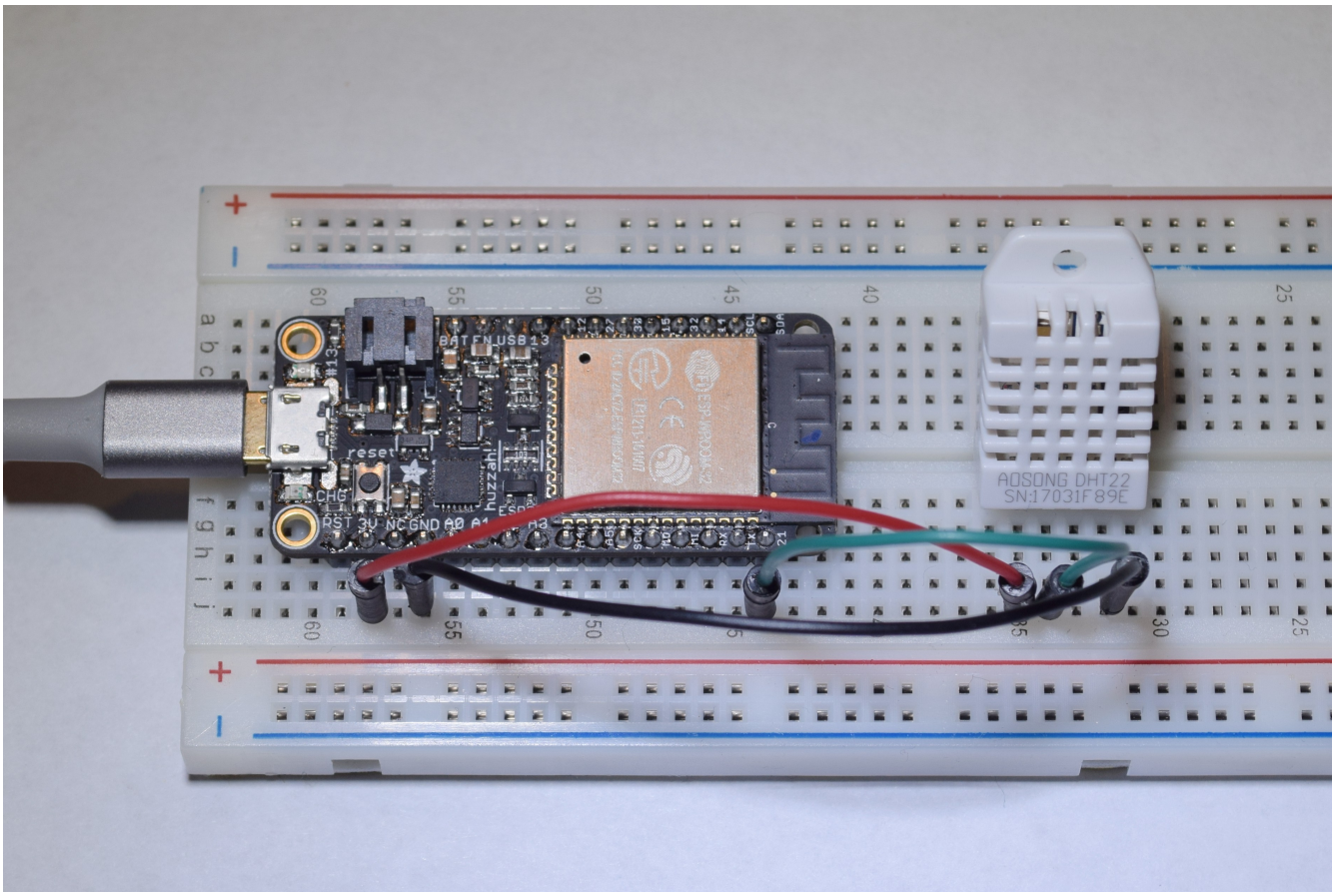
Mongoose-OS ESP32 Temperature Sensing with Websocket and D3.js Display



Gregory Raven

November 2, 2017

Contents

Chapter 1

Introduction

This is the documentation for a project based on the Mongoose-OS operating system. The development board contains an ESP32 system-on-chip. A DHT22 temperature-humidity sensor chip is wired to the ESP32 board.

Assembly of the project can be done on an inexpensive breadboard. Most of the ESP32 development boards should work, however, some of them have wide pin-spacings which do not work well with the common breadboard. The latest board from Adafruit, the "Feather" board is the narrowest seen yet, allowing two rows of pins on one side and one side on the other. This board also has capability for battery power which is a nice feature.

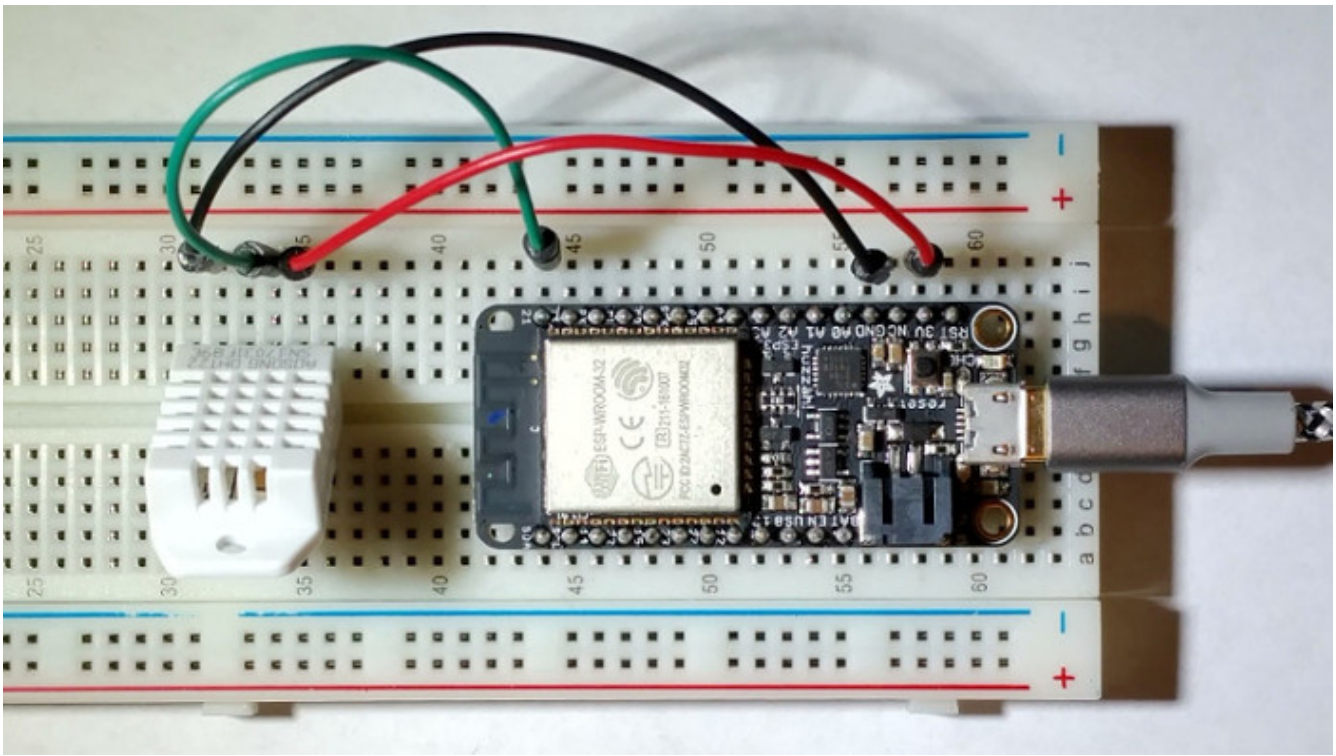


Figure 1.1: Adafruit HUZZAH32 – ESP32 Feather Board

The system is an HTTP5 web server. The server provides a web page which then connects to the server via Websocket. The browser must be Websocket capable. After a Websocket connection is established, the server begins publishing JSON data. The data is received by the web page via embedded Javascript code. D3.js visualization is used to plot the incoming temperature data versus time. An example plot is shown below.

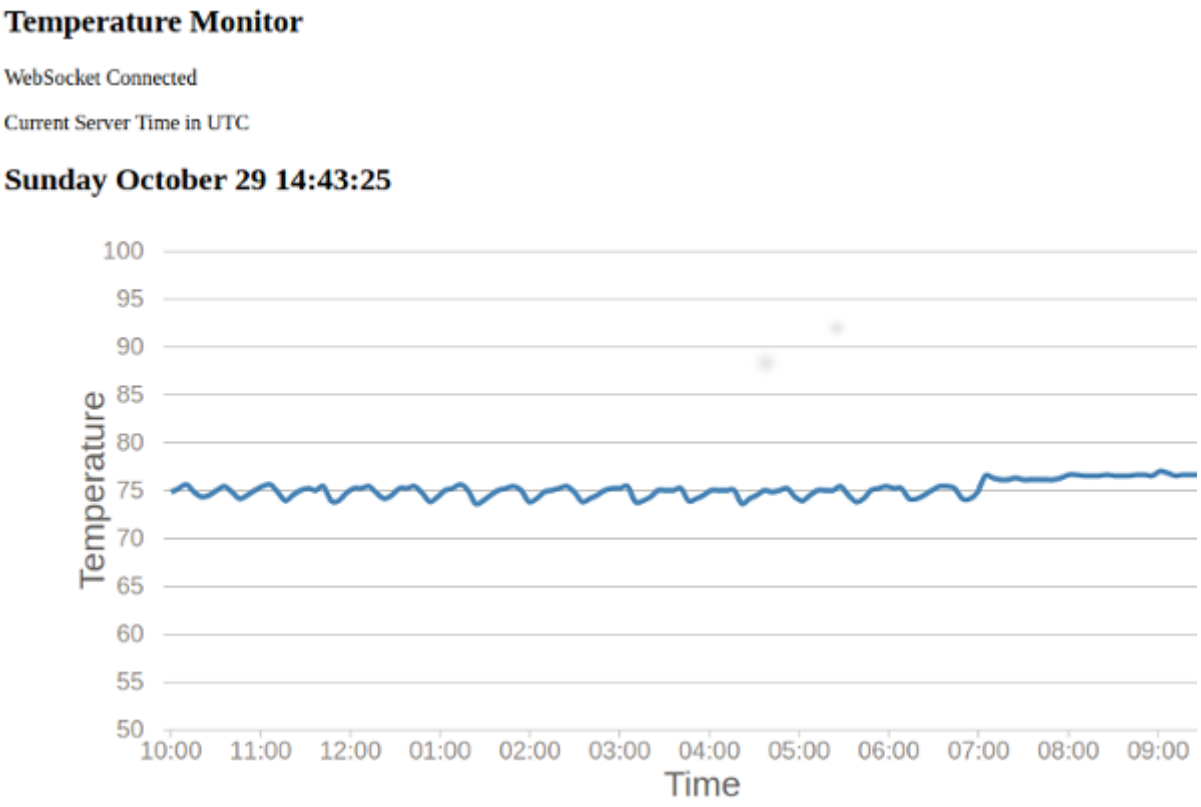


Figure 1.2: D3.js Plot of Temperature Data from the ESP32

A note of caution on this project. None of the available security features are enabled. This project should only be deployed in a local network behind a firewall. It should not be “internet facing”.

Wiring

Only three wires from the ESP32 Feather to the DHT22 are required per this table:

Huzzah Feather Pin	DHT22 Pin
3V	1, VDD power supply
21 (GPIO)	2, DATA
GND	4, GROUND

Chapter 2

Mongoose-OS Code on the ESP32

This chapter describes the Mongoose-OS code used with the ESP32.

The code is all done in C. A Javascript version may be published in the future. C was chosen as the author was interested in learning embedded C programming. There is substantial Javascript code in the web page which receives and displays data.

Using the built-in capability of Mongoose-OS, the code is surprisingly simple. The libraries and facilities provided make the web server code brief; perhaps even simpler than using a high-level framework like Node.js.

Creating an "Empty App"

Install the mos tool per the instructions:

https://mongoose-os.com/software.html

The author prefers operating at the command line in Linux. However, everything described should be possible using the mos GUI, which is launched as follows:

```
mos ui
```

Create the "empty app" as follows:

```
mkdir websocket_send_periodic_temp
cd websocket_send_periodic_temp
mos init --arch=esp32
```

The above command will create the file structure necessary to build the app with the mos tool:

```
mos --clean build
```

To flash the firmware to the device (assuming a solid USB connection to the device):

```
mos flash
```

Github Repository

The repository for this project:

https://github.com/Greg-R/websocket_send_periodic_temp

"Main" Function

It is not "main" as in typical C programming. For Mongoose-OS, the entry function is called "mgos_app_init(void)".

```
enum mgos_app_init_result mgos_app_init(void) {
    struct mg_connection * nc;
    void * user_data = "";
    bool dhtInit;

    //  1. Get the server handle.
    if((nc = mgos_get_sys_http_server()) == NULL) {
        puts("The value of nc is NULL");
    }

    //  2. Bind the event handler to the HTTP server.
    mgos_register_http_endpoint("/", ev_handler, user_data);

    //  3. Create the DHT22 sensor.
    if((dht11 = mgos_dht_create(2, DHT22)) == NULL) {
        puts("DHT22 handle not created");
    }

    //  4.  Initialize the sensor.
    dhtInit = mgos_dht_init();
    if(dhtInit) {
        puts("DHT sensor initialized.");
    }

    return MGOS_APP_INIT_SUCCESS;
}
```

The four significant lines above get the http server and the sensor primed for operation:

1. mgos_get_sys_http_server() is the entire invocation of a Mongoose-OS http server!
2. Since the system will be responsive to events, an event handler function "ev_handler" is bound to the http server.
3. The DHT22 sensor object is created. Note that a DHT22 device is used, as it has better resolution than the DHT11.

4. The last step is to initialize the sensor.

That is all, it is very easy!

Configuration

Note that the "empty app" does not include the http server and DHT22 device related code. Mongoose-OS partitions these features into "libraries" which must be incorporated into the app. There are two steps which are required to activate these features:

1. Add the libraries to the mos.yml file. A default mos.yml file is created by the mos init command. Add the http-server and dht lines to the mos.yml file. The "libs" section of the file should look like this:

```
# List of libraries used by this app, in order of initialisation
libs:
  - origin: https://github.com/mongoose-os-libs/ca-bundle
  - origin: https://github.com/mongoose-os-libs/rpc-service-config
  - origin: https://github.com/mongoose-os-libs/rpc-service-fs
  - origin: https://github.com/mongoose-os-libs/rpc-uart
  - origin: https://github.com/mongoose-os-libs/wifi
  - origin: https://github.com/mongoose-os-libs/http-server
  - origin: https://github.com/mongoose-os-libs/dht
```

   The libraries are described here:

   https://mongoose-os.com/docs/reference/api.html

2. Add the #include lines to the main.c file. The top of the file should look like this:

```
#include "mgos.h"
#include "mgos_http_server.h"
#include "mgos_dht.h"
```

The Event Handler Function

The "Event Handler" function responds to events generated by the http server. This includes Websocket events. The events are fired by Mongoose-OS, and the event handler uses a switch statement to deal with each individual type of event as required.

The events can be determined by examining the mongoose.h header file. It is recommended to download the Mongoose-OS git repository and study the header file:

https://github.com/cesanta/mongoose-os

The file is found in the directory mongoose.

Other examples of event handlers can be found in the example apps:

https://mongoose-os.com/docs/reference/apps.html

Here is the source code:

```
static void ev_handler(struct mg_connection *nc, int ev, void *ev_data, void *user_data) {
    struct http_message *hm = (struct http_message *) ev_data;
    switch (ev) {
    case MG_EV_HTTP_REQUEST: {              // #1 HTTP request
        mg_serve_http(nc, hm, http_opts);
        break;
    }
    case MG_EV_SEND: {
        // puts("MG_EV_SEND event fired!");
        break;
    }
    case MG_EV_WEBSOCKET_HANDSHAKE_DONE: {
        //  This sets the periodic time updating in motion.
        mg_set_timer(nc, mg_time() + INTERVAL_SECONDS);  // #2 Timer initialization.
        break;
    }
    case MG_EV_WEBSOCKET_FRAME: {
        // printf("Websocket message received:\n");
        break;
    }
    case MG_EV_CLOSE: {
        break;
    }
    //  The timer event causes a JSON string to be sent to the
    //  webpage via Websocket.
    case MG_EV_TIMER: {          //  #3 Most of the action is here!
        maintime = time(NULL);
        localstruct = localtime(&maintime);
        //  printf("The seconds are %d.\n", localstruct->tm_sec);
        strftime(utctime, sizeof(utctime), "%A %B %d %T", localstruct);
        tempC = mgos_dht_get_temp(dht11);
        //  printf("The temperature in degrees C is %2.1f.\n", ctof(tempC));
        humidity = mgos_dht_get_humidity(dht11);
        //  printf("The humidity in percent is %2.1f.\n", humidity);
        //  printf("Time is %s.\n", utctime);
        //  #4 Update time every 1 second; temp and humidity every five minutes.
        mg_printf_websocket_frame(nc, WEBSOCKET_OP_TEXT, timeString, utctime);
        if (count == 300) {
            mg_printf_websocket_frame(nc, WEBSOCKET_OP_TEXT, tempString, ctof(tempC));
            mg_printf_websocket_frame(nc, WEBSOCKET_OP_TEXT, humidString, humidity);
            count = 0;
        }
        count++;
```

```
        //  A new timer is set.
        mg_set_timer(nc, mg_time() + INTERVAL_SECONDS);
        break;
    }
    }
}
```

The event handler function must follow the correct prototype pattern. Again, refer to the header file mongoose.h.

Mongoose-OS takes care of calling the event handler function when it is required. This is a fundamental feature of the OS and the http-server library. All is built-in and nicely integrated with WIFI. WIFI configuration will be covered later.

The developer is responsible for customizing the event handler to meet the system's requirements. This is where the action happens with regards to what behavior is desired from the system.

Note that several unused event types appear in the source code fragment. These were left in place to show other possible Mongoose-OS events. Now to describe the events used in this project:

1. MG_EV_HTTP_REQUEST. This is the usual response to an http "get" request, and this sends the response. In terms of this project, this will be the html and embedded javascript, and the style sheet. The event handler is bound to the "root" which is "/". In use the URL will be as follows, making the assumption the IP address is 192.168.1.4:

   `http://192.168.1.4/websocketweather.html`

2. MG_EV_WEBSOCKET_HANDSHAKE_DONE. This event is fired upon completion of the Websocket handshake process. This sets the Mongoose timer in motion, and Websocket messages will begin to flow.
3. MG_EV_TIMER. This event is fired at the expiration of the Mongoose timer. This event creates all of the action in this project. The timer is initially set in step #2, and then re-set in this branch of the switch statement. The function "mg_printf_websocket_frame" is used to transmit Websocket frames. Note the use of POSIX type functions to access time data. These functions are included by default in Mongoose-OS. Note that the ESP32 must successfully connect to the internet and an NTP server for these functions to work properly. The time data is not used for any timing purposes— it is only used for time display in the web page.
4. This is where the mg_printf_websocket_frame function is used to sent Websocket messages. The messages are formatted strings in the form of JSON objects which can easily be parsed and utilized on the receiving end which is the Websocket enabled web page served by the event MG_EV_HTTP_REQUEST. The Mongoose-OS timer is set to 1 second by the pre-processor define "INTERVAL_SECONDS". A counter and if statement cause the DHT22 sensor data to be sent every 300 seconds (five minutes). The last line of this case initiates another timer. The cycle then repeats...

ESP32 Code Summary

C code used to implement complex functionality is easy thanks to Mongoose-OS. The low-level details are taken care of thanks to the provided libraries. Commonly used functionality such as timers are built-in to the OS.

Chapter 3

The Web Browser Code: HTML, CSS, and Javascript

The single-page app displays the incoming temperature data from the ESP32 server. The data is plotted in a single line chart versus time.

The current hard-coded default time period displayed in the chart is 12 hours.

D3.js is used to build the line chart. The web browser must have Websocket and ES6 version Javascript.

The Websocket and message parsing Javascript is easy to follow. However, the D3.js code used to display the data in the line chart is a little more complicated. There are numerous resources available for learning D3.js. Here are the top resources:

https://d3js.org/

and

https://bl.ocks.org/

This project used this as a starting point:

https://bl.ocks.org/curran/ba21316eafc6b84b22d1a5d49ad2a798

The author of the above line chart has a nice youtube tutorial on D3.js:

https://www.youtube.com/watch?v=8jvoTV54nXw

HTML5, CSS, and Javascript Code in websocketweather.html

The ESP32 connects via websocket to the web page websocketweather.html. The file is located in the fs directory in the Mongoose-OS project.

The event handler code is bound to the route "/", so the URL will be:

```
http://192.168.1.4/websocketweather.html
```

where the example IP address is 192.168.1.4 is shown.

If the ESP32 is configured for station mode, the IP address can be determined using the router web interface. Another method is to monitor the ESP32 as it boots. This is done using this command:

```
mos console
```

at the command line, and assuming the ESP32 is connected via USB/UART. The IP address will be shown as the ESP32 successfully connects to an access point.

Onto the code...

For this "single page app" the CSS and Javascript are embedded in the web page. In the "head" block, there is a script tag which loads the D3.js code for building the line chart. This is followed by a block of conventional CSS styling which is applied to various pieces of the line chart.

The structure of the page is simple, there being only a few lines of HTML5 text and then the line chart. Most of the code is a script block with the Javascript necessary to build the line chart. There is a message handler which parses incoming JSON into time and temperature data which is used to update the chart.

The x axis is fixed to a 12 hour time span. Temperature data is plotted at 5 minute intervals. This seems sufficient to produce a smoothly changing line.

The y axis is fixed to a range of 50 to 100 degrees Fahrenheit. This can be changed to any desired range easily enough, and the Celsius to Fahrenheit conversion function can be removed to display the native Celsius output of the DHT22 sensor.

Websocket Connecting

This bit of code determines the correct IP address, and then negotiates a Websocket connection to the ESP32. The trick is to use a Regular Expression to extract the IP address. Then it is simple to use the Websocket API to open the connection.

```
const serverUrlRegex = /\d+\.\d+\.\d+\.\d+/; //  Matches 192.168.1.8 etc.
const currentUrl = window.location.origin; //  Get the URL from the browser.
console.log(`The currentURL is ${currentUrl}.`);
const serverUrl = currentUrl.match(serverUrlRegex);
console.log(`The server URL is ${serverUrl[0]}`);
let ws = new WebSocket(`ws://${serverUrl}`);
```

Again using the Websocket API, the status of the Websocket is indicated on the web page. This little textual indication of Websocket status in the web page proved very important.

```
ws.onopen = () => {
    console.log('Web browser opened a WebSocket.');
    //  Update the connection status in the browser.
    connectstatus.textContent = "WebSocket Connected";
```

```
}
ws.onclose = () => {
    console.log('Web browser WebSocket just closed.');
    //  Update the connection status in the browser.
    connectstatus.textContent = "WebSocket is disconnected";
}
```

D3.js Line Chart

This D3.js code is perhaps a little unusual in that the chart is not constructed until the first time data is received from the ESP32. The code gets the first time stamp, and then uses this to construct the x axis. So the first incoming time is the left end of the time axis, and the end is set to 12 hours later.

This is done by putting some of the line chart building code in a function, and then only running that function after the first time data is received. This is done only once, thus the variable

```
let chartBuilt = false;
```

is initialized to false, and is changed to true at the conclusion of the processing of the first incoming time data:

```
if (chartBuilt === false) buildChart();
```

and then chartBuilt = true at the conclusion of the buildChart() function.

Websocket Message Handling

The Websocket API is once again called upon, this time to respond to events caused by incoming messages from the ESP32.

```
ws.onmessage = (message) => {
    console.log(message.data);
    //  Parse the incoming JSON data from the ESP32.
    let statusMessage = JSON.parse(message.data);
    //  Check the message type and handle as required.
    if (statusMessage.messageType === "temperature") {
        console.log(`wxData message received`)
        let newTempData = {
            'time': currentTime,
            'temp': +statusMessage.currentTemp
        };
        //  Add a new data object to the end of the array.
        tempData.push(newTempData);
        //  Update chart.
```

```
        update(tempData);
        //  Update textual display.
        temperature.textContent = statusMessage.currentTemp;
    } else if (statusMessage.messageType === "humidity") {
        humidity.textContent = statusMessage.currentHumidity;
    } else if (statusMessage.messageType === "serverTime") {
        //  Current Server Time Updater.  Updated every 1 second by server.
        currentTime = new Date(statusMessage.serverTime);
        console.log(`The current time from the server is ${currentTime}`);
        let hourMinuteFormat = d3.timeFormat("%H:%M");
        let hourMinute = hourMinuteFormat(currentTime);
        console.log(`hour and minute ${hourMinute}`);
        clock.textContent = currentTime;
        //  The chart is built only after receiving the first time data.
        if (chartBuilt === false) buildChart();
    }}
```

The code is almost self-explanatory. Incoming Websocket messages are parsed into proper Javascript objects, and an if-else-if block is used to handle the messages. Each message has a "messageType" field, and this is used to process the message accordingly.

Note that in the case of incoming temperature data, the array tempData is updated, and then the line chart is updated via the call to the update() function.

The humidity data is current only used in the textual display. A second line chart for humidity could be added easily enough.

Volatility of the Chart Data

This is a simple demonstration, and the data is not stored in non-volatile memory. If the browser is refreshed, the data will be lost. Also, the data does not wrap when it reaches the end of the 12 hour time period.

The next project will use Google IOT or AWS to allow non-volatile data storage.

Chapter 4

WIFI Configuration for STA and AP

This chapter is a tutorial on how to configure the device for either WIFI station or access point modes.

There may be other methods to accomplish this. I found the following method to work successfully and to have some flexibility. Keep in mind that Mongoose-OS configuration facilities are designed for commercial products and end-user configuration. This will deviate from that goal a bit, being that this project is targeted at the hobbyist and experimenter.

Mos tool for Configuring as Station

It is easy to use the mos tool at the command line to configure WIFI.

For example, let us say you have a home router, and you would like your device to connect to it, preferably with DHCP active in the router.

The default configuration of the device will be as an "Access Point". The following series of commands will switch your device to a station:

```
mos wifi WIFI_NETWORK_NAME WIFI_PASSWORD
```

The device should connect to the router quickly. If it does not, make sure any sort of router access control settings are temporarily disabled.

The settings are volatile with regards to flashing the device. The mos wifi command will be have to run again each time the device is flashed.

However, this can easily be made permanent. Use this command:

```
mos ls
```

You should see a listing of files something like this:

```
ca.pem
conf0.json
conf9.json
```

```
index.html
sys_config_schema.json
sys_ro_vars_schema.json
```

The key here is the file conf9.json which is the “user configuration file”. To make this permanent, a copy needs to be made and moved to the fs directory in the project directory:

```
mos get conf9.json > fs/conf9.json
```

The above command pipes the file from the device into the fs directory with the name conf9.json. Now you need to build and flash the project again:

```
mos --clean build
mos flash
```

Now run mos ls again, and you should see that the file con9.json is in the device’s local file system.

Back to an Access Point

Simply delete the file conf9.json from the fs directory and flash the device. The default IP address seen is 192.168.4.1 and the default password is Mongoose.

A good way to monitor the boot-up process is to use the console:

```
mos console
```

Pressing the reset button on the development board should start the boot process, and numerous messages will be written to the terminal. This will include IP addresses in the case of a station connecting to a router with DHCP and also connection status. Very useful!

Chapter 5

Resources

The Mongoose-OS documentation:

https://mongoose-os.com/docs/quickstart/setup.html

The Adafruit ESP32 Feather Board:

https://www.adafruit.com/product/3405

The DHT22 temperature and humidity sensor:

https://www.adafruit.com/product/385