Group N
Dylan Gardner
Rich DeFano
Nick Grove

ECE: 4880 Lab 1 Report

Design Documentation:

Considerations/Tradeoffs:

One of the first choices we had to make was the microcontrollers used for our server and client. We decided on the ESP8266 for both the server and client as the chip was small, and contained the hardware for wifi connection on board. We did not want to use something overly complex like a raspberry pi, as we felt it was a waste of resources and power. This did leave us with space issues, as the server had a very small amount of memory. The few files for the web page needs to be compressed, and things like special fonts left out. The board itself is powered off of AA batteries with the voltage stepped down, which was easier to work with than a Lithium-Ion battery due to not needing recharge/discharge circuitry and protection circuitry. Our main focus was making sure we had enough resources to finish the project, although in a real-world setting things like server design and data structuring for storage would play a much larger role.

System Overview:

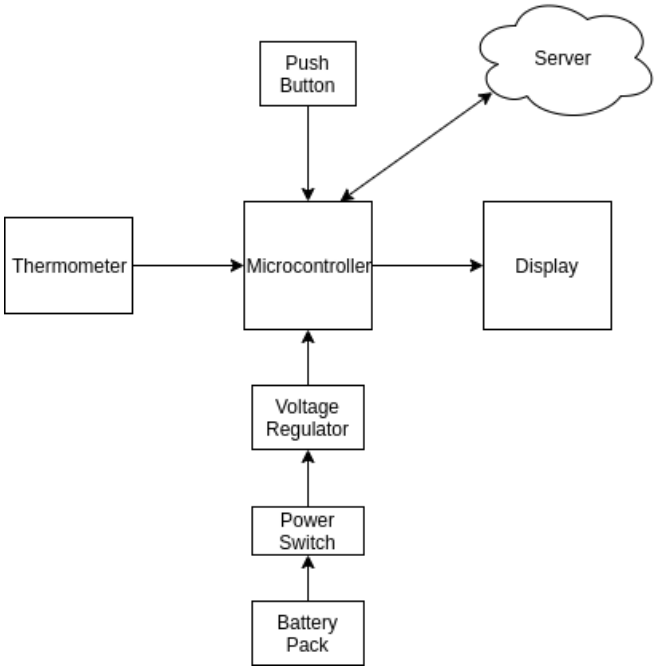

Figure 1: Block diagram showing interaction in the system

As seen in the high-level overview of the system in **Figure 1** there is the main thermometer box which is the bulk of the system. This contains the display, power switch, temperature sensor, and button for toggling the display. The thermometer is

designed to be a contained unit that when the button is pressed, the current temperature is shown on the display. This temperature is also sent to a server to be viewed on a graphical interface.

Schematic:

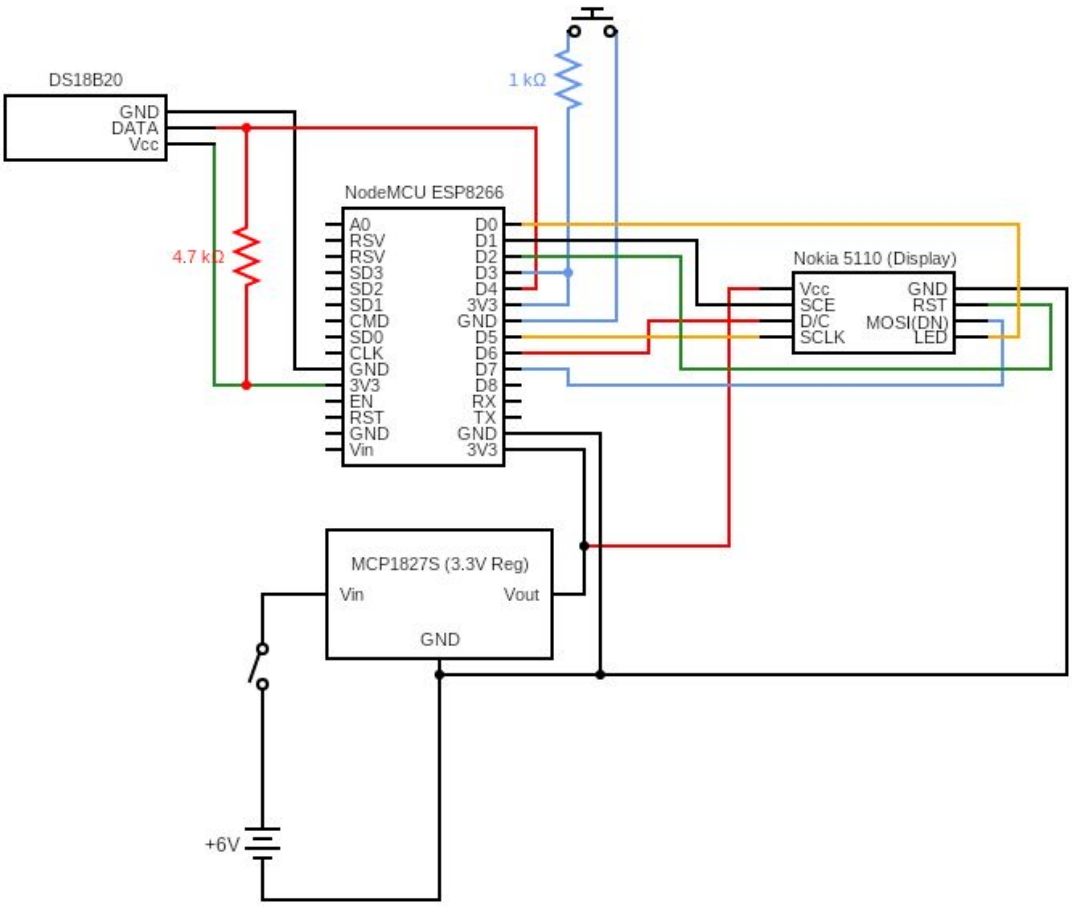

Figure 2: Schematic showing third box connections

Figure 2 shows how the third box is wired with the other components. The box uses 4 AA batteries which generates 6 V. The LCD needs 3.3V so the voltage has to be stepped down using a voltage regulator. Since the ESP8266 microcontroller and DS18B20 temperature sensor can also use 3.3V for their input voltages, the entire system was powered from 3.3V. This reduced our overall power consumption and extended battery life. The box contains a power switch along with a push button that triggered a pin interrupt in the microcontroller causing the microcontroller to communicate with the display, using SPI, and wrote the current temperature to it. The microcontroller also used a hardware timer to get the temperature from the temperature sensor every second as the process of getting the temperature took at max 750 ms.

Server Code Description:

The server does not have as many tasks as the client, and mostly handles incoming and outgoing data. The server will load up and connect to wifi, which allows it to open a websocket connection with a web page, and HTTP connection with the client. As the client sends data once a second through interrupts, the server receives the data. This data is then written to the CSV, send to the web page, and compared to the set minimum and maximum values. If the value is outside the bounds, a text is sent through an email SMTP server. Most differences in operation depend on what code is sent with the requests. Specifically for the websocket, it will send various strings to tell the server what to do, such as if the software button is pressed, send code “software”. If a phone number is put in, send that number to a variable in the server.

Client Code Description:

Figure 3 demonstrates how the client works internally. When powered on the client has to do some setup for both the temperature sensor, display, and some other miscellaneous configurations. Next, it will attempt to connect to one of the provided wifi networks, if it can't within 10 seconds it will timeout and store that it isn’t connected. After this setup is complete the client goes into an infinite loop. Within this loop, it checks for two main things; if the temp is requested and if updateDisplay is requested. Every second, triggered by a timer interrupt, the temperature is read from the temperature sensor and a new one is requested since the process can take up to 750 ms. If this new temp is read, the client sends the temp to the server. The next thing the client does is see if the display needs to be updated. This happens when either the push-button changes state or when a new temperature is available.

Since the client sends the temperature to the server every second and gets a response code from the server, an easy way for the server to tell the client to turn on the display (software button) is to include something in the response that the client can parse to identify when to toggle the display.

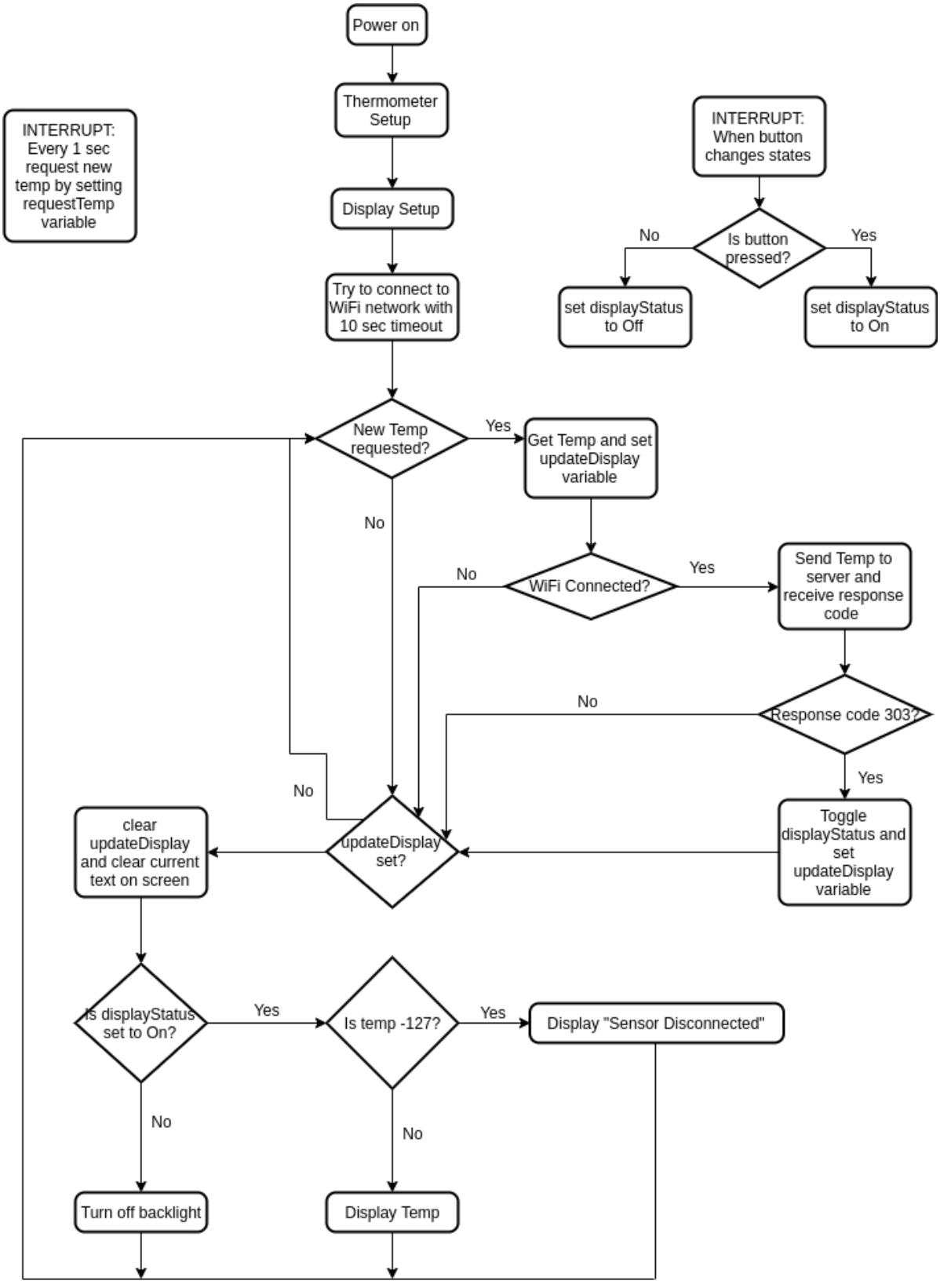

Figure 3: Flow diagram of the client process

User Interface Description:

Figure 4 shows the Web User Interface for our thermometer. The graph shows the user all data from the last 300 seconds and also displays messages to the user whether the temperature sensor is unplugged or data is unavailable. The graph side of the UI is mainly just for general data whereas the banner is for current data and user interaction. The banner has sections for text updates setup, current temperature, and toggling buttons (for toggling temperature units and toggling the LCD on the display on and off). Having these two sides of the UI is perfect for the user to switch their focus between seeing what is happening overall with the device to device interaction.

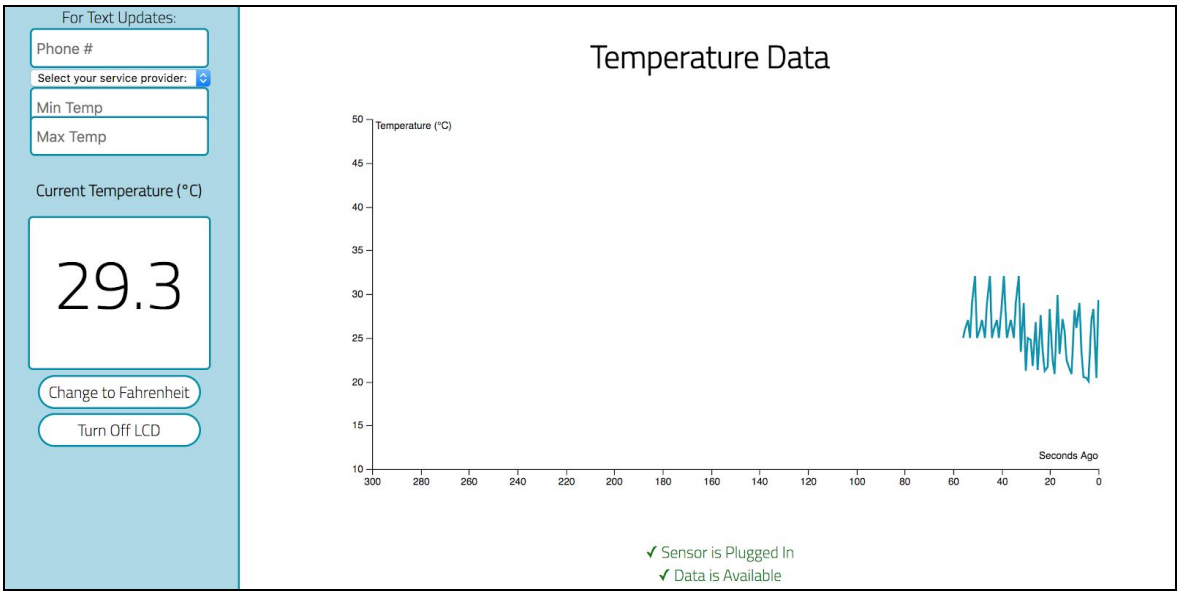

Figure 4: User Interface that displays temperature data

Figure 5: Final enclosure and connections

The enclosure was 3D printed, as seen in **Figure 5**, as we felt that was the best way to minimize the footprint of the device and get a sturdy and custom fit for all of our components. The display was fairly small which helped reduce power consumption as well as reducing the footprint. The button and switch are designed to be mounted onto a box like we have and made the final prototype look much cleaner and sturdy. The connector for the temperature sensor was not ideal since it was a 4-wire connector but it was the best we could find at the time, but we made it work by pulling one of the wires out and only using 3.

Project Retrospective

Team Roles:

- Dylan - Focused primarily on the hardware aspect of the lab. Designed the physical enclosure for the thermometer as well as choosing, wiring and assembling the components. Also wrote the code on the client (thermometer) that reads the temp, turns the display on and off as well as sending the temp to the server.
- Rich - Created the code for the server, as well as focused on the interconnection between the web page, server, and client. The server and the web page communicate through a websocket, and the server and client communicate through HTTP Post requests. A large focus from my end was getting and sending variables from each portion to be used in the other parts, such as temperature being sent to the server, which then is written to a CSV and sent through the websocket.
- Nick - Programmed the Web User Interface for the Thermometer. Used CSS and HTML for the interface's elements and used JavaScript for the functionality. Utilized JavaScript's d3 data visualization library for graphing the data and reading the CSV file that was sent for initial data.

Overall, we believe we met a majority of the requirements. We had a few small issues, such as sending too much voltage to our LCD very close to checkoff (which ruined a section of display pictures, although it was still readable), and cleaning up some graphical errors (before displaying a null temperature, it would shoot down to -127 on the graph for one tick). I believe our main concerns moving forward have to do with the merging of each group member's work and giving ourselves ample time to fix errors that come up post-merge. In terms of the graph, it was displaying correctly, although once merged with the server and CSV file started to display this line. This error was not in one portion(JavaScript/Server/Client), but how the data was being sent between the three. We believe the group successfully completed a valid prototype for the design, with a small number of constraints being close but incomplete for checkoff.

Test Report

Connection Testing		
Input / Condition	**Pass Criteria / Expected Result**	**Pass / Fail**
The IP address of the server, typed into a browser, along with any /extra additions to a URL to specify page	The page shown in the browser will be fetched from SPIFFS memory and displayed to the user. If there is no file path given, assume its /index.html. If the page does not exist in SPIFFS, throw a 404 error	**Pass**
Receiving codes tied with variables over the websocket	The server may receive various strings from the web UI, which will update values held on the server for variables like phone number or carrier.	**Pass**
Sending temperature readings over the websocket for real time graphing	When the server receives a temperature from the client, it also needs to send it to the web page to be displayed and appended to the current data	**Pass**
Checking if the client is no longer connected	The server has an internal timer of 5 seconds from the last time the board received data from a client. If the timer runs out, the board is pinged and its active status checked. This will tell the website if the board is disconnected/off.	**Pass**

Functional Testing		
Input / Condition	**Pass Criteria / Expected Result**	**Pass / Fail**
Pressing the push button, while on with the temperature sensor **plugged in**	The display shows the current temp in C in less than 20 ms, and on UI within 1 s	**Pass**
Pressing the push button, while on with the temperature sensor **unplugged**	The display shows an error saying the sensor is unplugged	**Pass**
Pressing the software button on the UI, while the third box is on	The display toggles (turns on/ turns off) and shows temperature or sensor disconnected within 1 second	**Pass**
Unplugging the temperature sensor, while the third box is on	The UI shows that the sensor is disconnected, and shows a gap in	**Pass**

	data within 1 second	
The power switch is set to ‘off’ position	The UI shows that real-time is not available, and shows a gap in data in the graph within 5 seconds	**Pass**
The power switch is set to ‘on’ but not connected to the network	The push-button still produces a temperature but the UI shows real-time data is unavailable	**Pass**
Loading webpage for the first time	Previous 300 seconds of data should be visible on the graph within 10 seconds	**Pass**
Text updates can be setup from the web UI	The user can give their number, provider, and set a range for updates. Once the temperature goes outside of that range the user gets a text update.	**Pass**
Live data is continually graphed and graph scrolls with new data	Every second a new temperature is pushed onto the graph and the line scrolls along the page as new data comes in	**Pass**
Page is scalable	Page is the same unless it is taken under 1000 pixel width and then the left banner is taken away to make room for the graph	**Pass**
Measurement Testing		
Measure the voltage supplied from the batteries, before the voltage regulator	The voltage should be between 5.5 - 6.5 V	**Pass**
Measure the voltage supplied from the batteries, after the voltage regulator	The voltage should be between 2.9 - 3.5 V	**Pass**
Performance Testing		
The battery must last 15 hours with continuous use	Given new batteries, the third box shouldn’t die within 15 hours. Calculated with a capacity of 2800mAh and an avg current drain of 135 mA. That gives about 20 hours of use	**Pass**
The third box and connectors must be	When dropped from a workbench the	**Pass**

of sturdy construction	third box and connectors will not break or have minimal, user fixable damage (cord disconnecting, battery cover coming off)	
The connectors must be easy for a user to connect and disconnect	A user, unfamiliar to the third box, connect and disconnect the temperature sensor with one hand within 20 seconds	**Pass**