anvil\"/>\n");
                break;
            case IMAGE_SPIN_TED2:
                Ws.SendText("<img src=\"SpinTed2.gif\" alt=\"SpinTed2\"/>\n");
                break;
            case IMAGE_VEG:
                Ws.SendText("<img src=\"Veg.jpg\" alt=\"Veg\"/>\n");
                break;
    }
    Ws.SendText("<br>Enjoy!<br>\n");

    Hh.SendFooter();
}
```

A non-zero return from the HtmlHelper method Parse means that the response to the request has been sent so no further work is needed. The ParseRequest function prints the request from the web browser to stdout as well. If ParseRequest returns zero, the normal page is displayed starting with the HTML header then the three buttons to select between the images followed by the desired image. This is selected using the state variable ImageId which is set in the button callback function named ImageButtonPushed. Note that the filename listed for the src= part of the image must match the name given to the HtmlHelper method SetImageCallBack.

A little message is displayed under the image followed by the HTML footer. Similar to the WebButton example, the image selection button that is the currently selected image is colored red while the other two buttons are left their default color.

WebGraphics

This example adds the JavaScript class to draw rectangles, lines and text as well as using the HtmlHelper class to create interactive buttons that change color of the graphics display based on the button state. Similar to the WebButton example, two buttons are added that control the on board LED plus a text input box to allow the title of the graphics display to be changed. The JavaScript class is used to create a canvas element and then use JavaScript to draw on the canvas.

The header file for JavaScript needs to be included and the class created as follows:

```
#include "JavaScript.h
JavaScript Js;
```

The main function is similar to the WebButton example except ProcessMessages is used instead of reading the messages and the waveform to be displayed is initialized with the following:

```
for(i=0;i<WAVE_SIZE;i++) State.Wave[i] = i*((i&1)?1:-1);
```

The callback functions for the buttons are identical to WebButton as is the ParseRequest function except that this example does not display the browser request. The main difference in ProcessRequest compared to WebButton is the JavaScript code to draw the graphics and text. The JavaScript code starts out as follows:

```
Js.InitCanvas(&Ws,600,400);
    Js.FillStyle(0xff,0x00,0x00);  // Red
    Js.FillRect(0,0,600,300);
    Js.FillStyle(0x00,0xf0,0xc0);  // Green/Blue
    Js.Font(80,"Arial");
    Js.FillText(10,80,State.Title);
```

The dimensions of the drawing area (canvas) are set with the JavaScript class method InitCanvas which sends a pointer to the WebServer class as well as the width and height of the canvas. If no drawing area is needed, just call Init(&Ws). The next two lines draw a red 600x300 pixel rectangle at location 0,0 (top left of the canvas). The three lines after that draw the title in the color cyan using an 80 pixel high Arial font. The Title text is a state variable which can be modified using the input box just like the WebButton example did.

The next lines of code draw the expanding sawtooth waveform using lines:

```
Js.BeginPath();
    Js.LineWidth(2);
    Js.MoveTo(10,100+128-State.Wave[0]);
    for(i=1;i<WAVE_SIZE;i++){
        Js.LineTo(20+16*i,100+128-State.Wave[i]);
    }
Js.EndPath();
```

Lines have to be drawn using a path so the BeginPath method call starts a path while the last line above (EndPath) ends the path and tells the browser to update the display. The code between the BeginPath and EndPath sets the width of the lines and then sets a starting point using MoveTo then loops setting lines between the previous point and the current point using LineTo.

The JavaScript code finishes up by drawing a 10 pixel high gray line under the red rectangle as well as 90 pixel high rectangle under the gray line. This last rectangle is set to the color of the corresponding color button that was last pushed (red, green or blue). The script is ended with a call to EndScript.

```
Js.FillStyle(0x80,0x80,0x80);
Js.FillRect(0,300,600,10);
if(State.LampMode==MODE_RED)          Js.FillStyle(0xff,0x00,0x00);  // Red
else if(State.LampMode==MODE_GREEN)   Js.FillStyle(0x00,0xff,0x00);  // Green
else                                  Js.FillStyle(0x00,0x00,0xff);  // Blue
Js.FillRect(0,310,600,90);
Js.EndScript();
```

The end effect is that if a color button is pushed, the page is reloaded with the rectangle on the bottom of the graphics display shaded the color of the button. If the title text box is changed, the page is reloaded with the text in the graphics display changed to the value in the text box. The JavaScript class is not needed to create JavaScript web pages as the JavaScript text can be sent with the WebServer class method SendText. The JavaScript class just tries to simplify the code and help prevent syntax errors which browsers handle by just killing off the script with no explanation as to what is wrong. The Development tools in most browsers may help.

The WebAjax1 example shows how to periodically update the graphics display with the current time without reloading the page. The WebAjax2 example shows how to animate the graphics display with temperature data from the on die temperature sensor to create a scrolling display of the temperature.

WebAjax1

This example creates a simple clock that updates the time faster than twice a second without requiring the page to be reloaded using JavaScript and AJAX (Asynchronous JavaScript And XML). This example also shows how to create multiple pages and how to switch between them in response to button presses. The Pico real-time clock is used to keep track of the time and a timer interrupt is used to flash an external LED if connected to GPIO 16. The LED is not required to make the project work but was useful for detecting panics when developing the WebSever class.

The main function is similar to the previous examples except that it initializes the real-time clock and, sets up the GPIO and starts up a repeating timer that fires every ½ second:

```
add_repeating_timer_ms(500,TimerHandler,NULL,&TimerInfo);
```

The clock is started with SDK calls to `rtc_init` and `rtc_set_datetime` with a default date and time in the function InitClock. Callbacks are setup for the buttons and the input field which sets the time:

```
    Hh.SetCallBack("MODE",ColorButtonPushed);
    Hh.SetCallBack("UPDATE_TIME",SetClockButtonPushed);
    Hh.SetCallBack("TimeInfo",TimeInfoUpdate);
    Hh.SetCallBack("Update",UpdateButtonPushed);
```

The SetClockButtonPushed function will change the state variable PageId so that the clock set page will be displayed instead of the clock page. The ProcessRequest function is different from the previous examples in that it will display one of two pages based on the PageId. The function is as follows:

```
void ProcessRequest(char *Req)
{
    datetime_t t;

    if(ParseRequest(Req)==UPDATE_TIME_DISPLAY){  // Just update time display
        Ws.SendText("Access-Control-Allow-Origin: *\n");  // Needed to get iphone browser to get a
non-zero status
        rtc_get_datetime(&t);
        Ws.Printf("Data:%02d:%02d:%02d\n",t.hour,t.min,t.sec);   // Send time back to client
        return;
    }

    switch(State.PageId){
        case MAIN_PAGE:
            DisplayMainPage();
            break;
        case CLOCK_SET_PAGE:
            DisplayClockSetPage();
            break;
    }
}
```

ParseRequest is called to determine what the browser requested. The main page will use JavaScript to cause the browser to send a request to the Pico (Server) every 435ms for the current time. If one of these requests is detected in ParseRequest, it will return UPDATE_DISPLAY_TIME. If the request is a button press or just a page reload, ParseRequest will return UPDATE_FULL_PAGE. The ProcessRequest function will return to the web browser a magic string followed by Data: and the current time read from the real-time clock if ParseRequest returned UPDATE_TIME_DISPLAY. Otherwise, it will call either DisplayMainPage or DisplayClockSetPage based on the PageId state variable.

The start of ParseRequest code is the same as some other examples that print to stdout the first line of what is received by the browser. The next few lines detect if the request is from the periodic JavaScript timer which will sent a request with “TimeUpdate”.

If so, it will return a code that will not resend the entire page but will only send a few strings back to the browser so that it can just update the time display. The code to cause this to occur is explained in the description of the function DisplayMainPage:

```
if(!strncmp(Req,"GET /TimeUpdate",15)){
    return(UPDATE_TIME_DISPLAY);
}
```

The remainder of the code for ParseRequest checks to see if the text box has been updated and if so, parses the text and sets the time on the real-time clock if the time entered is a valid time.

```
State.UpdateTime = 0; // Callback function will change this if Time is updated
Hh.Parse(Req);

if(State.UpdateTime){
    rtc_get_datetime(&t);
    Hour = -1;  Min = -1;
    sscanf(TmpTime,"%02d:%02d",&Hour,&Min);
    if(Hour>=0 && Hour<=23 && Min>=0 && Min<=59){
        t.hour = Hour;  t.min = Min; t.sec = 0;
        rtc_set_datetime(&t);
    }
}
return(UPDATE_FULL_PAGE);
```

The call to the HtmlHelper method Parse will make calls to the button and text field callbacks. If the text field update button was pushed, the code will try to update the time by first fetching the current time and then updating just the hour, minute and zeroing the seconds field before setting the clock if the hour and minute are in range. The code will then return a value which will cause the entire page to be reloaded.

The DisplayClockSetPage function is as follows:

```
void DisplayClockSetPage(void)
{
    char s[100];
    datetime_t t;

    Hh.SendHeader();
    Ws.SendText("<h1>Set the Clock:</h1>\n");
    Ws.SendText("<form action=\"/action_page.php\">\n");
        rtc_get_datetime(&t);
        sprintf(s,"%02d:%02d",t.hour,t.min);
            // (Characters,PixelWidth,Value,Name,Label)
        Hh.CreateWidthInputField(TIME_SIZE,150,s,"TimeInfo","<br>24 hour hh:mm format");
        Ws.SendText("<br>\n");
        Hh.CreateSubmitButton("Update","1","Update Time");
        Hh.CreateSubmitButton("Update","0","Cancel");
    Ws.SendText("</form>\n");
    Hh.SendFooter();
}
```

The code simply creates a page with a title that says “Set the Clock” and a form that has a text box which is initialized to the current time and two submit buttons. One updates the time with the value entered and the other discards the changes. Both buttons will cause the browser to request a page redraw and send a request with the string Update and the value set to 1 if the change should be made and set to 0 if it should be ignored. When the request is parsed, the Pico (server) will call the UpdateButtonPushed function which will set the PageId state variable back to MAIN_PAGE so that the main page is sent back to the web browser instead of the Clock Set page.

The last piece of code is the DisplayMainPage function. This starts out just like the WebGraphics example in that it creates three buttons (Red, Green and Blue) followed by code to create a JavaScript Canvas to display the time. The JavaScript code starts out as follows by creating the canvas to draw on plus creating a web browser timer that will call the JavaScript method ScriptTimerCallback every 0.435 seconds.

```
Js.InitCanvas(&Ws,SCREEN_WIDTH,SCREEN_HEIGHT);
            // Set up call back (a little less than a 1/2 second per callback)
    Js.CreateTimerCallback("ScriptTimerCallback",435);  // Units are milliseconds
    Js.Font(200,"Arial");

    Ws.SendText("function ScriptTimerCallback()\n{\n");
        Js.RequestUpdate("ProcessReqChange","TimeUpdate");
    Ws.SendText("}\n");
```

The ScriptTimerCallback function will send a request to the Pico (server) from the web browser (client) to fetch the current time of the server by sending it a request with the string “TimeUpdate”. This request will be sent back to the web browser and handled by the JavaScript function named ProcessReqChange. This is the AJAX code. So that the display is not left blank until the first timer expires, a call the RequestUpdate is made to force an update when the page is first loaded as follows:

```
        // Draw first image before timer expires so there is no wait for the screen to update.
    Js.RequestUpdate("ProcessReqChange","TimeUpdate");
```

The remainder of the JavaScript is the special ProcessReqChange function. This function has some validation code at the start to make sure it is the AJAX request that has been returned so the function is declared with the RequestChangeHeader method of the JavaScript class. The function ends with a call to the RequestChangeFooter method of the JavaScript class. The entire JavaScript function to display the time is shown below:

```
            // Java script handler for AJAX updates
        Js.RequestChangeHeader("ProcessReqChange");
            Js.FillStyle(0x30,0x40,0x20);
            Js.FillRect(0,0,SCREEN_WIDTH,SCREEN_HEIGHT);
            if(State.LampMode==MODE_RED){
                Js.FillStyle(0xff,0x00,0x30);
            }
            else if(State.LampMode==MODE_GREEN){
                Js.FillStyle(0x00,0xff,0x30);
            }
            else{
                Js.FillStyle(0x00,0x30,0xff);
            }
                // Remove Access control message that is required for iphone browser
            Ws.SendText("var DataIndex = gReq.responseText.indexOf(\"Data:\");\n");
            Ws.SendText("if(DataIndex>=0){\n");
                Ws.SendText("var TimeInfo = gReq.responseText.slice(DataIndex+5);\n");
                Ws.SendText("ctx.fillText(TimeInfo,10,170);\n");
            Ws.SendText("} else { ctx.fillText(\"Bummer\",10,150);}\n");
        Js.RequestChangeFooter();
    Js.EndScript();
```

The code first draws dark background then selects a text color based on the current state of the color buttons indicated by the LampMode state variable. Then, the global variable that was defined in the InitCanvas call named gReq and is also used by the code generated for RequestUpdate is used to fetch the string sent by the Pico (server) to get the current time as a string. Since some extra characters are sent with the update to make Safari happy, the JavaScript code will look for the marker “Data:” and use that to get the string after that marker and load it into a variable named TimeInfo. If “Data:” can be found, the time is displayed using the global variable ctx that was defined in the InitCanvas call to display the time. Otherwise, the word Bummer is displayed. The script is competed with the EndScript call.

Keep in mind that this ProcessReqChange function stays static after the entire page is loaded. The only way to have it display or do something different without the page being reloaded is to send it a message and parse the message to determine what to do. For example, instead of having the time just update in a fixed color after the page has been loaded, the code could parse the time and change the color based on the hour of the day. It could display the time in a gray color at night and white during the daylight hours without the page needing to be reloaded. Also, the string could include more info after the time. For example, it could send a 3 or 4 character temperature and a 3 character humidity from a temperature/humidity sensor hooked up to the Pico. The JavaScript code could split these strings out and have all this info displayed every 435ms.

The DisplayMainPage function finishes off by adding a button to change the PageId state variable to display the CLOCK_SET_PAGE and then sends the footer.

WebAjax2

This example creates a more complex JavaScript example that displays an historic auto-ranging graph of the processor temperature that is updated every second with the latest temperature. This example is similar to WebAjax1 in that it uses AJAX to periodically update the web page without requiring the page to be manually reloaded. The main function is similar as well except that it initializes the analog to digital converter in the Pico to read the temperature of the Silicon as follows:

```
adc_init();
adc_set_temp_sensor_enabled(true);
adc_select_input(4);
```

The temperature history that will be display is also initialized to something near room temperature and a timer is set up to read the temperature every second as follows:

```
for(i=0;i<HIST_SIZE;i++) State.HistoryC[i] = 20.0f;
add_repeating_timer_ms(1000,TimerHandler,NULL,&TimerInfo);
TimerHandler(NULL);  // Call once to initialize some numbers
```

Only two buttons are used in this example. One button to turn on the LED and one button to turn it off. These same buttons are used to change the color of the display. The TimerHandler function is called once a second from a Pico timer interrupt. This function will read the temperature sensor voltage, convert it to Centigrade and then shift the HistoryC data and add the current temperature to the end of the list (most current temperature). While doing this, the code determines the minimum and maximum temperature for the auto ranging math (simple Y=mX+b stuff). The HistoryC array is then converted to screen coordinates and stored in an array named History and the legend start and step rate are calculated. This information will be sent to the web browser using AJAX every second so that the waveform displayed will cover the entire canvas and the scale will change with the minimum and maximum of the data. The code for TimerHandler is shown below:

```
bool TimerHandler(repeating_timer_t *Info)
{
    int i;
    float Delta,Min = 1000.0f,Max = -1000.0f;
    int Top=SCREEN_HEIGHT-20,Bottom=40;

    float voltage = adc_read() * 3.3f / (1 << 12);
    State.CurrentTempC = 27.0f - (voltage - 0.706f) / 0.001721f;
    for(i=0;i<HIST_SIZE-1;i++){
        State.HistoryC[i] = State.HistoryC[i+1];
        if(State.HistoryC[i]>Max) Max = State.HistoryC[i];
        if(State.HistoryC[i]<Min) Min = State.HistoryC[i];
    }
    State.HistoryC[HIST_SIZE-1] = State.CurrentTempC;
    if(State.CurrentTempC>Max) Max = State.CurrentTempC;
    if(State.CurrentTempC<Min) Min = State.CurrentTempC;

    Delta = Max-Min;
    if(Delta < 1.0f){ Max=Min+1.0f;  Delta = 1.0f; }
    State.GainY = (float)(Top-Bottom)/(Max-Min);
    State.OffsetY = (float)Bottom - State.GainY*Min;

    for(i=0;i<HIST_SIZE;i++){
        State.History[i] = (int16_t)(State.GainY*State.HistoryC[i] + State.OffsetY);
    }

    State.LegendStepY = Delta/(float)(NUM_LEGEND_LINES-1);
    State.LegendMin = Min;

    return(true);
}
```

The voltage calculation is simply converting the 12 bit value from the analog to digital converter to a value between the supply voltage (3.3V) and ground (0V).

The ProcessRequest function is similar to WebAjax1 but sends a more complex string in response to the update request. Also, the request name is GraphUpdate instead of TimeUpdate. The string returned is described in the comments of the code. The code to push all the data to draw the graph is shown below:

```
if(ParseRequest(Req)==UPDATE_GRAPH){
        // Send the following info as though it was in a structure as a string to the browser
        // to just update the graph.
        //  16 bit integers are converted to 4 digit hex values
        //  strings are sent as is but without terminating '\0'
        //   struct{
        //      char CurrentTemp[6];
        //      char YLegendText[NUM_LEGEND_LINES][6];
        //      int16_t YLegendValues[NUM_LEGEND_LINES];
        //      int16_t GraphData[HIST_SIZE];
        //  }
    Ws.SendText("Access-Control-Allow-Origin: *\n");  // get iphone browser to get a non-zero status
    strcpy(s,"Data:");  Offset = 5;
    sprintf(s,"Data:%5.1fC",State.CurrentTempC);  Offset+=6;  // space for negative sign
    for(f=State.LegendMin,i=0;i<NUM_LEGEND_LINES;i++){
        sprintf(&s[Offset],"%5.1fC",f);  Offset+=6; f+=State.LegendStepY;
    }
    for(f=State.LegendMin,i=0;i<NUM_LEGEND_LINES;i++){
        sprintf(&s[Offset],"%04x",(int16_t)(State.GainY*f + State.OffsetY));
        Offset+=4; f+=State.LegendStepY;
    }
    for(i=0;i<HIST_SIZE;i++){
        sprintf(&s[i*4+Offset],"%04x",State.History[i]&0xffff);
    }
    s[4*HIST_SIZE+Offset] = 0;
    Ws.SendText(s);
    return;
}
```

The entire string is terminated with a '\0'. The `strcpy(s,"Data:");` is wasted code as the next line writes the same data. The code for the JavaScript portion of the web page is similar in form to the WebAjax1 example except that there is a call to the JavaScript class member CreateXferFuncs.

```
Js.InitCanvas(&Ws,SCREEN_WIDTH,SCREEN_HEIGHT);
    Js.CreateXferFuncs(JS_XFER_INT16 | JS_XFER_STRING);

        // Set up call back (every 1 second)
    Js.CreateTimerCallback("ScriptTimerCallback",1000);
    Js.Font(20,"Arial");

    Ws.SendText("function ScriptTimerCallback()\n{\n");
        Js.RequestUpdate("HandleReqChange","GraphUpdate");
    Ws.SendText("}\n");
    Js.RequestUpdate("HandleReqChange","GraphUpdate");  // Send one on first page load
```

The call to CreateXferFuncs will create little functions that are sent to the web browser that can be used to extract arrays of 16 bit signed integers (JS_XFER_INT16) and strings (JS_XFER_STRING). Other functions can be added by looking at the JavaScript.h file. There are 8,16 and 32 bit signed and unsigned functions in addition to the string function.

The HandleReqChange code starts out by drawing the background based on the NightDisplay state variable then finding the start of the graph data. If the start is found, the code defines some variables and arrays and extracts the graph data using the functions that were created with the call to CreateXferFuncs. For the functions, the first parameter is the string to extract data from. The slice method extracts an offset into the string. The second parameter is the number of items to extract. The last parameter is the array to store the results. If the function is FetchArray_string, the size of each string is specified with the 3rd parameter. The functions return the number of characters used to extract the required data.

```
    Js.RequestChangeHeader("HandleReqChange");
        if(State.NightDisplay) Js.FillStyle(0x00,0x00,0x00); else Js.FillStyle(0x00,0x00,0xff);
        Js.FillRect(0,0,SCREEN_WIDTH,SCREEN_HEIGHT);
```

```
            // Remove Access control message that is required for iphone browser by searching for Data:
        Ws.SendText("var DataIndex = gReq.responseText.indexOf(\"Data:\");\n");
        Ws.SendText("if(DataIndex>=0){\n");
                // Parse the string to get the graph data
            Ws.SendText("var Loop,gOffset=0,Data = [];\n");
            Ws.SendText("var gCleanInfo = gReq.responseText.slice(DataIndex+5);\n");
            Ws.SendText("var Temperature = [];\n");
            Ws.SendText("var Legend=[],LegendY=[]\n");
            Ws.SendText("gOffset += FetchArray_string(gCleanInfo.slice(gOffset),1,6,Temperature);\n");
            Ws.Printf("gOffset += FetchArray_string(gCleanInfo.slice(gOffset),%d,6,Legend);\n"
                        ,NUM_LEGEND_LINES);
            Ws.Printf("gOffset += FetchArray_int16(gCleanInfo.slice(gOffset),%d,LegendY);\n"
                        ,NUM_LEGEND_LINES);
            Ws.Printf("gOffset += FetchArray_int16(gCleanInfo.slice(gOffset),%d,Data);\n",HIST_SIZE);
```

At this point the graph data is loaded into JavaScript arrays as follows.

Temperature[0] holds the current temperature as a 6 character string
Legend holds an array of text values for the Vertical legend where each value is a 6 character string
LegendY holds an array of signed integers that are the Y coordinates of the horizontal lines for the legend
Data holds an array of signed integers that are the Y coordinates of the temperature data to display.

The following code uses those variables to display the current temperature and a draw the legend text and horizontal lines for the graph (ctx is the Canvas context which was created in the InitCanvas method of the JavaScript class):

```
            if(State.NightDisplay) Js.FillStyle(0xff,0xff,0xff); else Js.FillStyle(0xf0,0xff,0x00);
            Ws.Printf("ctx.fillText(\"Current Temperature: \" + Temperature[0],%d,%d);\n"
                        ,SCREEN_WIDTH/2,SCREEN_HEIGHT-5);
                // Display Y legend
            Ws.Printf("for(Loop=0;Loop<%d;Loop++){\n",NUM_LEGEND_LINES);
                Ws.Printf("ctx.fillText(Legend[Loop],%d,%d-LegendY[Loop]);\n"
                        ,LEGEND_LEFT_MARGIN,SCREEN_HEIGHT);
            Ws.SendText("}\n");
                    // Draw Y legend lines
            Js.LineWidth(1);
            if(State.NightDisplay) Js.LineColor(0xff,0xff,0xff); else Js.LineColor(0xf0,0xff,0x00);
            Js.BeginPath();
                Ws.Printf("for(Loop=0;Loop<%d;Loop++){\n",NUM_LEGEND_LINES);
                    Ws.Printf("ctx.moveTo(%d,%d-LegendY[Loop]);\n",GRAPH_LEFT_MARGIN,SCREEN_HEIGHT);
                    Ws.Printf("ctx.lineTo(%d,%d-LegendY[Loop]);\n",GRAPH_RIGHT,SCREEN_HEIGHT);
                Ws.SendText("}\n");
            Js.EndPath();
```

The last bit of code for the JavaScript portion of the web page is to draw the temperature data and end the script in a manner similar to the WebGraphics example.

```
                    // Draw temperaure data
            Js.LineWidth(2);
            if(State.NightDisplay) Js.LineColor(0x00,0xff,0x00); else Js.LineColor(0xff,0x80,0x00);
            Js.BeginPath();
                Ws.Printf("ctx.moveTo(%d,%d-Data[0]);\n",GRAPH_LEFT_MARGIN,SCREEN_HEIGHT);
                Ws.Printf("for(Loop=1;Loop<%d;Loop++){\n",HIST_SIZE);
                    Ws.Printf("ctx.lineTo(Loop*%d+%d,%d-Data[Loop]);\n"
                        ,SAMPLE_STEP,GRAPH_LEFT_MARGIN,SCREEN_HEIGHT);
                Ws.SendText("}\n");
            Js.EndPath();
        Ws.SendText("} else { ctx.fillText(\"Bummer\",10,150);}\n");
    Js.RequestChangeFooter();
Js.EndScript();
```

The rest of the code for the ProcessRequest function does the same as the WebGraphics example and draws two buttons to control the on board LED. These buttons also control the color of the graph by modifying the NightDisplay state variable.