Better Barista: Brew Data Interface

Tejus Rao & Samin Zachariah

E155 - Microprocessor based systems

December 1st 2020

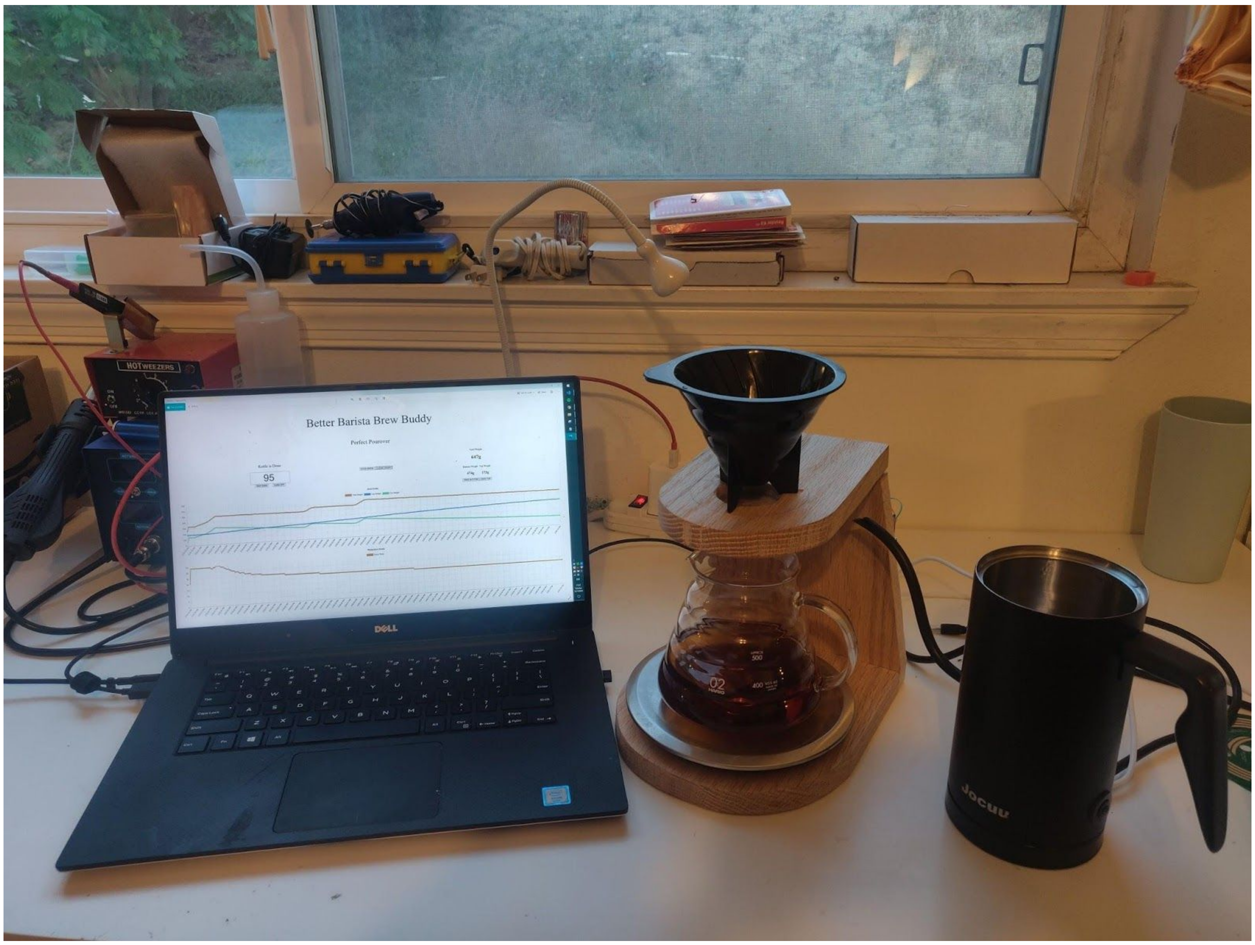


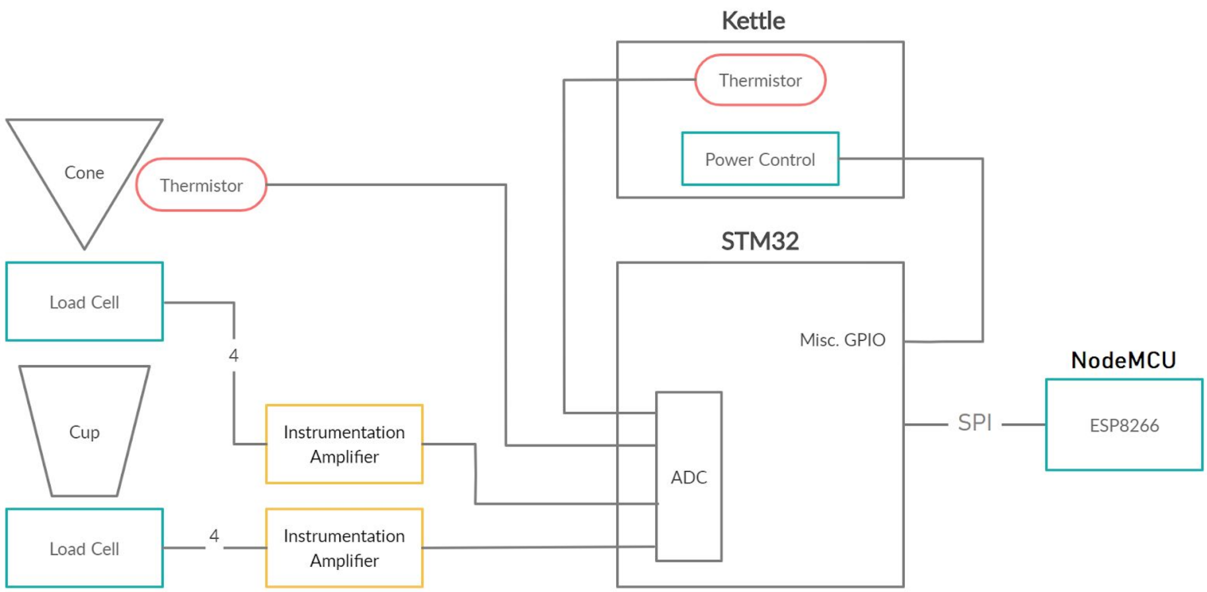
Table of Contents:

Overview

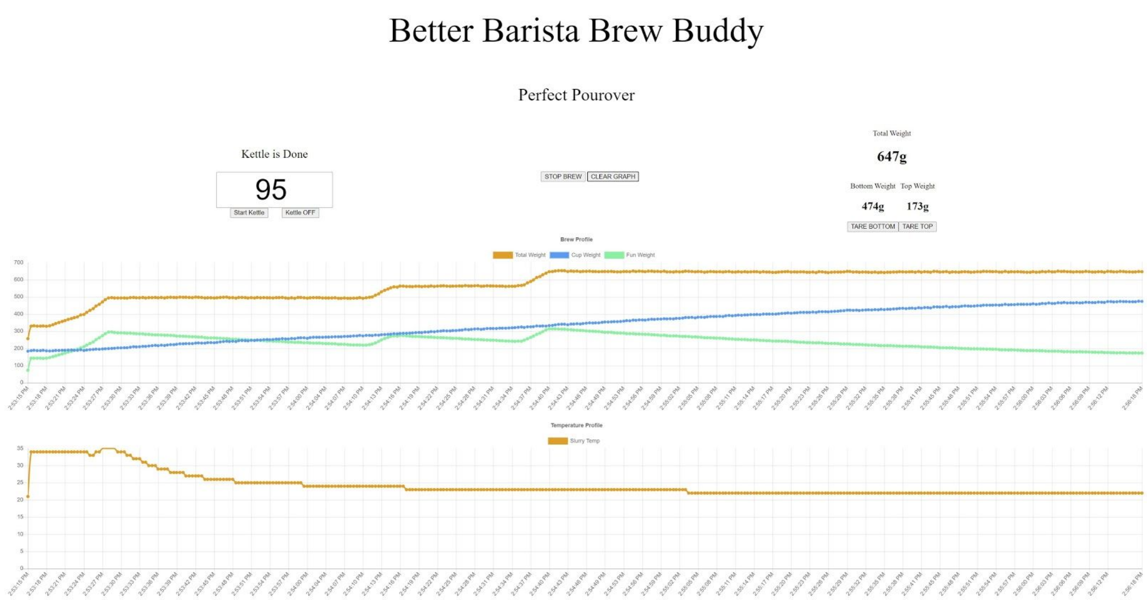
The Better Barista is a system designed to help home baristas improve their pour over technique and make consistently tasty coffee. In pour over brewing, due to the amount of manual control in the brewing process, it is hard for home baristas to control the variables involved in brewing. In order to make repeatable coffee however, controlling these variables is essential. Some of these variables are coffee ground size, coffee dose, water temperature, water pouring rate, the flow rate through the coffee bed, the total weight of the brew, and the total time to brew. Furthermore, certain styles of pourover have varying flow rates over time, which is very hard to make repeatable without external aid.

The Better Barista provides a solution to this problem by plotting the total weight, the cup weight, and the funnel weight on a live graph. This way the barista can visually see flow rate over time and use the live feedback to adjust their pour to match their desired brew profile. Users can also use the data from the resulting brew profiles to see which types of pouring styles they may prefer.

To accomplish this live feedback system the Better Barista uses two load cells to measure weight in the cup and in the funnel, as well as a thermistor to measure the temperature of the coffee/water slurry. This data is recorded by an STM32 microcontroller and sent to a NodeMCU. The NodeMCU, a development-board for the ESP8266, serves a web-app through which users can view the live graphing of weights and temperature. Additionally, the system interfaces with a Jocuu electric kettle to provide remote start/stop and digital temperature control features, accessible through the Better Barista web-app.



Block Diagram



Interface displaying a 3-pour brew with constant flow rate

Mechanical Design

Stand construction & Sensor Mounting

The requirements for the stand design were to use two load cells to measure the weight of the v60 cone and the weight in the cup, and to mount the thermistor in such a way that it is capable of measuring the temperature of the water in the cone.

Bottom and Top Load Cell Mounting

The primary consideration when designing this was fabrication constraints due to a significant restriction of available tools.

In mounting the thermistor in the v60 funnel a compromise is required between usability of the system and temperature measurement quality. For the best measurement of water-coffee-slurry temperature the thermistor should be in the middle of the cone. This is impractical however, as the user is required to change the paper filter in the cone each time, and the filter cannot be punctured to accommodate the thermistor. Because of this the thermistor is mounted on the wall of the funnel so that it is in contact with the outside of the paper filter.

Connectors

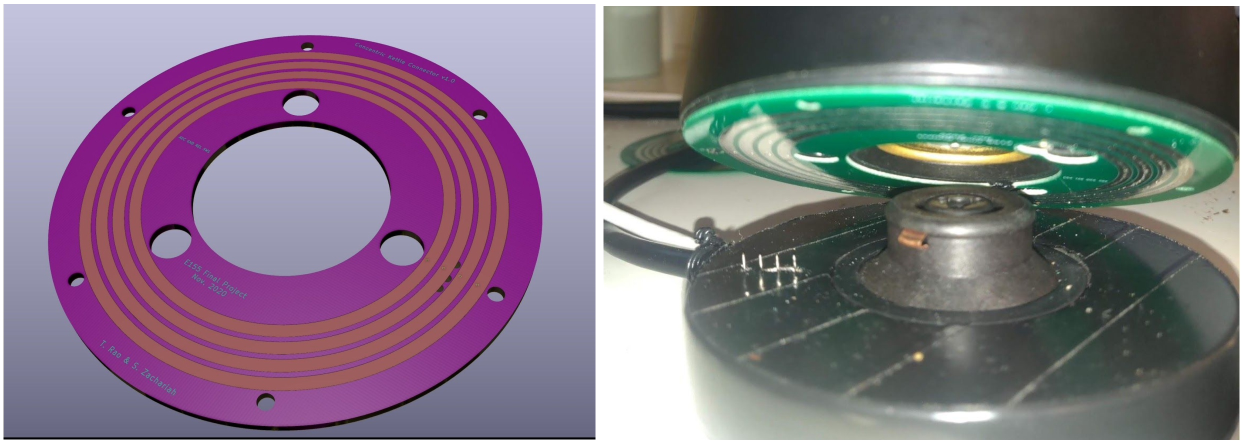
There are two components in this device that contain sensors that must also be removed from their respective resting places. These are the funnel, which must be removable from the top stand for washing, and the kettle which must be taken off its base to pour. This provides connector design challenges as ergonomics are extremely important for a design that is meant to be interacted with daily.

In order to allow the funnel to be removed, the thermistor inside the v60 funnel is connected to a female header embedded in the plastic of the funnel. This connects to a pair of male headers in the top stand. When the funnel is placed on the stand there is a robust connection between the thermistor inside the funnel and the voltage divider. The funnel can be removed and washed while still having a robust connection between the thermistor and the STM32 microcontroller. This design also allows the funnel to be used in a standard pour over setup.

Funnel Thermistor Connector

The thermistor used is a waterproof 10kΩ thermistor, connected to the STM32's ADC input through a voltage divider. As the temperature changes, the thermistors resistance changes and this results in a voltage change at the ADC pin.

For the kettle, in addition to being removed, it must also be able to rotate on its base. This is so that the user can place the kettle down on the stand in any orientation, as well as pick it up without worry of breaking a connector. In order to achieve this rotational flexibility a custom PCB was fabricated. The PCB contains large concentric pads on one side, and small pads wired to the signals from the microcontroller on the kettle. The PCB is then mounted on the bottom of the kettle. Four spring pins are mounted in the kettle base such that each contacts one of the concentric pads. The spring pins ensure solid connection between the kettle signals and the STM regardless of orientation. A four-conductor USB wire carries these signals to the main electronics system.

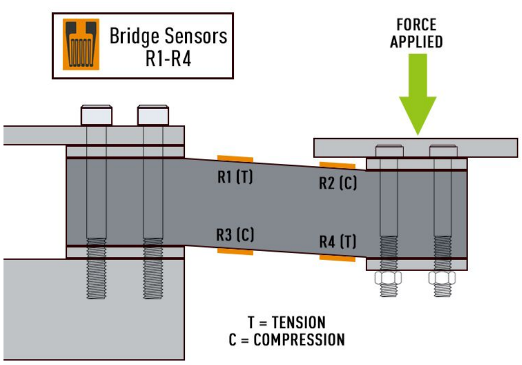


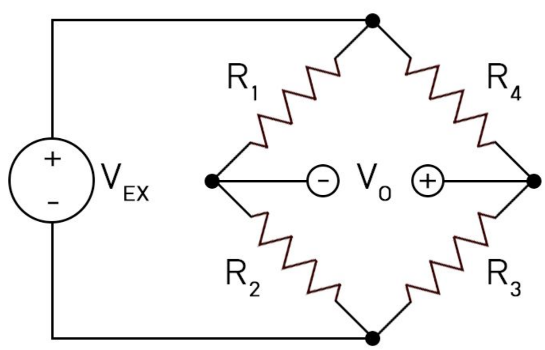
PCB and spring pin interface

Electrical Design

Load Cell Interface

The load cells, which are constructed from 4 strain gauges, can be modeled as wheatstone bridges. An excitation voltage is applied to two nodes, while fluctuations in force change the resistances such that the output node voltage differs. The quantity of interest is actually the difference in voltage at those two nodes, Vo + and Vo-, which is a very small quantity.





Initial testing showed that with a 5V excitation voltage, over a range of 0 to 1000 grams, the output voltage differential only fluctuates between approximately 0 to 3mV. Thus, large amounts of amplification were necessary, but achievable. Typically, load cells are followed by an instrumentation amplifier, such as the popular AD623, which offer a significantly better common mode rejection ratio than a standard non-inverting amplifier, for example, due to the use of a differential pair of signals. This is important when dealing with such small voltage signals.

Unfortunately, this was not available during testing. While a differential amplifier can be made with a single op-amp, the input impedance of signals can often affect the gain, even if it's been set by resistors. Instead of having to determine the input impedance and compensate for it, a three op-amp instrumental amplifier was implemented using a quad-package single ended op-amp (MCP6004), which is unaffected by the input impedance.

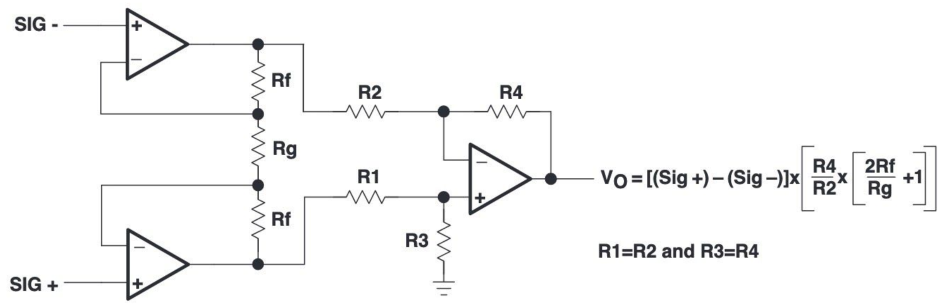


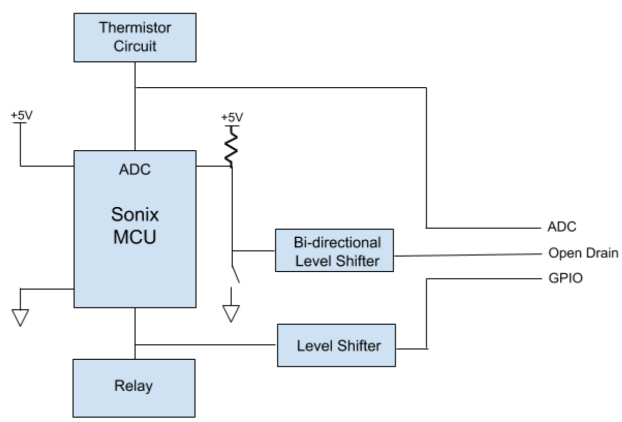
Figure 5. Three Op Amp Instrumentation Amplifier

For testing purposes, a R1-R4 and Rf were set to 10kΩ and a gain of 201 was achieved with Rg set to 100Ω. This worked very well, and allowed for the measurement of the following calibration curves for both load cells. Conveniently, the curves show the kind of linearity expected of these sensors (see appendix). Note that the gain of 201 has been factored out for these curves.

An additional differential amplifier with a gain of 6.9 and an adjustable offset follows the instrumentation amplifier for both load cells in order to spread out the range of values through the entire 0-3.3V ADC range. This offers more precision in measurement with the on-board Nucleo ADC. The adjustable offset for each load cell circuit is achieved through a voltage divider with a potentiometer, which allows for calibration. Unity gain buffers are used on the offset voltage signals so that the impedances of the voltage dividers do not interfere with the amplifier gains. It is also important to mention that though these load cells are rated for up to 2kg, the decision was made to only take advantage of their 0 to 1000g range. This decision was made so that the proposed precision of ±1g could be met, even with some noise. Collectively, these decisions and considerations have all been made to most effectively map the Nucleo ADC's 0-3.3V range to values between 0-1000g with the maximum possible precision, approximately ±1g.

Kettle Interface

The project proposal discussed the modification of a simple electric kettle to implement functionality such as remote start and temperature control. In order to achieve this, the kettle's internal circuit was studied closely. The major components include a full-bridge rectifier and large smoothing capacitor with a discharge resistor, a relay, a thermal cutoff switch, a thermistor, a SONIX 8-bit microcontroller, a pushbutton, and several passives and LEDs. After mapping out the majority of the circuit and developing a decent understanding of the microcontroller's signals by using a logic analyzer, the circuit was simplified to the components of interest. It was desirable, for several reasons, to have the kettle still able to function in its "standalone mode". An open-drain configured pin on the Nucleo connected to the kettle's pushbutton input on the SONIX MCU was used to act as an external switch. This allows both the pushbutton and the Nucleo to pull down the voltage on this bus at any point without interfering with pin output settings. Another Nucleo pin is used to read the digital control signal to the relay to determine when the kettle is on, and an ADC channel is used to read the analog voltage coming into the SONIX ADC from the kettle's thermistor circuit. After some testing, it was found that the kettle shuts off when this voltage reaches 1.37V.



One further challenge was that the SONIX MCU was 5V logic compatible, whereas the Nucleo is 3.3V compatible. As such, a bi-directional level-shifter circuit was constructed with an n-channel MOSFET that allows the switch bus to be pulled low by either the pushbutton or the open-drain Nucleo pin and to idle high at each side's respective logic level. An additional logic level shifter circuit constructed as a voltage divider was used to read the digital relay signal from the kettle. (See Appendix A for level shifter schematics).

Finally, a calibration curve was measured for the thermistor circuit voltage within the kettle. During tests, the device demonstrated "remote" kettle control temperature accuracy of ±1°C.

ADC Noise Reduction

Noise was a significant challenge in dealing with these signals. For the load cells, even with very stable instrumentation amplifier outputs, and very stable offset voltages into the differential amplifier, the signals going into the ADC fluctuated rather significantly by about ±5mV at times. Probing the power rails, periodic and gaussian noise were observable, though it was difficult to detect the source. Even with several bypass capacitors near ADC input pins, there was still fluctuation. For the kettle's thermistor, 60Hz pick-up was found to be a leading cause of fluctuation, which seems reasonable given that the kettle's signals are all very close to AC 110V mains power lines. This was dealt with in software by averaging 20 samples over one period of a 60Hz sinusoid. This yielded only ±1LSB of noise in the ADC measurements. For the load cells, due to long cables and many amplification stages, the best way to handle noise given the available resources was also in software. Over the course of approximately 30ms, 2000 samples are taken and averaged, which resulted in approximately ±3LSB of noise for a constant load. Given that 1LSB was equivalent to approximately 0.25g, this was within our tolerance specification.

Software

The software components are firmware running on the STM32 to collect sensor values, a server hosted on the NodeMCU, and a frontend web-application.

STM32 firmware

The STM32 conducts the sensor reads using the onboard ADC, and controls the kettle heating logic. These functions are triggered upon SPI interrupts generated from incoming signals from the NodeMCU. When the SPI interrupt is triggered the incoming data is parsed for the instruction type, temperature setting, and sensor ID, then executed on the STM32 with the result being written into the SPI TX register to be sent back to the NodeMCU. The ADC threshold watchdog is used to generate interrupts when the kettle reaches the desired temperature or is lifted.

We used the peripheral libraries provided by Professor Joshua Brake for the STM32F401RE, along with the `main.h` file from lecture 22

ADC

The new functionality used on the STM32 is the ADC peripheral. Four ADC channels are used, two for the output of the load cell circuits, one for the output of the thermistor voltage divider, and the kettle thermistor voltage. The four channels are scanned in sequence, and several measures were used to decrease noise in the samples.

SPI

The STM32 is configured as an SPI slave device to receive commands from the NodeMCU. This is done so that the NodeMCU can request sensor values only when the web-app is actively being used. If the STM were configured as a master device, it would require constant polling of the NodeMCU to determine if a request was made to the server. The commands received from the NodeMCU consist of 16 bit transmissions.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES		Instruction type		Instruction arguments											

Instruction type specifies whether to perform a sensor read or a kettle settings update. In the case of a sensor read, the instruction argument specifies which sensor to read. In the case of a kettle update, the instruction argument is a 12-bit ADC threshold value. This threshold is given to the analog watchdog which raises in interrupt when the target is reached.

The STM32 processes the required instruction, and sends the corresponding data back to the NodeMCU.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Kettle Status		Sensor ID		12-bit ADC read											

The Kettle status indicates whether the kettle is on/off, heating, or off the base. The sensor ID corresponds to which load cell or thermistor the ADC value is from.

NodeMCU

The web-app consists of a backend running on the NodeMCU that is written in C using an HTTP server library, and a frontend web-application that runs in a browser written in HTML, CSS, and javascript.

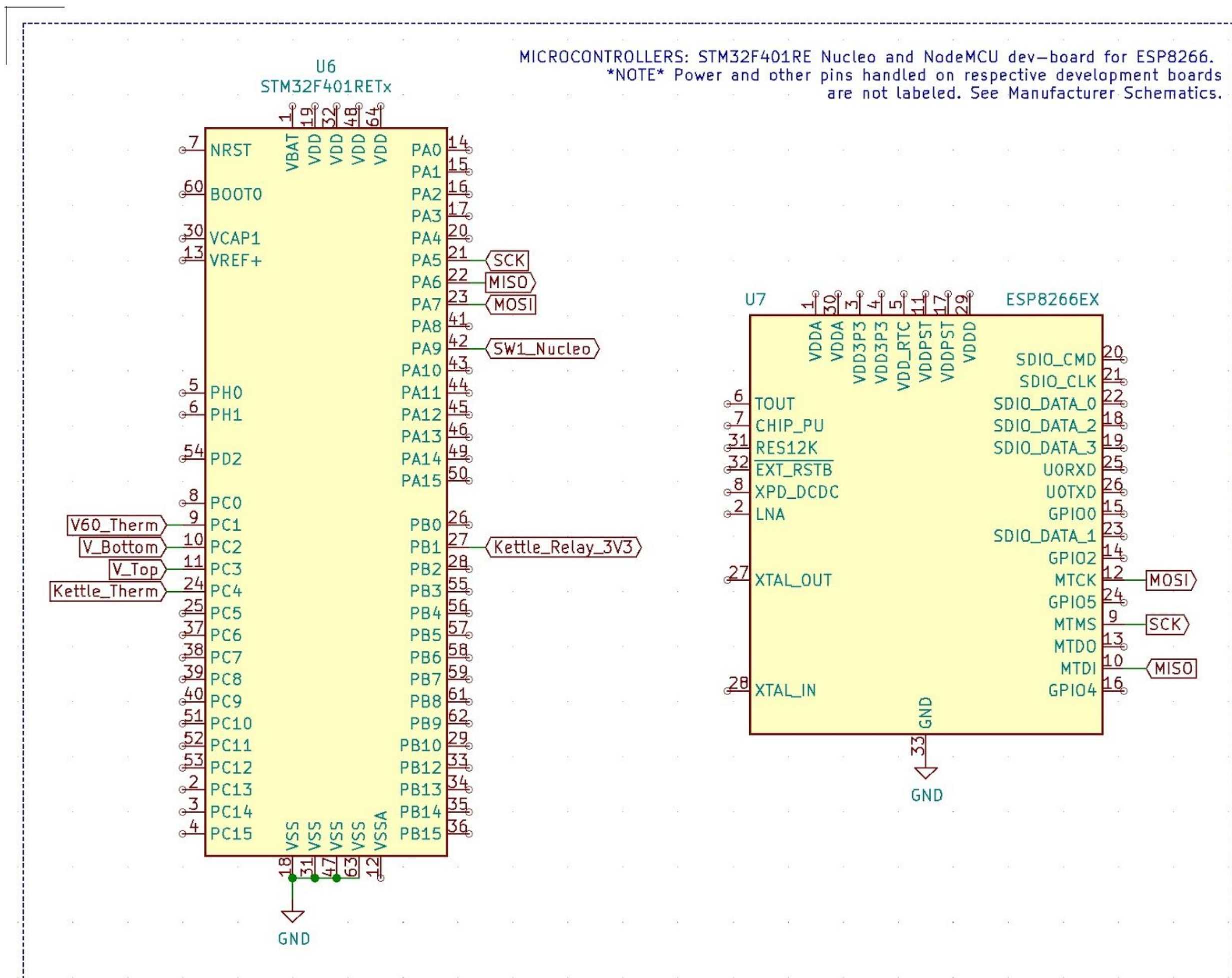
The frontend component handles the storage of current brew variables and the data required to create the graph over time. AJAX is used in order to create a responsive web-app that can plot live data and control the kettle without requiring page reloads. Using AJAX the web-app can generate asynchronous xHTTP requests to the NodeMCU server to begin routines on the STM32. The NodeMCU responds to the xHTTP request with the necessary data which then updates individual elements of the page without requiring a reload.

Storing the data for plotting on the browser side enables multiple users to connect to the same Better Barista simultaneously, and each user can control their graphing settings individually.

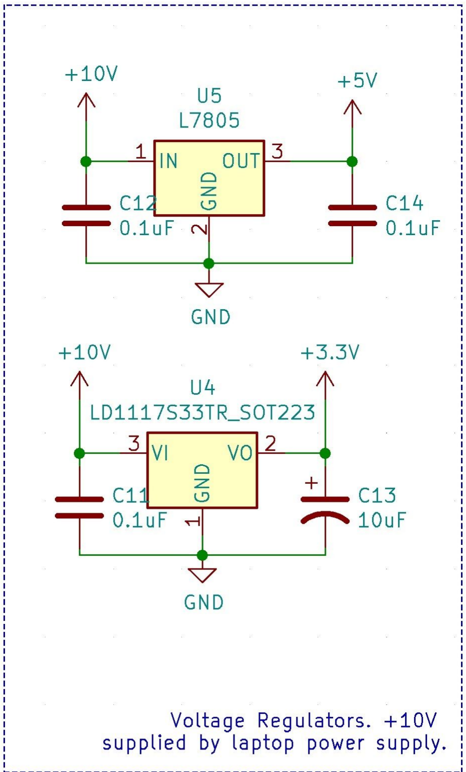
The backend component running on the NodeMCU receives xHTTP requests from the browser and generates corresponding SPI transmissions to the STM32. The ADC data received is then converted into the correct units and sent as a response to the xHTTP request.

Appendix A: Schematics

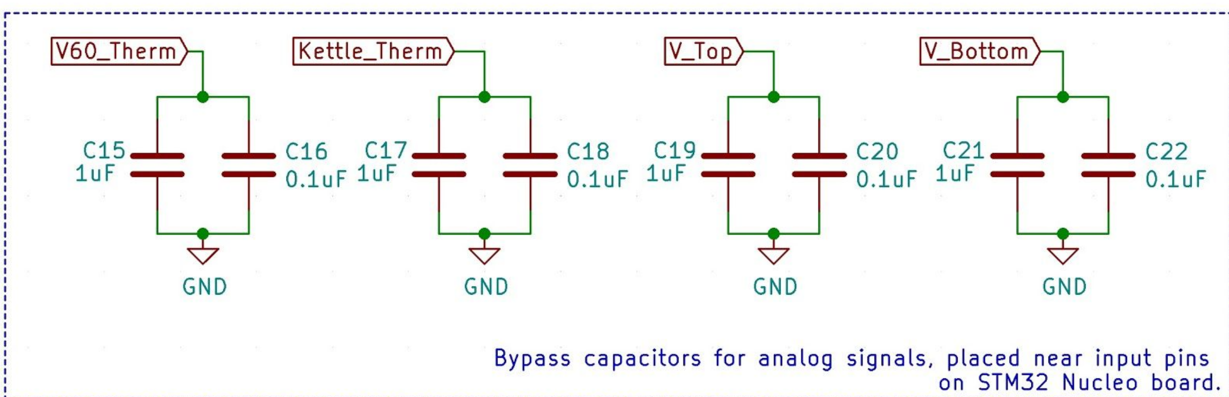
Microcontrollers



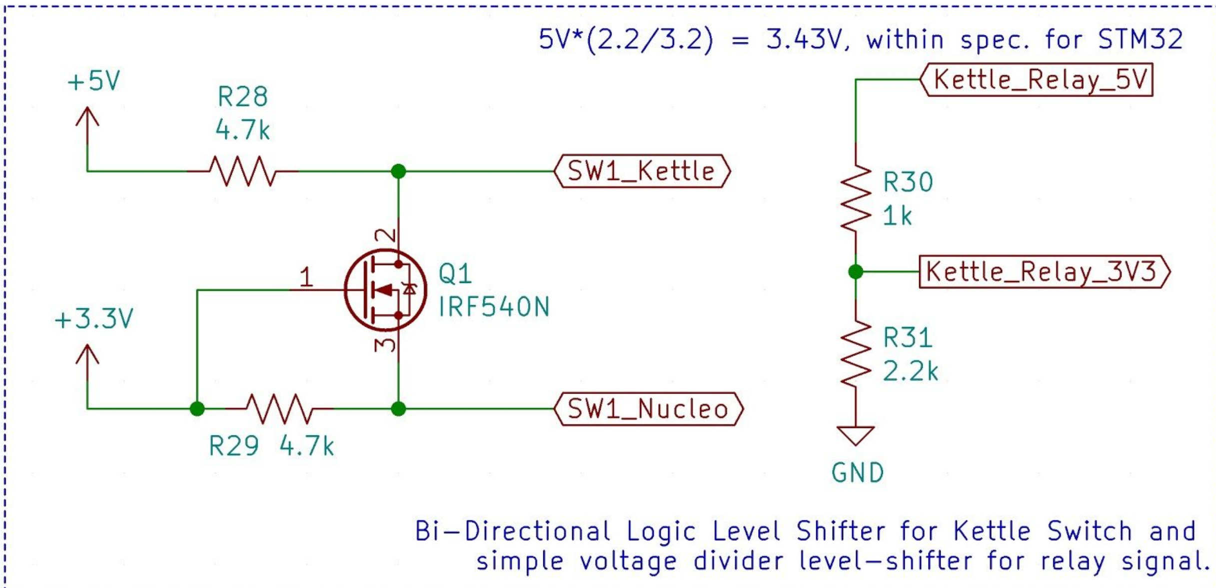
Voltage Regulators



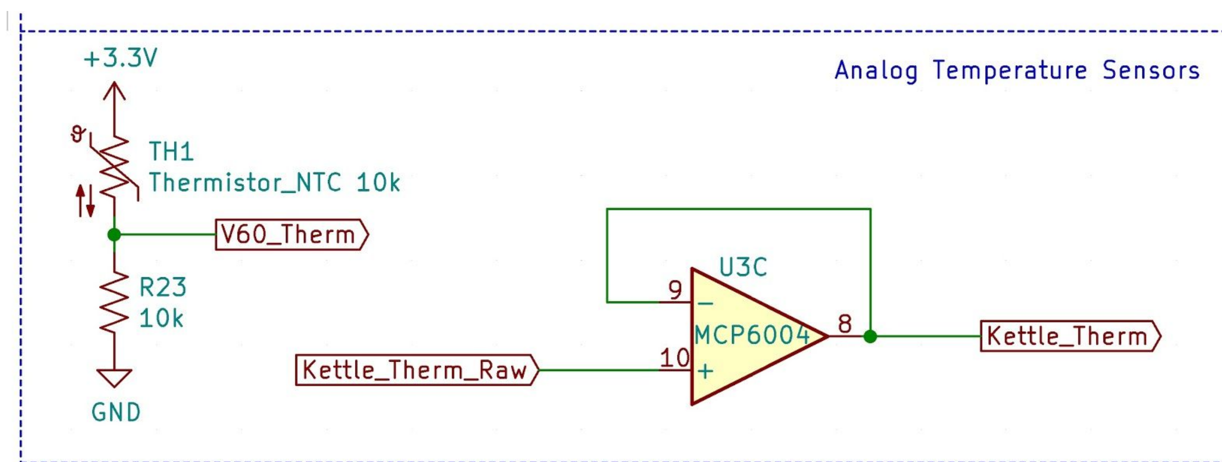
Bypass Capacitors



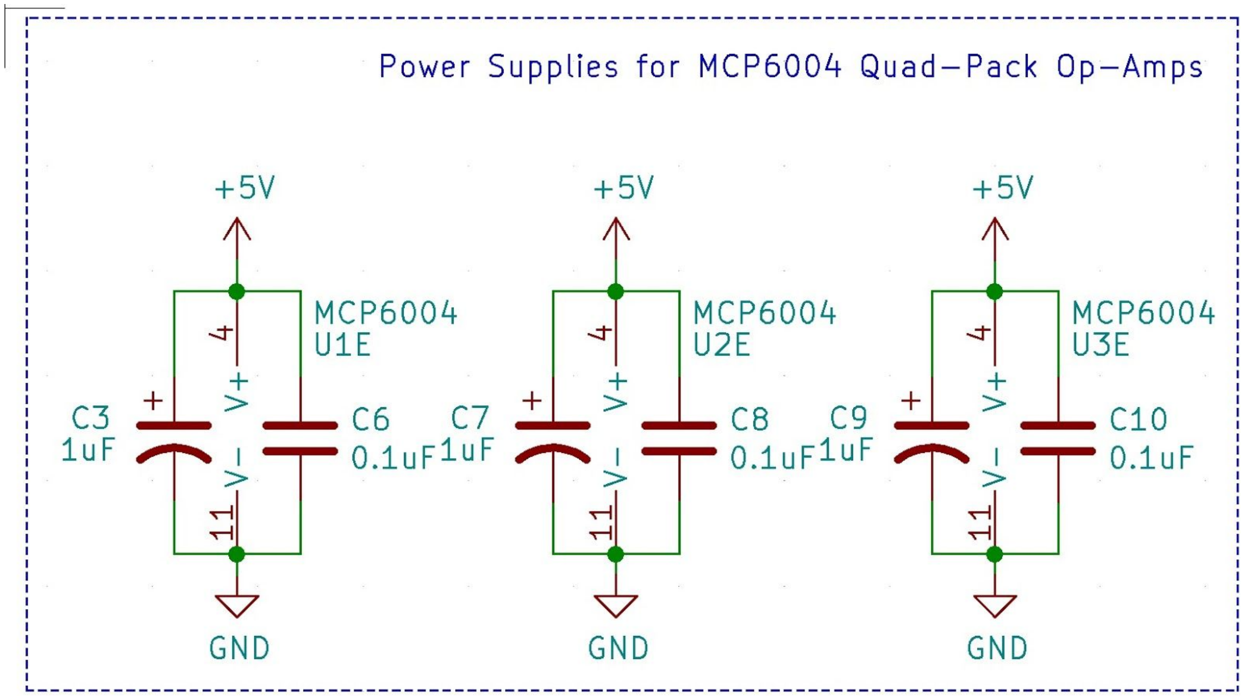
Logic Level Shifters



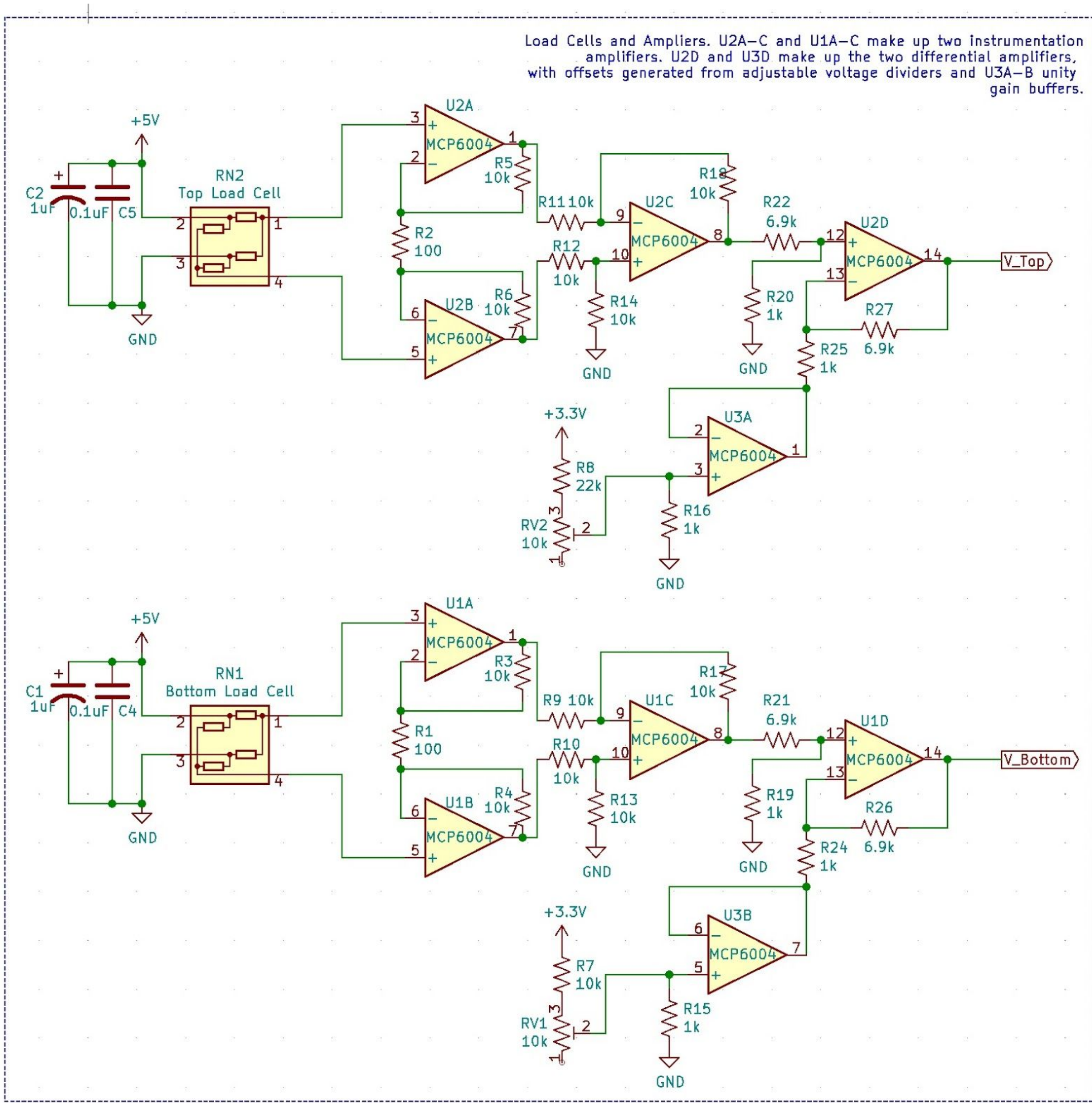
Analog Temperature Sensors



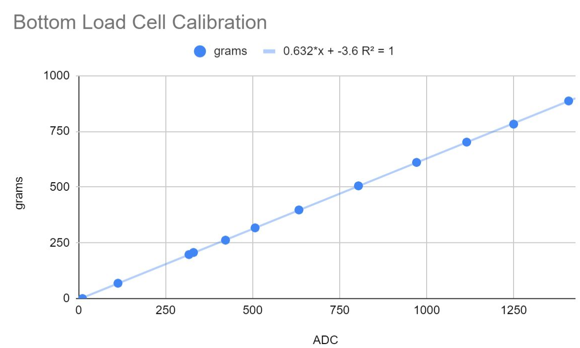
Op-amp power supplies

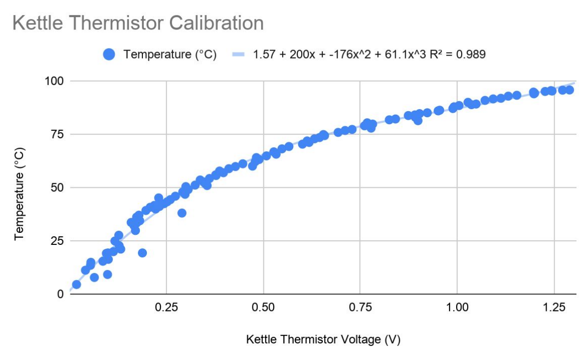


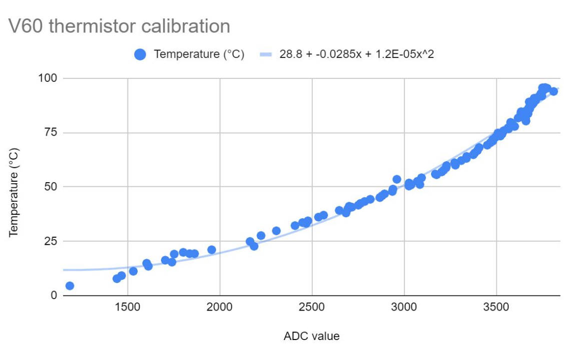
Load Cells and Amplifiers

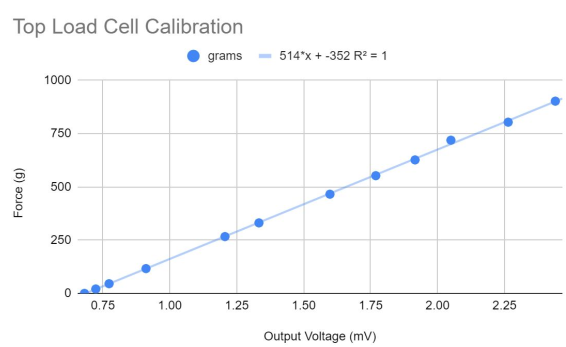


Appendix B: Sensor Calibration









Appendix C: Software

NodeMCU source code

```
/*
 * Node MCU Better Barista server
 * Polls STM for ADC values, converts units, and displays live graph
 * Updates and Gets data from webpage without page refresh
 */

#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ESP8266WebServer.h>
#include <SPI.h>

#include "index.h" //Our HTML webpage contents with javascripts

//SSID and Password of your WiFi router
const char* ssid = "NETGEAR53";
const char* password = "freshtuba566";

int g_kettle_temp;
int g_bottom_scale;
int g_top_scale;
int g_v60_temp;
int g_kettle_status;

ESP8266WebServer server(80); //Server on port 80

//===============================================================
// These routines are executed upon xhttp request
//===============================================================
void handleRoot() {
 String s = MAIN_page; //Read HTML contents
 server.send(200, "text/html", s); //Send web page
}

void handleADC() {
    // Send SPI commands to STM for all sensor reads
    uint16_t foo = sendSPICommand(8192);
    delay(90);
    g_kettle_temp = sendSPICommand(8193) & 4095;
    delay(90);
    g_bottom_scale = sendSPICommand(8194) & 4095;
    delay(90);
    g_top_scale = sendSPICommand(8195) & 4095;
    delay(90);
    g_v60_temp = sendSPICommand(0);
    delay(90);
    g_kettle_status = (g_v60_temp >> 14);
    g_v60_temp = (g_v60_temp & 4095);

    Serial.print("kettle_status: ");
    Serial.println(g_kettle_status);
```

```
    // Convert raw ADC values to grams
    float temp = 0.293*g_top_scale -13.6;
    g_top_scale = (int)temp;
    temp = 0.632*g_bottom_scale - 3.6;
    g_bottom_scale = (int)temp;

    // Convert raw ADC value to temperature
    temp = 28.8  - (0.0285 * g_v60_temp) + (0.000012 * g_v60_temp *
g_v60_temp);
    g_v60_temp = (int)temp;


    Serial.print("ADC return values: ");
    Serial.print(g_kettle_temp);
    Serial.print(", ");
    Serial.print(g_bottom_scale);
    Serial.print(", ");
    Serial.print(g_top_scale);
    Serial.print(", ");
    Serial.println(g_v60_temp);

    // Return sensor values as plain text via xhttp request
    String sensorData = String(String(g_kettle_temp) + " " +
String(g_bottom_scale) + " " + String(g_top_scale) + " "+ String(g_v60_temp) +
" " + String(g_kettle_status));
    server.send(200, "text/plane", sensorData); //Send ADC value only to client
ajax request
}

void handleKettle(){
    // Calculate ADC threshold value from input temperature
    float setTemp = server.arg("temp").toFloat();
    uint16_t setADC = (0.132 - 0.00512 * setTemp + 0.000171 * setTemp *
setTemp) *4095 / 3.3;

    // Send SPI command to STM to start kettle heating routine
    uint16_t foo = sendSPICommand((1 << 12) + setADC);
    delay(2000);
    Serial.print("Temperature set to: ");
    Serial.println(setTemp);

    server.send(200, "text/plane", String(foo)); // Return data from last SPI
request
}

//==============================================================
//                    SETUP
//==============================================================
void setup(void){
  Serial.begin(115200);
  SPI.begin();

  WiFi.begin(ssid, password);     //Connect to your WiFi router
  Serial.println("");

  // Wait for connection
```

```
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  //If connection successful show IP address in serial monitor
  Serial.println("");
  Serial.print("Connected to ");
  Serial.println(ssid);
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());  //IP address assigned to your ESP

  server.on("/", handleRoot);      //Routine to handle load page
  server.on("/readADC", handleADC); //Called via AJAX request by getData()
  server.on("/setKettle", handleKettle); // Called via AJAX request by
kettleControl()

  server.begin();                  //Start server
  Serial.println("HTTP server started");
}
//================================================================
//                     LOOP
//================================================================
void loop(void){
  server.handleClient();          //Handle client requests
}


//================================================================
//                     HELPER FUNCTIONS
//================================================================
uint16_t sendSPICommand(uint16_t command){
    // Opens transaction with STM
    uint16_t returnData;

    SPI.beginTransaction(SPISettings(400000, MSBFIRST, SPI_MODE0));
    returnData = SPI.transfer16(command);
    SPI.endTransaction();

    return returnData;
}
```

```
/*
 * index.h holds HTML, CSS, and js
 * for better barista web-app
*/

const char MAIN_page[] PROGMEM = R"=====(
<!doctype html>
<html>

<head>
  <title>Better barista</title>
  <script src =
"https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.7.3/Chart.min.js"></script>
  <style>
  canvas{
    -moz-user-select: none;
    -webkit-user-select: none;
    -ms-user-select: none;
  }
  .control_buttons{
      display: flex;
      justify-content: space-evenly;
      align-items: center;
  }

  #individual_weights{
      display: flex;
      flex-wrap: wrap;
      justify-content: space-evenly;
      align-items: center;
  }
  #bottom_weights{
      flex-grow: 1;
  }
  #top_weights{
        flex-grow:1;
  }
  #total_weights{
      flex-grow:2;
  }

  #total_weights * {
      text-align: center;
  }
  #bottom_weights * {
      text-align: center;
  }
  #top_weights * {
      text-align: center;
  }
  #user_set_temperature{
      text-align: center;
  }
  #kettle_buttons{
      display: flex;
      justify-content: space-evenly;
      align-items: center;
```

```
  }
  #kettle_form{
      margin: auto;
  }



  </style>
</head>

<body>
    <div style="text-align:center;"><p style="font-size: 3vw;
font:bold;">Better Barista Brew Buddy </p> <p style="font-size: 1.5vw;">Perfect
Pourover</p></div>
    <div class="control_buttons">
        <div id="kettle_controls">
            <p style="text-align: center; font-size: 1vw;"
id="kettle_status">Kettle is Off</p>
            <form id="kettle_form" onsubmit="return false;">
                <input type="number" id="user_set_temperature"
text-alignname="desired temperature" style="font-size: 2.5vw;" min="0"
max="105" value="95"><br>
                <div id="kettle_buttons">
                    <button id="kettle_on" onclick="kettleControl(1)">Start
Kettle</button>
                    <button id="kettle_off" onclick="kettleControl(0)">Kettle
OFF</button>
                </div>

            </form>
        </div>

        <div id="run_controls">
            <button id="start_stop" onclick="graphControl()"">START
BREW</button>
            <button id="clear_graph" onclick="clearGraph()">CLEAR
GRAPH</button>
        </div>

        <div id="scale_controls">
            <div id="total_weights">
                <p>Total Weight</p>
                <h1 id="total_scale_weight"> 0g</h2>
            </div>


            <div id="individual_weights">
                <div id="bottom_weights">
                    <p>Bottom Weight</p>
                    <h2 id="bottom_scale_weight"> 0g</h2>
                    <button id="tare_bot" onclick="scaleControl(0)">TARE
BOTTOM</button>
                </div>
                <div id="top_weights">
                    <p>Top Weight</p>
                    <h2 id="top_scale_weight"> 0g</h2>
```

```
                    <button id="tare_top" onclick="scaleControl(1)">TARE
TOP</button>
                </div>

            </div>

        </div>
    </div>

    <div class="chart-container" position: relative; height:500px; width:100%">
        <canvas id="weightChart" width="400" height="400"></canvas>
    </div>
    <div class="chart-container" position: relative; height:500px; width:100%">
        <canvas id="tempChart" width="400" height="400"></canvas>
    </div>
<br>
<br>

<script>
//Graphs visit: https://www.chartjs.org

var cupWeightArr = [];
var funnelWeightArr = []
var totalWeightArr = [];
var funnelTempArr = [];
var timeStampArr = [];
var kettleTemp = 0;

var topScale = 0;
var botScale = 0;
var topTare = 0;
var botTare = 0;

var kettleStatus = 0; // 0 off, 1 heating, 2 at temp, 3 lifted
var runStatus = 0; // 1: run graph, 0: stop graph

function showGraphs()
{
    var ctx1 = document.getElementById("weightChart").getContext('2d');
    var brewWeightChart = new Chart(ctx1, {
        type: 'line',
        data: {
            labels: timeStampArr,  //Bottom Labeling
            datasets: [{
                label: "Total Weight",
                fill: false,  //Try with true
                backgroundColor: 'rgba( 243, 156, 18 , 1)', //Dot marker color
                borderColor: 'rgba( 243, 156, 18 , 1)', //Graph Line Color
                data: totalWeightArr,
            },{
                label: "Cup Weight",
                fill: false,
                backgroundColor: 'rgba( 18, 156, 243, 1)',
                borderColor: 'rgba( 18, 156, 243, .5)',
                data: cupWeightArr,
            },{
                label: "Fun Weight",
```

```
                fill: false,
                backgroundColor: 'rgba( 18, 243, 156, 1)',
                borderColor: 'rgba( 18, 243, 156, 1)',
                data: funnelWeightArr,
            }],
        },
        options: {
            title: {
                    display: true,
                    text: "Brew Profile"
                },
            maintainAspectRatio: false,
            elements: {
            line: {
                    tension: 0.4 //Smoothening (Curved) of data lines
                }
            },
            scales: {
                    yAxes: [{
                        ticks: {
                            beginAtZero:true
                        }

                    }]
            },
            animation: {
            duration: 0 // general animation time
            }
        }
    });

    var ctx2 = document.getElementById("tempChart").getContext('2d');
    var tempChart = new Chart(ctx2, {
        type: 'line',
        data: {
            labels: timeStampArr,  //Bottom Labeling
            datasets: [{
                label: "Slurry Temp",
                fill: false,  //Try with true
                backgroundColor: 'rgba( 243, 156, 18 , 1)', //Dot marker color
                borderColor: 'rgba( 243, 156, 18 , 1)', //Graph Line Color
                data: funnelTempArr,
            }],
        },
        options: {
            title: {
                    display: true,
                    text: "Temperature Profile"
                },
            maintainAspectRatio: false,
            elements: {
            line: {
                    tension: 0.4 //Smoothening (Curved) of data lines
                }
            },
            scales: {
                    yAxes: [{
```

```
                        ticks: {
                            beginAtZero:true
                        }
                    }]
            },
            animation: {
            duration: 0 // general animation time
            }
        }
    });

}

//On Page load show graphs
window.onload = function() {
    console.log(new Date().toLocaleTimeString());
    showGraphs();
};

//Ajax script to get sensor reads every 500 ms

setInterval(function() {
    getData();
}, 500); //500mSeconds update rate

function getData() {
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            //Push the data in array
            var dataArrStr = this.responseText;
            var dataArr = dataArrStr.split(" ");
            var time = new Date().toLocaleTimeString();

            // Calculate display weights
            topScale = parseInt(dataArr[2]) - topTare;
            botScale = parseInt(dataArr[1]) - botTare;
            kettleTemp = parseInt(dataArr[0]);

            // Display weights
            document.getElementById("total_scale_weight").innerText = (topScale
+ botScale).toString() + "g";
            document.getElementById("bottom_scale_weight").innerText =
botScale.toString() + "g";
            document.getElementById("top_scale_weight").innerText =
topScale.toString() + "g";

            // Update kettle status
            kettleStatus = parseInt(dataArr[4]);
            switch(kettleStatus){
                case 0:
                    document.getElementById("kettle_status").innerHTML =
"Kettle is Off";
                    console.log("Kettle Off")
                    break;
                case 1:
```

```
                    document.getElementById("kettle_status").innerHTML =
"Kettle Heating";
                    console.log("Kettle is heaitng")
                    break;
                case 2:
                    document.getElementById("kettle_status").innerHTML =
"Kettle is Done";
                    console.log("Kettle is Done")
                    break;
                case 3:
                    document.getElementById("kettle_status").innerHTML =
"Kettle Lifted";
                    console.log("Kettle is Lifted")
                    graphControl();
                    break;

            }


            // Only add data if runStatus is true
            if(runStatus){
                console.log("case0");
                cupWeightArr.push(parseInt(dataArr[1]) - botTare);
                totalWeightArr.push(parseInt(dataArr[1]) + parseInt(dataArr[2])
- topTare - botTare);
                funnelWeightArr.push(parseInt(dataArr[2]) - topTare);
                funnelTempArr.push(parseInt(dataArr[3]));

                timeStampArr.push(time);
            }

            showGraphs();  //Update Graphs;
        }
    };
    xhttp.open("GET", "readADC", true); //Open request for sensor values
    xhttp.send();
}

function scaleControl(scale_id){
    // Function to tare scales
    console.log("scale Controlled");
    if(scale_id == 1){
        topTare = topScale + topTare;
    }
    else {
        botTare = botScale + botTare;
    }
}

function graphControl() {
    // Function to start/stop graphing
    console.log("graph Controlled");
    // Sets run status based on which button was clicked
    if (runStatus == 0){
        runStatus = 1;
        document.getElementById("start_stop").innerText = "STOP BREW";
    } else {
```

```
        runStatus = 0
        document.getElementById("start_stop").innerText = "START BREW";
    }
    console.log(runStatus);
}

function clearGraph() {
    console.log("cleared data");
    cupWeightArr = [];
    funnelWeightArr = []
    totalWeightArr = [];
    funnelTempArr = [];
    timeStampArr = [];
}


function kettleControl(reqtype) {
    console.log("Kettle Controled");
    var xhttp = new XMLHttpRequest();
    var temp;
    xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
        }
    };

    if (reqtype) {
        temp = document.getElementById("user_set_temperature").value;
        console.log("Set to");
    } else {
        temp = 0;
    }
    xhttp.open("GET", "setKettle?temp="+temp, true);
    xhttp.send();
}

</script>
</body>

</html>

)=====";
```

STM32F401RE Source Code

Main.c

```
// main.c
// Tejus Rao & Samin Zachariah
// Controls better barista hardware

#include "STM32F401RE.h"
#include <string.h>
#include <stdio.h>
#include <math.h>
#include "main.h"

#define KETTLE_OFF 0b00
#define KETTLE_HEATING 0b01
#define KETTLE_AT_TEMP 0b10
#define KETTLE_LIFTED 0b11

#define KETTLE_ID 0b00
#define BOTTOM_CELL_ID 0b01
#define TOP_CELL_ID 0b10
#define V60_ID 0b11

#define USART_ID USART2_ID
#define TEMP_SET 70
#define KETTLE_SWITCH 9
#define KETTLE_RELAY 1 //PB1

volatile uint8_t kettle_set_temp = 0;
volatile uint8_t g_kettle_status = 0; // 0b00 - off, 0b01 - heating, 0b10 - at
temp, 0b11 - lifted

uint16_t g_bottom_scale;
uint16_t g_top_scale;


uint16_t adcAverage(uint16_t senseID){

    if(senseID == 0 || senseID == 3){
        int sum = 0;
        for(int i = 0; i < 20; i++){
            sum += ADCRead(senseID);
            delay_millis(TIM2, 1);
        }
        return sum/20;
    }
    else{
        int total = 0;
        for(int i = 0; i < 20; i++){
            int sum = 0;
            for(int i = 0; i < 100; i++){
                sum += ADCRead(senseID);
                delay_micros(TIM2, 10);
            }
            total += sum / 100;
```

```c
            delay_millis(TIM2, 1);
        }
        return total/20;
    }
}

void kettle_switch(){
    digitalWrite(GPIOA, KETTLE_SWITCH, 0);
    delay_millis(TIM2, 300);
    digitalWrite(GPIOA, KETTLE_SWITCH, 1);
}

void kettle_on_off(uint8_t bool){
    if(digitalRead(GPIOB, KETTLE_RELAY) != bool){
        kettle_switch();
    }
}

uint16_t kettleSetTemp(uint16_t tempSet){
    /**
     * Sets Analog Watchdog threshold to temperature corresponding to tempSet
     * Output: current kettle temperature */
    if(tempSet == 0){
        g_kettle_status = KETTLE_OFF;
        kettle_on_off(0);
    }
    else{
        // TODO: kettle calibration math
        ADC1->ADC_HTR = tempSet;
        g_kettle_status = KETTLE_HEATING;
        kettle_on_off(1);
        uint8_t msg[64];
        sprintf(msg, "Kettle got set to: %d", tempSet);
        serialPrintUSART(msg);
    }
    return ADCRead(KETTLE_ID);
}

int main(void) {
    configureFlash();
    configureClock();
    initUSART(USART2_ID);

    RCC->AHB1ENR.GPIOAEN = 1; // Enable GPIOA
    RCC->AHB1ENR.GPIOCEN = 1; // Enable GPIOC
    RCC->AHB1ENR.GPIOBEN = 1;

    RCC->APB1ENR |= (1 << 0); // TIM2EN
    RCC->APB2ENR |= (1 << 14); //Enables SYSCFG

    ADC_Init();

    initTIM(TIM2);

    pinMode(GPIOB, KETTLE_RELAY, GPIO_INPUT,0,0,0b00); // Kettle Relay
    pinMode(GPIOA, KETTLE_SWITCH, GPIO_OUTPUT, 1, 0b11, 0b01); // Kettle Switch
    delay_millis(TIM2, 200);
```

```
    kettle_on_off(0);

    spiInitSlave(0, 0);

        __enable_irq();
    NVIC_EnableIRQ(SPI1_IRQn);
    NVIC_EnableIRQ(ADC_IRQn);

    while(1){
        __NOP();
    }
}

void SPI1_IRQHandler(){
    while(!SPI1->SR.RXNE);  // Wait for end of transmission

    uint16_t fromNode = SPI1->DR.DR; // Read incoming transmission - Clears
interrupt flag
    uint8_t msg[64];

    // Parse command from node & Execute resulting ADC / kettle command
    uint16_t instr_type = (fromNode >> 12) & 0b11;
    uint16_t instr_arg = fromNode & 4095;
    uint16_t toNode = 0x00;
    uint16_t sensor_data = 0;
    switch(instr_type){
        case 0b01:
            toNode = (g_kettle_status << 14) + (instr_arg << 12) +
kettleSetTemp(instr_arg);
            break;
        case 0b10: ;
            sensor_data = adcAverage(instr_arg);
            switch (instr_arg){
                case 0:
                    sprintf(msg, "Kettle: %d, Status: %d ", sensor_data,
g_kettle_status);
                    break;
                case 1:
                    sprintf(msg, "Bottom: %d, ", sensor_data);
                    break;
                case 2:
                    sprintf(msg, "Top: %d, ", sensor_data);
                    break;
                case 3:
                    sprintf(msg, "V60: %d \n\r", sensor_data);
                    break;
            }

            toNode = (g_kettle_status << 14) + (instr_arg << 12) + sensor_data;
            break;

        default:
                __NOP();
                break;
    }

    serialPrintUSART(msg);
```

```
    SPI1->DR.DR = toNode; // result write to data register - will be
transferred out on next transaction
}

void ADC_IRQHandler(){
    while(!ADC1->ADC_SR.AWD);

    uint16_t kettleVal = ADCRead(KETTLE_ID);

    if(kettleVal >= 2000){
        g_kettle_status = KETTLE_LIFTED;
    }
    else{
        g_kettle_status = KETTLE_AT_TEMP;
        kettle_on_off(0);
    }

    uint8_t msg[64];
    sprintf(msg, "Kettle HELLO: %d\n\r", ADCRead(KETTLE_ID));

    serialPrintUSART(msg);

    ADC1->ADC_HTR = 4095; // Reset high threshold to avoid constant interrupts
    ADC1->ADC_SR.AWD = 0; // Clear interrupt flag

    return;
}
```

SPI Library

```c
// STM32F401RE_SPI.h
// Header for SPI functions

#ifndef STM32F4_SPI_H
#define STM32F4_SPI_H

#include <stdint.h> // Include stdint header

///////////////////////////////////////////////////////////////////////////////
// Definitions
///////////////////////////////////////////////////////////////////////////////

#define SPI1_BASE (0x40013000UL)
#define __IO volatile

///////////////////////////////////////////////////////////////////////////////
// Bitfield structs
///////////////////////////////////////////////////////////////////////////////

typedef struct {
  __IO uint32_t CPHA        : 1;
  __IO uint32_t CPOL        : 1;
  __IO uint32_t MSTR        : 1;
  __IO uint32_t BR          : 3;
  __IO uint32_t SPE         : 1;
  __IO uint32_t LSBFIRST    : 1;
  __IO uint32_t SSI         : 1;
  __IO uint32_t SSM         : 1;
  __IO uint32_t RXONLY      : 1;
  __IO uint32_t DFF         : 1;
  __IO uint32_t CRCNEXT     : 1;
  __IO uint32_t CRCEN       : 1;
  __IO uint32_t BIDIOE      : 1;
  __IO uint32_t BIDIMODE    : 1;
  __IO uint32_t             : 16;
} SPI_CR1_bits;

typedef struct {
  __IO uint32_t RXDMAEN     : 1;
  __IO uint32_t TXDMAEN     : 1;
  __IO uint32_t SSOE        : 1;
  __IO uint32_t             : 1;
  __IO uint32_t FRF         : 1;
  __IO uint32_t ERRIE       : 1;
  __IO uint32_t RXNEIE      : 1;
  __IO uint32_t TXEIE       : 1;
  __IO uint32_t             : 24;
} SPI_CR2_bits;

typedef struct {
  __IO uint32_t RXNE        : 1;
  __IO uint32_t TXE         : 1;
  __IO uint32_t CHSIDE      : 1;
  __IO uint32_t UDR         : 1;
  __IO uint32_t CRCERR      : 1;
```

```
  __IO uint32_t MODF        : 1;
  __IO uint32_t OVR         : 1;
  __IO uint32_t BSY         : 1;
  __IO uint32_t FRE         : 1;
  __IO uint32_t DFF         : 1;
  __IO uint32_t CRCNEXT     : 1;
  __IO uint32_t CRCEN       : 1;
  __IO uint32_t BIDIOE      : 1;
  __IO uint32_t BIDIMODE    : 1;
  __IO uint32_t             : 16;
} SPI_SR_bits;

typedef struct {
  __IO uint32_t DR  : 16;
  __IO uint32_t     : 16;
} SPI_DR_bits;


typedef struct {
  __IO SPI_CR1_bits CR1;        /*!< SPI control register 1 (not used in I2S
mode),      Address offset: 0x00 */
  __IO SPI_CR2_bits CR2;        /*!< SPI control register 2,
Address offset: 0x04 */
  __IO SPI_SR_bits SR;         /*!< SPI status register,
Address offset: 0x08 */
  __IO SPI_DR_bits DR;         /*!< SPI data register,
Address offset: 0x0C */
  __IO uint32_t CRCPR;      /*!< SPI CRC polynomial register (not used in I2S
mode), Address offset: 0x10 */
  __IO uint32_t RXCRCR;     /*!< SPI RX CRC register (not used in I2S mode),
Address offset: 0x14 */
  __IO uint32_t TXCRCR;     /*!< SPI TX CRC register (not used in I2S mode),
Address offset: 0x18 */
  __IO uint32_t I2SCFGR;    /*!< SPI_I2S configuration register,
Address offset: 0x1C */
  __IO uint32_t I2SPR;      /*!< SPI_I2S prescaler register,
Address offset: 0x20 */
} SPI_TypeDef;

// Pointers to GPIO-sized chunks of memory for each peripheral
#define SPI1 ((SPI_TypeDef *) SPI1_BASE)

///////////////////////////////////////////////////////////////////////////////
// Function prototypes
///////////////////////////////////////////////////////////////////////////////

/* Enables the SPI peripheral and intializes its clock speed (baud rate),
polarity, and phase.
 *    -- clkdivide: (0x01 to 0xFF). The SPI clk will be the master clock /
clkdivide.
 *    -- cpol: clock polarity (0: inactive state is logical 0, 1: inactive
state is logical 1).
 *    -- cpha: clock phase (1: data changed on leading edge of clk and captured
on next edge,
 *          0: data captured on leading edge of clk and changed on next edge)
 * Note: the SPI mode register is set with the following unadjustable settings:
 *    -- Master mode
```

```
 *    -- Fixed peripheral select
 *    -- Chip select lines directly connected to peripheral device
 *    -- Mode fault detection enabled
 *    -- WDRBT disabled
 *    -- LLB disabled
 *    -- PCS = 0000 (Peripheral 0 selected), means NPCS[3:0] = 1110
 * Refer to the datasheet for more low-level details. */
void spiInitMaster(uint32_t clkdivide, uint32_t cpol, uint32_t ncpha);

void spiInitSlave(uint32_t cpol, uint32_t cpha);

/* Transmits a character (1 byte) over SPI and returns the received character.
 *    -- send: the character to send over SPI
 *    -- return: the character received over SPI */
uint8_t spiSendReceive(uint8_t send);

/* Transmits a short (2 bytes) over SPI and returns the received short.
 *    -- send: the short to send over SPI
 *    -- return: the short received over SPI */
uint16_t spiSendReceive16(uint16_t send);



#endif
```

```c
// STM32F401RE_SPI.c
// SPI function declarations

#include "STM32F401RE_SPI.h"
#include "STM32F401RE_RCC.h"
#include "STM32F401RE_GPIO.h"

/* Enables the SPI peripheral and intializes its clock speed (baud rate),
polarity, and phase.
 *    -- br: (0b000 - 0b111). The SPI clk will be the master clock / 2^(BR+1).
 *    -- cpol: clock polarity (0: inactive state is logical 0, 1: inactive
state is logical 1).
 *    -- ncpha: clock phase (0: data changed on leading edge of clk and
captured on next edge,
 *          1: data captured on leading edge of clk and changed on next edge)
 * Note: the SPI mode register is set with the following unadjustable settings:
 *    -- Master mode
 *    -- Fixed peripheral select
 *    -- Chip select lines directly connected to peripheral device
 *    -- Mode fault detection enabled
 *    -- WDRBT disabled
 *    -- LLB disabled
 *    -- PCS = 0000 (Peripheral 0 selected), means NPCS[3:0] = 1110
 * Refer to the datasheet for more low-level details. */
void spiInitMaster(uint32_t br, uint32_t cpol, uint32_t cpha) {
    // Turn on GPIOA and GPIOB clock domains (GPIOAEN and GPIOBEN bits in
AHB1ENR)
    RCC->AHB1ENR.GPIOAEN = 1;
    RCC->AHB1ENR.GPIOBEN = 1;

    RCC->APB2ENR |= (1 << 12); // Turn on SPI1 clock domain (SPI1EN bit in
APB2ENR)

    // Initially assigning SPI pins
    pinMode(GPIOA, 5, GPIO_ALT,0,0,0); // PA5, Arduino D13, SPI1_SCK
    pinMode(GPIOA, 7, GPIO_ALT,0,0,0); // PA7, Arduino D11, SPI1_MOSI
    pinMode(GPIOA, 4, GPIO_ALT,0,0,0); // PA4, Arduino A2, SPI1_NSS
    pinMode(GPIOB, 6, GPIO_OUTPUT,0,0,0); // PB6, Arduino D10, Manual CS

    // Set to AF05 for SPI alternate functions
    GPIOA->AFRL |= (1 << 22) | (1 << 20);
    GPIOA->AFRL |= (1 << 30) | (1 << 28);
    GPIOA->AFRL |= (1 << 18) | (1 << 16);

    SPI1->CR1.BR = br;       // Set the clock divisor
    SPI1->CR1.CPOL = cpol;   // Set the polarity
    SPI1->CR1.CPHA = cpha;   // Set the phase
    SPI1->CR1.LSBFIRST = 0; // Set least significant bit first
    SPI1->CR1.DFF = 1;       // Set data format to 16 bits
    SPI1->CR1.SSM = 0;       // Turn off software slave management
    SPI1->CR2.SSOE = 1;      // Set the NSS pin to output mode
    SPI1->CR1.MSTR = 1;      // Put SPI in master mode
    SPI1->CR1.SPE = 1;       // Enable SPI
}

void spiInitSlave(uint32_t cpol, uint32_t cpha){
```

```
    RCC->AHB1ENR.GPIOAEN = 1;
    RCC->AHB1ENR.GPIOBEN = 1;
    RCC->APB2ENR |= (1 << 12); // Turn on SPI1 clock domain (SPI1EN bit in
APB2ENR)

    pinMode(GPIOA, 5, GPIO_ALT,0,0,0); // PA5, Arduino D13, SPI1_SCK
    pinMode(GPIOA, 7, GPIO_ALT,0,0b11,0b10); // PA7, Arduino D11, SPI1_MOSI
    pinMode(GPIOA, 6, GPIO_ALT, 0,0b11,0b00); //PA6, Arduino D12, SPI1, MISO
    // pinMode(GPIOA, 4, GPIO_ALT,0,0,0); // PA4, Arduino A2, SPI1_NSS
    // pinMode(GPIOB, 6, GPIO_OUTPUT,0,0,0); // PB6, Arduino D10, Manual CS


    // Set to AF05 for SPI alternate functions
    GPIOA->AFRL |= (1 << 22) | (1 << 20);
    GPIOA->AFRL |= (1 << 30) | (1 << 28);
    GPIOA->AFRL |= (0b0101 << 24);
   // GPIOA->AFRL |= (1 << 18) | (1 << 16);

    SPI1->CR1.SPE = 0;       // nable SPI
    SPI1->CR1.CPOL = cpol;  // Set the polarity
    SPI1->CR1.CPHA = cpha;  // Set the phase
    SPI1->CR1.LSBFIRST = 0; // Set most significant bit first
    SPI1->CR1.DFF = 1;       // Set data format to 16 bits
    SPI1->CR1.SSM = 0;       // Turn on software slave management
    SPI1->CR1.MSTR = 0;      // Put SPI in slave mode
    SPI1->CR2.TXEIE = 1;
    SPI1->CR1.SPE = 1;       // Enable SPI
}

// /* Transmits a character (1 byte) over SPI and returns the received
character.
//  *    -- send: the character to send over SPI
//  *    -- return: the character received over SPI */
// uint8_t spiSendReceive(uint8_t send) {
//     SPI1->DR.DR = send; // Transmit the character over SPI
//     while (!(SPI->SPI_SR.RDRF)); // Wait until data has been received
//     return (char) (SPI->SPI_RDR.RD); // Return received character
// }

/* Transmits a short (2 bytes) over SPI and returns the received short.
 *    -- send: the short to send over SPI
 *    -- return: the short received over SPI */
uint16_t spiSendReceive16(uint16_t send) {
    if(SPI1->CR1.MSTR){
        digitalWrite(GPIOB, 6, 0);
    }

    SPI1->CR1.SPE = 1;
    SPI1->DR.DR = send;

    while(!(SPI1->SR.RXNE));
    uint16_t rec = SPI1->DR.DR;

   SPI1->CR1.SPE = 0;

    if(SPI1->CR1.MSTR){
        digitalWrite(GPIOB, 6, 1);
```

```
    }

    return rec;
}

uint8_t spiSendReceive8(uint8_t send) {
    if(SPI1->CR1.MSTR){
        digitalWrite(GPIOB, 6, 0);
    }

    SPI1->CR1.SPE = 1;
    SPI1->DR.DR = send;

    while(!(SPI1->SR.RXNE));
    uint8_t rec = SPI1->DR.DR;

   // SPI1->CR1.SPE = 0;

    if(SPI1->CR1.MSTR){
        digitalWrite(GPIOB, 6, 1);
    }

    return rec;
}
```

ADC Library

```c
// STM32F401RE_ADC.h
// Header file for ADC functions
#ifndef STM32F4_ADC_H
#define STM32F4_ADC_H

#include <stdint.h>
#include "STM32F401RE_RCC.h"
#include "STM32F401RE_GPIO.h"


#define ADC1_BASE (0x40012000UL)
#define ADC1_CCR_ADDR (0x40012304UL)

typedef struct {
  volatile uint32_t AWD      : 1;
  volatile uint32_t EOC      : 1;
  volatile uint32_t JEOC     : 1;
  volatile uint32_t JSTRT    : 1;
  volatile uint32_t STRT     : 1;
  volatile uint32_t OVR      : 1;
  volatile uint32_t          : 26;
} ADC_SR_bits;

typedef struct {
  volatile uint32_t AWDCH    : 5;
  volatile uint32_t EOCIE    : 1;
  volatile uint32_t AWDIE    : 1;
  volatile uint32_t JEOCIE   : 1;
  volatile uint32_t SCAN     : 1;
  volatile uint32_t AWDSGL   : 1;
  volatile uint32_t JAUTO    : 1;
  volatile uint32_t DISCEN   : 1;
  volatile uint32_t JDISCEN  : 1;
  volatile uint32_t DISC_NUM: 3;
  volatile uint32_t          : 6;
  volatile uint32_t JAWDEN   : 1;
  volatile uint32_t AWDEN    : 1;
  volatile uint32_t RES      : 2;
  volatile uint32_t OVRIE    : 1;
  volatile uint32_t          : 5;
} ADC_CR1_bits;

typedef struct {
  volatile uint32_t ADON     : 1;
  volatile uint32_t CONT     : 1;
  volatile uint32_t          : 6;
  volatile uint32_t DMA      : 1;
  volatile uint32_t DDS      : 1;
  volatile uint32_t EOCS     : 1;
  volatile uint32_t ALIGN    : 1;
  volatile uint32_t          : 4;
  volatile uint32_t JEXTSEL : 4;
  volatile uint32_t JEXTEN   : 2;
  volatile uint32_t JSWSTART: 1;
  volatile uint32_t          : 1;
```

```c
  volatile uint32_t EXTSEL  : 4;
  volatile uint32_t EXTEN   : 2;
  volatile uint32_t SWSTART : 1;
  volatile uint32_t         : 1;
} ADC_CR2_bits;

typedef struct {
  volatile uint32_t SQ13   : 5;
  volatile uint32_t SQ14   : 5;
  volatile uint32_t SQ15   : 5;
  volatile uint32_t SQ16   : 5;
  volatile uint32_t L      : 4;
  volatile uint32_t        : 8;
} ADC_SQR1_bits;

typedef struct {
  volatile uint32_t SQ7   : 5;
  volatile uint32_t SQ8   : 5;
  volatile uint32_t SQ9   : 5;
  volatile uint32_t SQ10  : 5;
  volatile uint32_t SQ11  : 5;
  volatile uint32_t SQ12  : 5;
  volatile uint32_t       : 2;
} ADC_SQR2_bits;

typedef struct {
  volatile uint32_t SQ1   : 5;
  volatile uint32_t SQ2   : 5;
  volatile uint32_t SQ3   : 5;
  volatile uint32_t SQ4   : 5;
  volatile uint32_t SQ5   : 5;
  volatile uint32_t SQ6   : 5;
  volatile uint32_t       : 2;
} ADC_SQR3_bits;

typedef struct{
    volatile uint32_t DATA  : 16;
    volatile uint32_t       : 16;
} ADC_DR_bits;

typedef struct {
    volatile uint32_t            : 16;
    volatile uint32_t ADCPRE     : 2;
    volatile uint32_t            : 4;
    volatile uint32_t VBATE      : 1;
    volatile uint32_t TSVREFE    : 1;
    volatile uint32_t            : 8;
} ADC_CCR_bits;

typedef struct {
    volatile ADC_SR_bits ADC_SR;
    volatile ADC_CR1_bits ADC_CR1;
    volatile ADC_CR2_bits ADC_CR2;
    volatile uint32_t ADC_SMPR1;
    volatile uint32_t ADC_SMPR2;
    volatile uint32_t ADC_JOFR1;
    volatile uint32_t ADC_JOFR2;
```

```
    volatile uint32_t ADC_JOFR3;
    volatile uint32_t ADC_JOFR4;
    volatile uint32_t ADC_HTR;
    volatile uint32_t ADC_LTR;
    volatile ADC_SQR1_bits ADC_SQR1;
    volatile ADC_SQR2_bits ADC_SQR2;
    volatile ADC_SQR3_bits ADC_SQR3;
    volatile uint32_t ADC_JSQR;
    volatile uint32_t ADC_JDR1;
    volatile uint32_t ADC_JDR2;
    volatile uint32_t ADC_JDR3;
    volatile uint32_t ADC_JDR4;
    volatile ADC_DR_bits ADC_DR;
} ADC_TypeDef;

typedef struct {
    volatile ADC_CCR_bits ADC_CCR;
} ADC_CCR_TypeDef;

#define ADC1 ((ADC_TypeDef *) ADC1_BASE)
#define ADC1_CCR ((ADC_CCR_TypeDef *) ADC1_CCR_ADDR)

void ADC_Init();
uint16_t ADCRead(uint16_t sense_ID); // Reads PC0,1,2,3

#endif
```

```
// STM32F401RE_ADC
// Source code for ADC functions

#include "STM32F401RE_ADC.h"
#include "STM32F401RE_RCC.h"
#include "STM32F401RE_GPIO.h"



void ADC_Init(){
    RCC->APB2ENR |= (1 << 8); //ADC Digital Clock enable
    ADC1_CCR->ADC_CCR.ADCPRE = 0b01; // ADC analog clock = PCLK2/4;
    ADC1->ADC_CR2.ADON = 1;      // Turn ADC on

    ADC1->ADC_SQR1.L = 0b11; // Specify 4 channels in squence
    pinMode(GPIOC, 4, GPIO_ANALOG,0,0,0); // Give PC4 analog functionality,
Kettle
    pinMode(GPIOC, 1, GPIO_ANALOG,0,0,0); // Give PC1 analog functionality, v60
    pinMode(GPIOC, 2, GPIO_ANALOG,0,0,0); // Give PC1 analog functionality, Top
Load Cell
    pinMode(GPIOC, 3, GPIO_ANALOG,0,0,0); // Give PC1 analog functionality,
Bottom load cell

    // Thresholding
    ADC1->ADC_LTR = 20;
    ADC1->ADC_CR1.AWDIE = 1;
    ADC1->ADC_CR1.AWDSGL = 1;
    ADC1->ADC_CR1.AWDEN = 1;
    ADC1->ADC_CR1.JAWDEN = 0;
    ADC1->ADC_CR1.AWDCH = 14;

    //===================//
    // Kettle | PC4 | 14 //
    // Bottom | PC2 | 12 //
    //  Top   | PC3 | 13 //
    //  V60   | PC1 | 11 //
    //===================//

    ADC1->ADC_SQR3.SQ1 = 14; // ADC sequence starts with ch10, Kettle
    ADC1->ADC_SQR3.SQ2 = 12; // ch11 2nd, Bottom Load Cell
    ADC1->ADC_SQR3.SQ3 = 13; // ch12 3rd, Top Load Cell
    ADC1->ADC_SQR3.SQ4 = 11; // ch13 4th, V60 Thermistor

    // ADC1->ADC_SMPR1 |= 0b111 << 3;
    // ADC1->ADC_SMPR1 |= 0b111 << 6;
    // ADC1->ADC_SMPR1 |= 0b111 << 9;
    // ADC1->ADC_SMPR1 |= 0b111 << 12;

    ADC1->ADC_CR1.SCAN = 1; // Enable Scan mode
    ADC1->ADC_CR2.EOCS = 1; // Set EOC after individual conversions
    //ADC1->ADC_CR1.EOCIE = 1; // Enable EOC to trigger interrupt

}

uint16_t ADCRead(uint16_t sense_ID){
    uint16_t data[4];
    ADC1->ADC_CR2.CONT = 0; // Single conversion mode
```

```
    ADC1->ADC_CR2.SWSTART = 1; // Begin conversion
    while(!(ADC1->ADC_SR.EOC)); // Wait for end of convt
    data[0] = ADC1->ADC_DR.DATA; // read & return adc

    while(!(ADC1->ADC_SR.EOC)); // Wait for end of convt
    data[1] = ADC1->ADC_DR.DATA; // read & return adc

    while(!(ADC1->ADC_SR.EOC)); // Wait for end of convt
    data[2] = ADC1->ADC_DR.DATA; // read & return adc

    while(!(ADC1->ADC_SR.EOC)); // Wait for end of convt
    data[3] = ADC1->ADC_DR.DATA; // read & return adc

    return data[sense_ID];
}
```

Modified section of GPIO library

```c
// STM32F401RE_GPIO.c
// Source code for GPIO functions

#include "STM32F401RE_GPIO.h"

void pinMode(GPIO_TypeDef* GPIO_PORT_PTR, int pin, int function, int otype, int
ospeed, int pupd) {
    switch(function) {
        case GPIO_INPUT:
            GPIO_PORT_PTR->MODER &= ~(0b11 << 2*pin);
            GPIO_PORT_PTR->PURPDR |= (ospeed << pin * 2);
            GPIO_PORT_PTR->OTYPER |= (otype << pin);
            GPIO_PORT_PTR->OSPEEDR |= (pupd << pin * 2);
            break;
        case GPIO_OUTPUT:
            GPIO_PORT_PTR->MODER |= (0b1 << 2*pin);
            GPIO_PORT_PTR->MODER &= ~(0b1 << (2*pin+1));
            GPIO_PORT_PTR->PURPDR |= (ospeed << pin * 2);
            GPIO_PORT_PTR->OTYPER |= (otype << pin);
            GPIO_PORT_PTR->OSPEEDR |= (pupd << pin * 2);
            break;
        case GPIO_ALT:
            GPIO_PORT_PTR->MODER &= ~(0b1 << 2*pin);
            GPIO_PORT_PTR->MODER |= (0b1 << (2*pin+1));

            GPIO_PORT_PTR->OTYPER |= (otype << pin);
            GPIO_PORT_PTR->PURPDR |= (ospeed << pin * 2);
            GPIO_PORT_PTR->OSPEEDR |= (pupd << pin * 2);
            break;
        case GPIO_ANALOG:
            GPIO_PORT_PTR->MODER |= (0b11 << 2*pin);
            GPIO_PORT_PTR->PURPDR |= (0b00 << pin * 2);
            break;
    }
}

int digitalRead(GPIO_TypeDef* GPIO_PORT_PTR, int pin) {
    return ((GPIO_PORT_PTR->IDR) >> pin) & 1;
}

void digitalWrite(GPIO_TypeDef* GPIO_PORT_PTR, int pin, int val) {
    if(val == 1){
        GPIO_PORT_PTR->ODR |= (1 << pin);
    }
    else if(val == 0){
        GPIO_PORT_PTR->ODR &= ~(1 << pin);
    }
}

void togglePin(GPIO_TypeDef* GPIO_PORT_PTR,int pin) {
    // Use XOR to toggle
    GPIO_PORT_PTR->ODR ^= (1 << pin);
}
```