Thermistor HVAC System

Final Project by

Matthew Williams and Jenna Yoder

December 15, 2020

Table of Contents

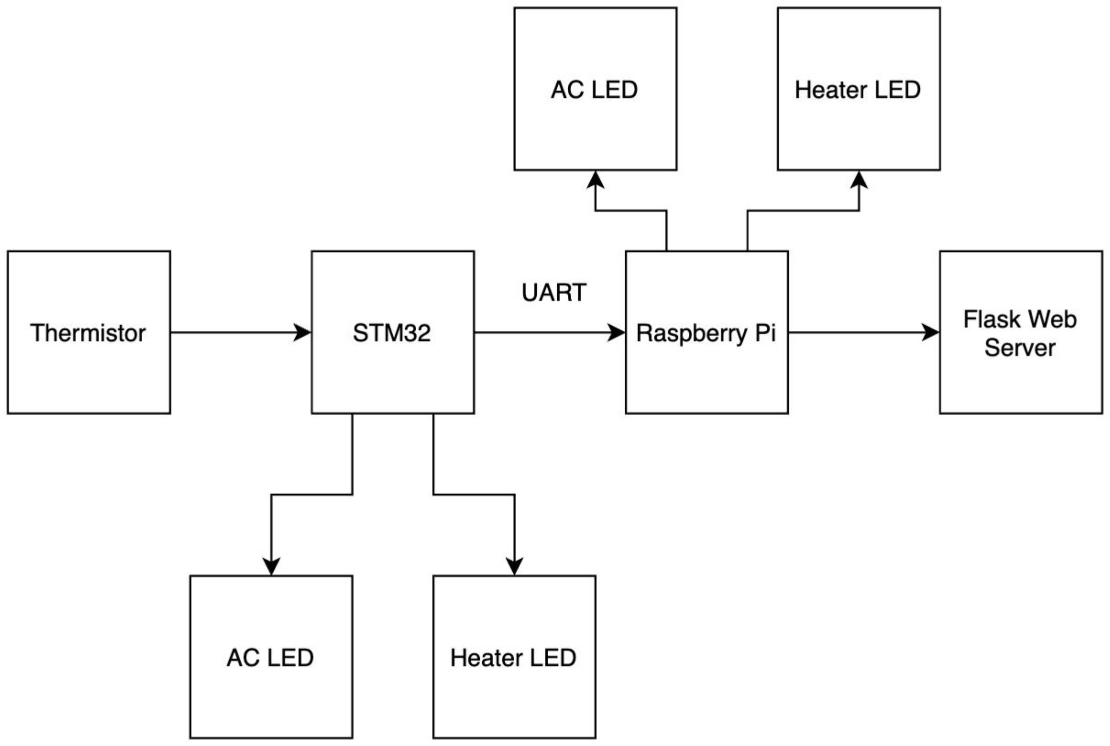
Abstract

Our project uses a STM32F303K8 Nucleo board to record temperature from a thermistor. The goal was to build a station that can detect changes in the weather from the STM32 and send that data to a raspberry pi computer. The readings are sent to a Raspberry Pi via UART protocol which then generates a plot showing the changes. The graph is displayed through a flask webpage running on the raspberry pi. Using an FSM configuration on the STM32, LEDs are used to indicate when the heater or AC is activated when it meets the conditions. We approached the problem by breaking it up into modules of software (for each device) and circuit analysis.

Project Design

The temperature monitor uses UART communication to efficiently share data between two devices. The STM32 is used to record the data from the thermistor and then convert the data to Farenheit. It also has LED outputs to indicate when the heater or ac is activated according to the temperature readings. The Raspberry Pi is used to display the readings on a user friendly interface on a Flask webpage. Figure 1 shows the block diagram setup.

Figure 1: Block Diagram

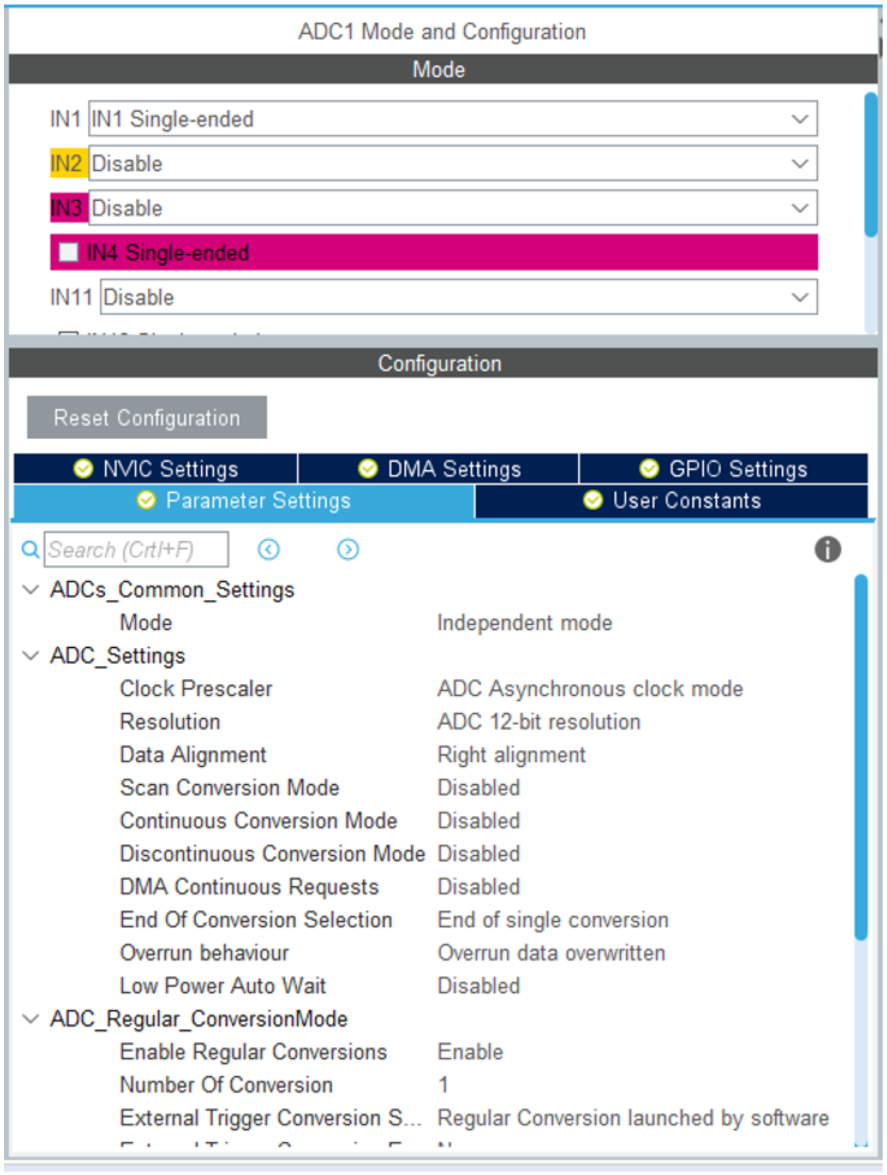


STM32 Configuration

For this lab we are going to implement an Analog to digital converter for the STM32 to read voltage. The circuitry for this is very simple as it is just a potentiometer that a GPIO pin reads the data pin, and it connects to voltage and ground. ADC is important in embedded systems as it is critical for using sensors and interpreting raw signals. The ADC on the STM32F303K8 is 12 bits meaning it has a range of 0 to 4095.

We will use STM32CubeIDE and open a project for our board. At the screen displaying the pinout we select the PA0 pin and select ADC1_IN1 and match the settings to the picture below:

Figure 2: ADC1_IN1 settings



IN1	IN1 Single-ended
IN2	Disable
IN3	Disable
	IN4 Single-ended
IN11	Disable

ADCs_Common_Settings	
Mode	Independent mode
ADC_Settings	
Clock Prescaler	ADC Asynchronous clock mode
Resolution	ADC 12-bit resolution
Data Alignment	Right alignment
Scan Conversion Mode	Disabled
Continuous Conversion Mode	Disabled
Discontinuous Conversion Mode	Disabled
DMA Continuous Requests	Disabled
End Of Conversion Selection	End of single conversion
Overrun behaviour	Overrun data overwritten
Low Power Auto Wait	Disabled
ADC_Regular_ConversionMode	
Enable Regular Conversions	Enable
Number Of Conversion	1
External Trigger Conversion S...	Regular Conversion launched by software

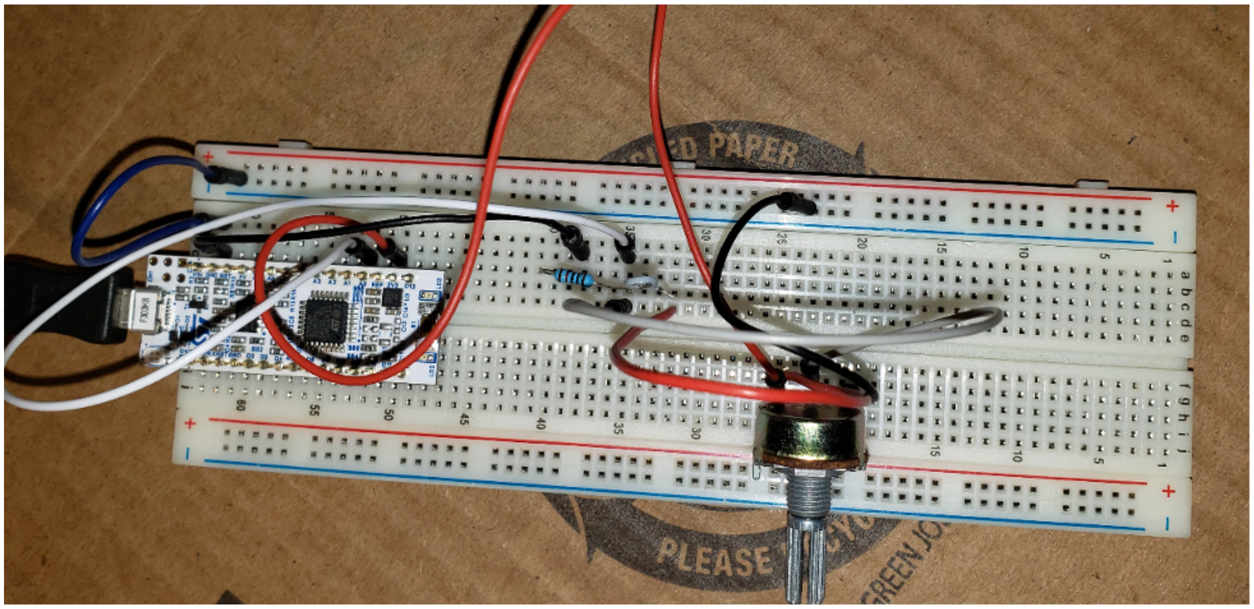
We are using single channel continuous conversion mode as the ADC needs to scan for changes in voltage. ADC can be told to run by either hardware or software signals but our method will use software. A GPIO output pin is also configured to read the voltage drop from a resistor.

The PollForConversion method is used to poll the ADC. Most of the code for the STM32 is auto generated but in order to get the ADC working properly we have to write more. Figure 3 shows the added code. Figure 4 displays the circuit design.

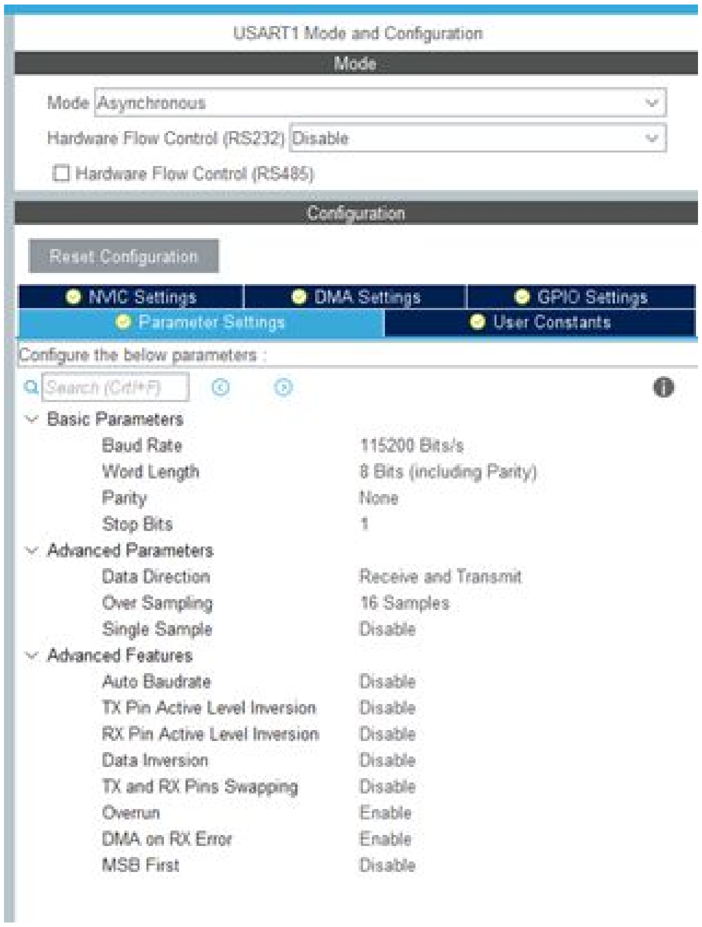
Figure 3: ADC code

```
ADC_HandleTypeDef hadc1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);
uint16_t num; //add above main
int main (void){
   HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  MX_ADC1_Init();
While(1){
            Int volt = 0;
            HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_SET);
            HAL_ADC_Start(&hadc1); // start the adc
            HAL_ADC_PollForConversion(&hadc1, 100); // poll for conversion
            num = HAL_ADC_GetValue(&hadc1); // get the adc value
            HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_SET);
            HAL_Delay (500); // wait for 500ms
}
}
```

Figure 4: Circuit design



For this section we will show how to set up UART to display our output. UART is a protocol to transmit serial data between devices and is easy to use. Because we are not using the synchronous version USART, we are able to do this with minimal tools for timing analysis. The baud rate must be the same on both devices and for this lab we will use the polling method which blocks other processes until the transmission is complete. The settings for the configuration are shown below:


USART1 Mode and Configuration

Mode

Mode: Asynchronous

Hardware Flow Control (RS232): Disable

☐ Hardware Flow Control (RS485)

Configuration

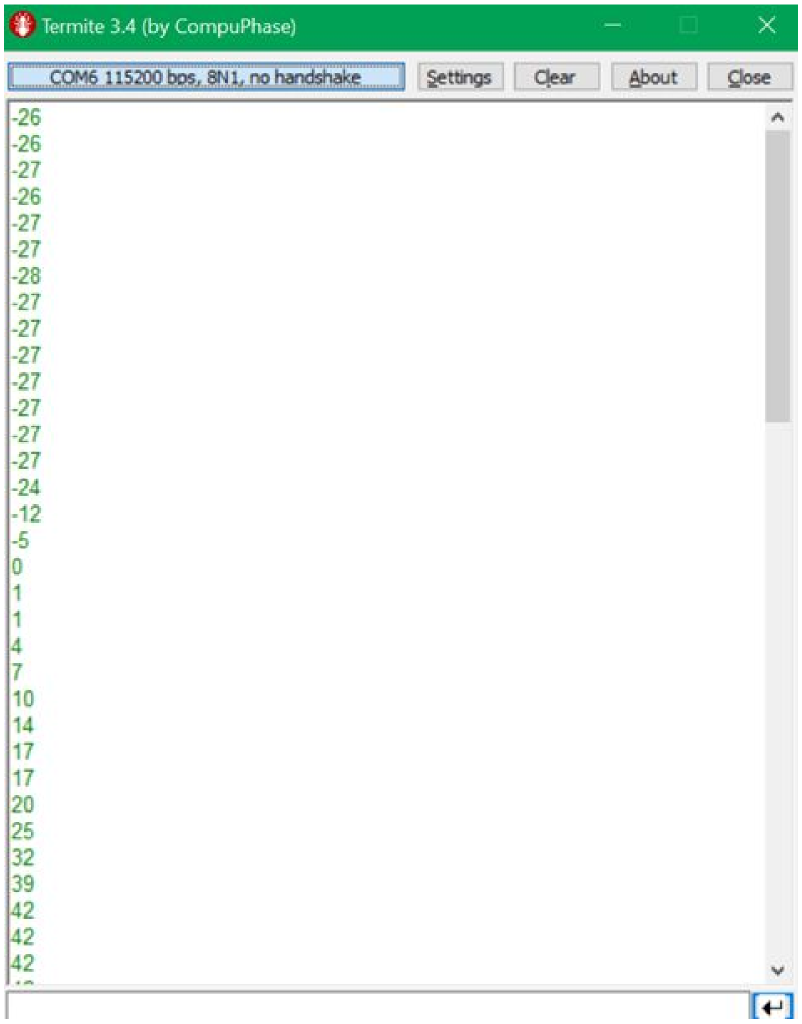
Configure the below parameters :

Basic Parameters	
Baud Rate	115200 Bits/s
Word Length	8 Bits (including Parity)
Parity	None
Stop Bits	1
Advanced Parameters	
Data Direction	Receive and Transmit
Over Sampling	16 Samples
Single Sample	Disable
Advanced Features	
Auto Baudrate	Disable
TX Pin Active Level Inversion	Disable
RX Pin Active Level Inversion	Disable
Data Inversion	Disable
TX and RX Pins Swapping	Disable
Overrun	Enable
DMA on RX Error	Enable
MSB First	Disable

Now the majority of the code is automatically generated but here is the section that needs to be added in the main function:

```
      uint16_t raw;
       char msg[10];
       while(1){

         HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_SET);

         HAL_ADC_Start (&hadc1);

         HAL_ADC_PollForConversion (&hadc1, HAL_MAX_DELAY);

         raw = HAL_ADC_GetValue(&hadc1);
         HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_SET);
        double temp = aveTemp();  //aveTemp is a function for collecting ADC, can
       //be replaced with counter to simply observe the output

         sprintf(msg,"%d\r\n", (int)temp);
         HAL_UART_Transmit(&huart2, (uint8_t*)msg , strlen(msg), HAL_MAX_DELAY);
        HAL_Delay(100);
        }
```

For the output we use a program called Termite which is easily configured to match the UART settings from earlier to get the output printed. The console in the IDE can also be used.


Termite 3.4 (by CompuPhase)
COM6 115200 bps, 8N1, no handshake
Settings
Clear
About
Close

```
-26
-26
-27
-26
-27
-27
-28
-27
-27
-27
-27
-27
-27
-27
-24
-12
-5
0
1
1
4
7
10
14
17
17
20
25
32
39
42
42
42
```

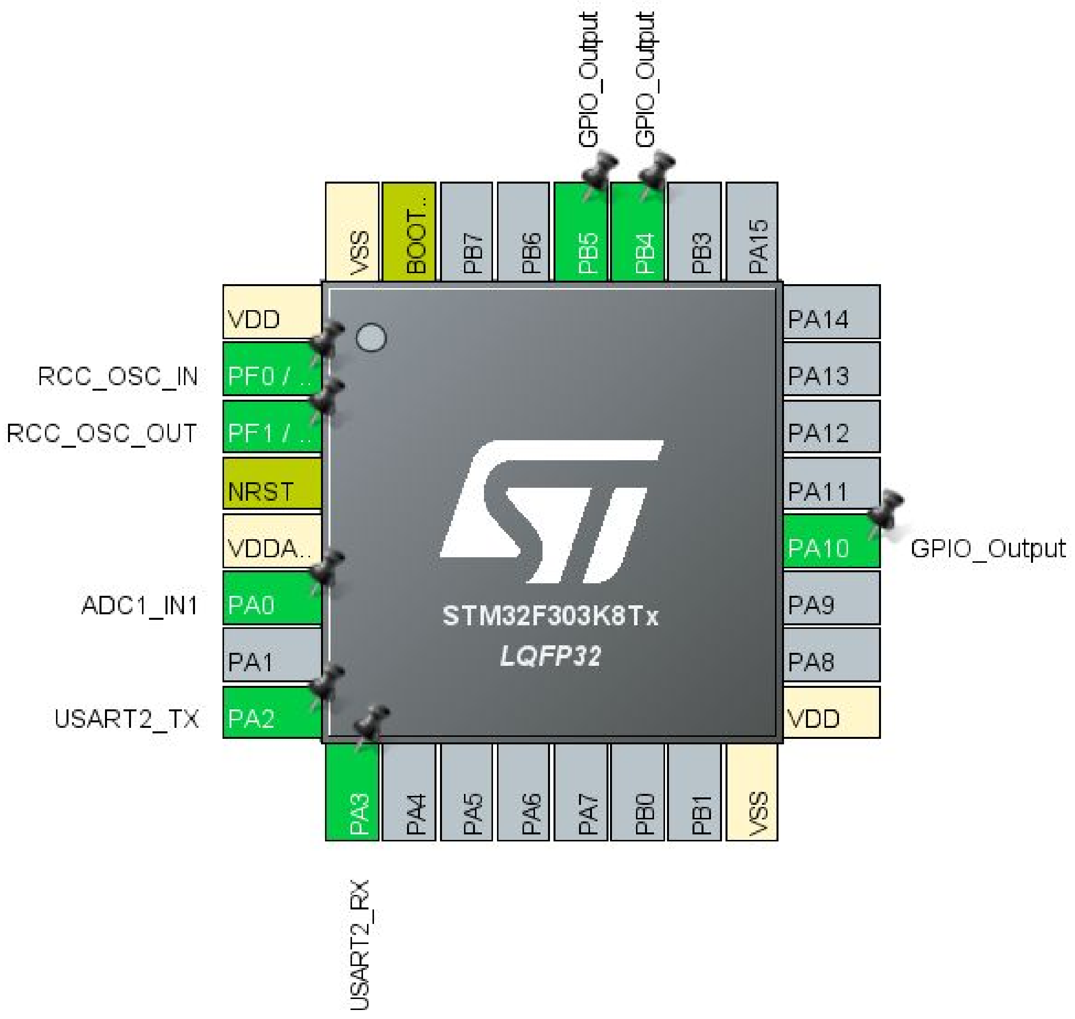
To begin assembling the STM32 software in its final formation, you want to make sure the pinout & configuration figure looks like this:

The only new addition from both the ADC and UART portions was the addition of 2 GPIO output pins to control the HVAC LEDs. Once, generate code and begin editing the main file. Before the main function, there are a few modules to add so we can call on them from main(). The first are some variables to maintain or track items of interest such as the temperature, maximum ADC value, and inverse of beta. All of this code is linked in the bitbucket link. Next is our aveTemp() function which takes 5 samples from the ADC and adds them together. This total is divided by the number of samples to get an average reading and this value is inserted in the equation that will tell us the temperature in Kelvin. We then convert to Celsius and return that number. After that is a simple test called ACtest() which checks if the temperature is right for one of the LEDs to turn on.

In the main function, we repeat the process for starting and managing the ADC and transmitting data via UART although now we make function calls to aveTemp() and ACtest() and pass the temperature to the UART function.

Circuit/Hardware:

Figure 5: STM32 pinout



Initially our project was to include a DHT11 temperature and humidity sensor that would connect (either directly or via I2C) a 16x2 LCD that the raspberry pi would use to display the live temperature. While significant progress was made in building it, ultimately the design had to start over as the timing analysis for the DHT11 was failing. It communicated via UART but needed precise timing of pulling up and down an internal resistor to indicate its use and the program would fail or get stuck in that process. So we switched to a thermistor which requires less complex code but more complex circuit analysis.

The thermistor[1] is 10k Ohms and initially was in series with a 10k Ohm resistor so that a simple voltage division would work to calculate the voltage across the thermistor. However, this configuration did not give us the ability to vary the voltage enough to see changes, collect data, and confirm the functionality. So the next design choice was made and we used a 10k Ohm potentiometer in series with the thermistor. (This was at the cost of knowing the exact resistance of the potentiometer at any given moment). To use the most accurate equation for the temperature of a thermistor, the Steinhart-Hart equation, you need 3 variables from the datasheet however one of them was not provided for ours. Instead, it is common for them to provide a data table organized by beta, resistance, and temperature. So we used a common derivation of Steinhart-Hart that works with our datasheet :

$$1/T = 1/T0 + 1/B * \ln(R/R0)$$

"The variable T is the ambient temperature in Kelvin, T0 is usually room temperature, also in Kelvin (25°C = 298.15K), B is the beta constant, R is the thermistor resistance at the ambient temperature (same as Rt above), and R0 is the thermistor resistance at temperature T0. The values for T0, B, and R0 can be found in the manufacturer's datasheet."[2]

To modify the above formula in code we needed R/R0 to be the ratio of the ADC measurement over its maximum value:

$$\text{Temperature in Kelvin} = 1/(\ 1/298.15\ +\ 1/4300\ *\ log(4095/ADC_{value}\ - 1)$$

To convert to Celcius, subtract 273.15 from Kelvin.

We can calculate the resistance at any given temperature from this equation:

$$R = R\_T0 * e\text{^}(B*(1/T - 1/T\_0))$$

R is the resistance we want, R_T0 is the standard resistance at temperature T_0 which is 10k Ohms for 25 degrees C, and B is the beta value given which was 4300.

Because the circuit elements are in series we can use a voltage divider formula to get the voltage drop across the thermistor:

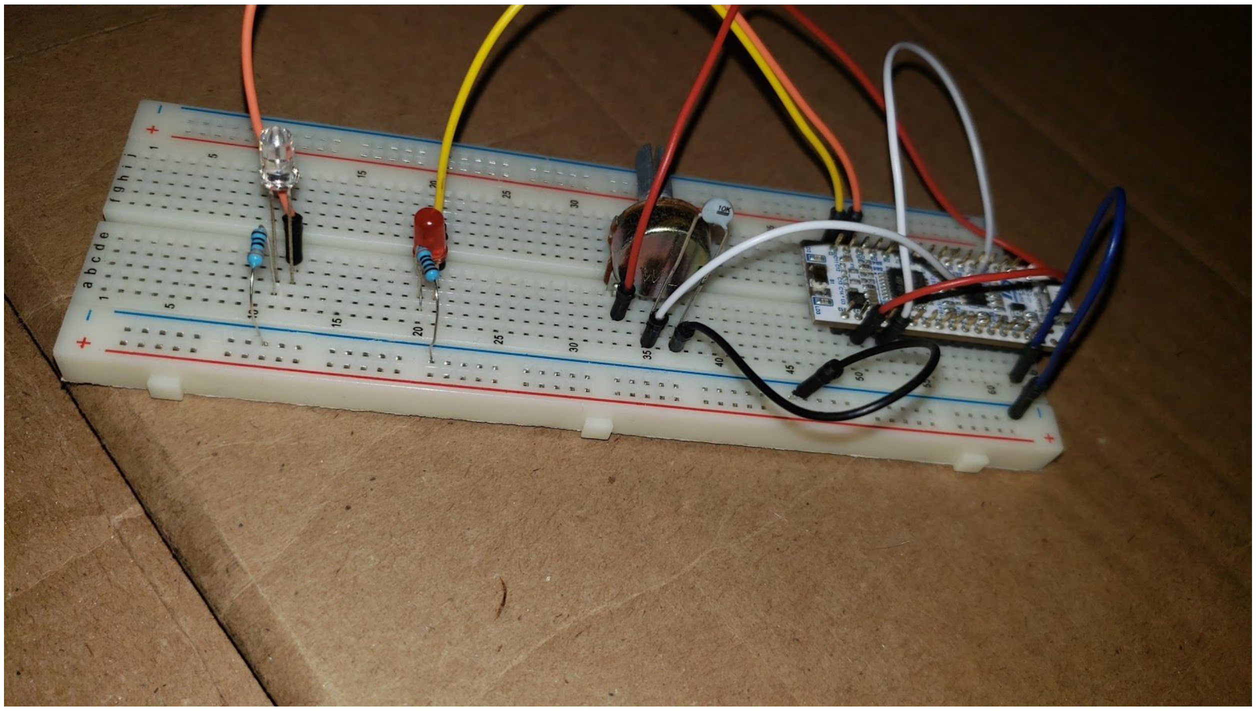
$$Vout = Vin * (R\_T/(R\_T + Ro))$$

Where R_T is the thermistor resistance and Ro is the potentiometer resistance.

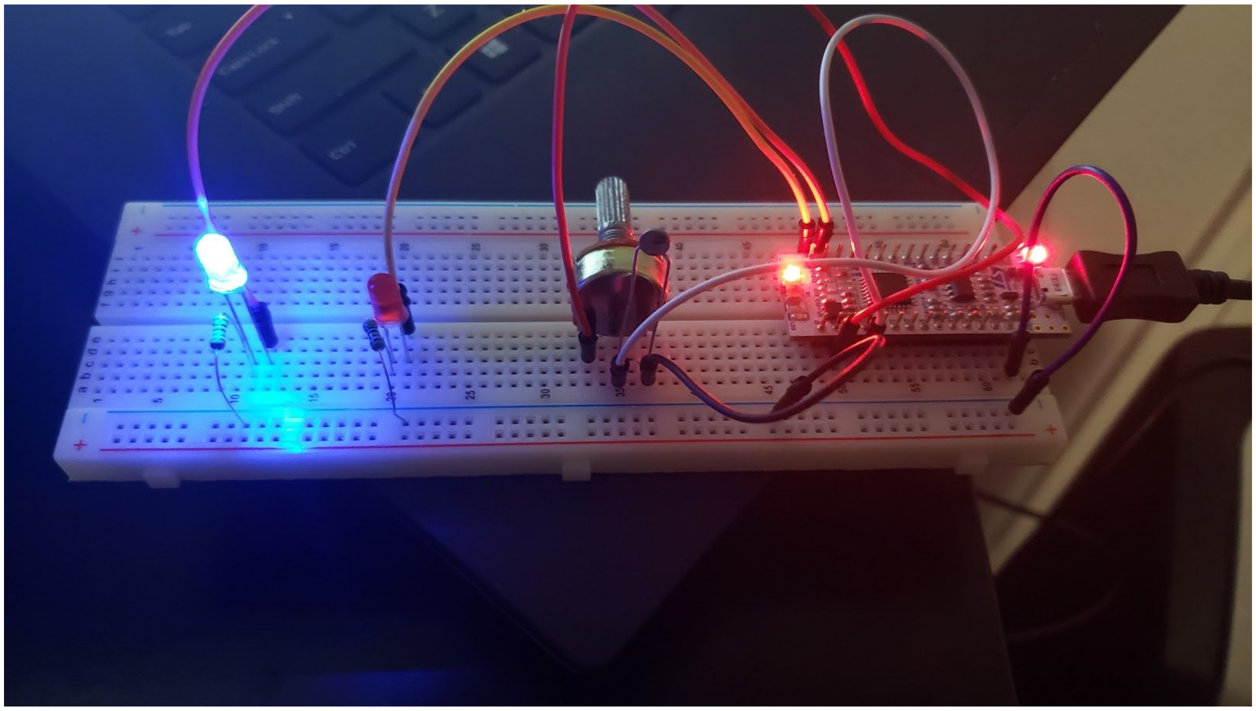
We added 2 LEDs one red and one blue to indicate when the temperature was under or over a value to simulate an HVAC system: red for heater if under 30C and blue for air conditioning if over 85C.

[1] Has a B value of 4300, Listed i docs https://product.tdk.com/info/en/documents/data_sheet/50/db/ntc/NTC_Leaded_disks_K164.pdf

[2] https://www.jameco.com/Jameco/workshop/TechTip/temperature-measurement-ntc-thermistors.html



Circuit working with AC LED on:



Test cases:

The original test cases proposed in the Final Project Description are as follows:

1. Testing limits of the sensor and what to do if it exceeds these limits:
As we expected the thermistor is not as accurate as the DHT11 sensor and this became more obvious around the upper and lower limits of it's temperature range. Initially for the C program we wanted to test if the edge case integers would be a problem such as negative numbers, zero, and triple digits. Although there were issues initially with the negative sign being interpreted as a string on python we eventually got that all working. The discrepancies were when the potentiometer was at its lowest point of zero ohms in which the output number would always be zero even if an extremely low ADC value would result in an expected negative temperature value so this was fixed with a simple if statement to force a zero ADC value to be -55 degrees C. However, it was accurate in reading the voltage and change in resistance of the thermistor and thus using the equations described in the circuit and hardware section we can verify using a multimeter across the thermistor that the voltage and resistance readings were reasonably accurate.

2.Ensure heater LED turns on and off for the given threshold.
Heater LED did consistently turn on after the temperature was under 30C.

3.Ensure AC LED turns on and off for the given threshold.
AC LED did consistently turn on after the temperature was over 85C.

4.Test that flask webpage properly displays sensor readings.
The flask webpage did also display the temperature although it was via a static image of a graph.
5.Test that STM32 is properly communicating with Raspberry Pi.
We confirmed the UART communication was working and could read the data to a csv file.

Known issues:

There is a bug when plotting the database where to read the csv file and place the data into a list, the program may fail due to finding an unusual character typically “”. This character is generated from the write to database program specifically when quitting with Ctrl C this sometimes causes an error, even though it is in try-catch block with the exception handled it still enters a character into the file. This could be solved with another python script to find the character and remove it however initially the data is in string format and when converting to int and finding the odd character the program would fill to interpret a negative sign making the data unusable. The more practical solution is to scroll to the bottom of the csv file after writing to it to check for “” and delete it.

When the potentiometer is at 0 the thermistor and ADC measurement also gets 0 so the test case for the lowest values are less reliable. To work around this we added an if statement to output the lowest value from the datasheet if the ADC input is 0.

We wanted to make a live graph that displays current temperatures on the web page but that needed a version of SQLite that was not working on the pi with the plot packages as there was a dependency error that made it too difficult to use.

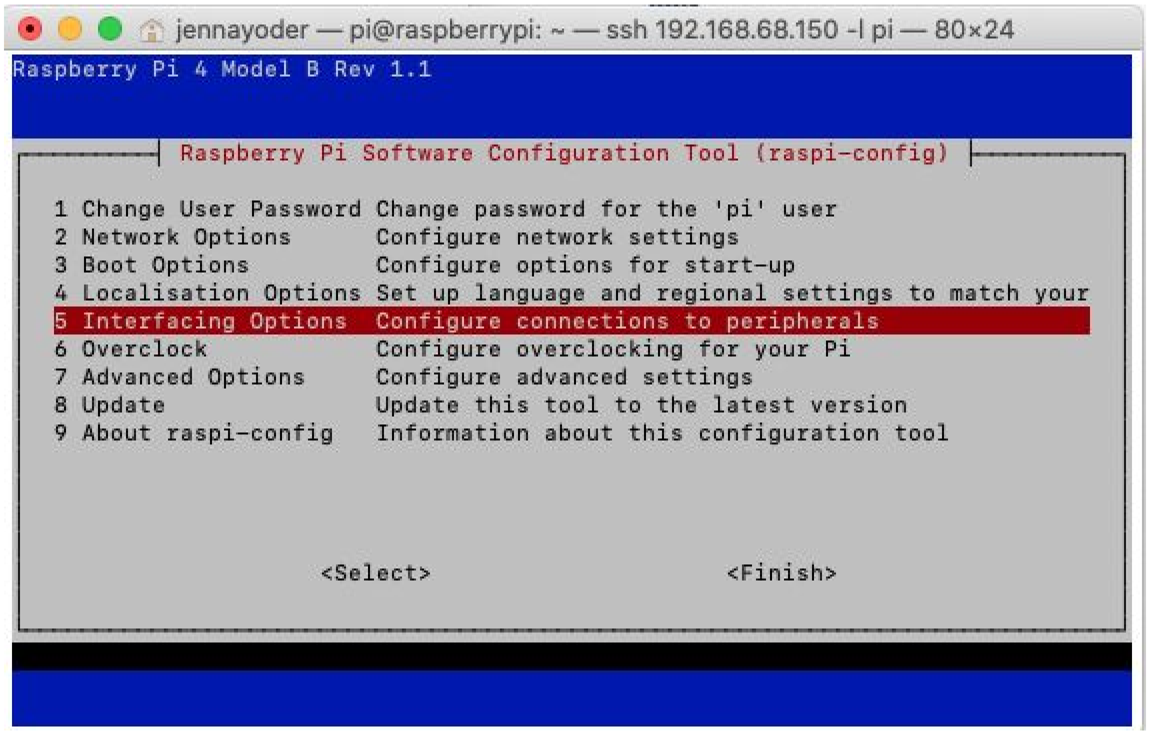
The flask page when launched, if not using the most recent png image the user must press shift+ctrl+R to refresh the page.

Raspberry Pi Configuration

In order to set up a serial communication to other devices, the serial port communications have to be set up on the Raspberry Pi. This project uses Serial0 on the Pi.

The command $ sudo raspi-config takes us to the menu shown in figure 7. We select "interfacing options" to enable the serial port connection.

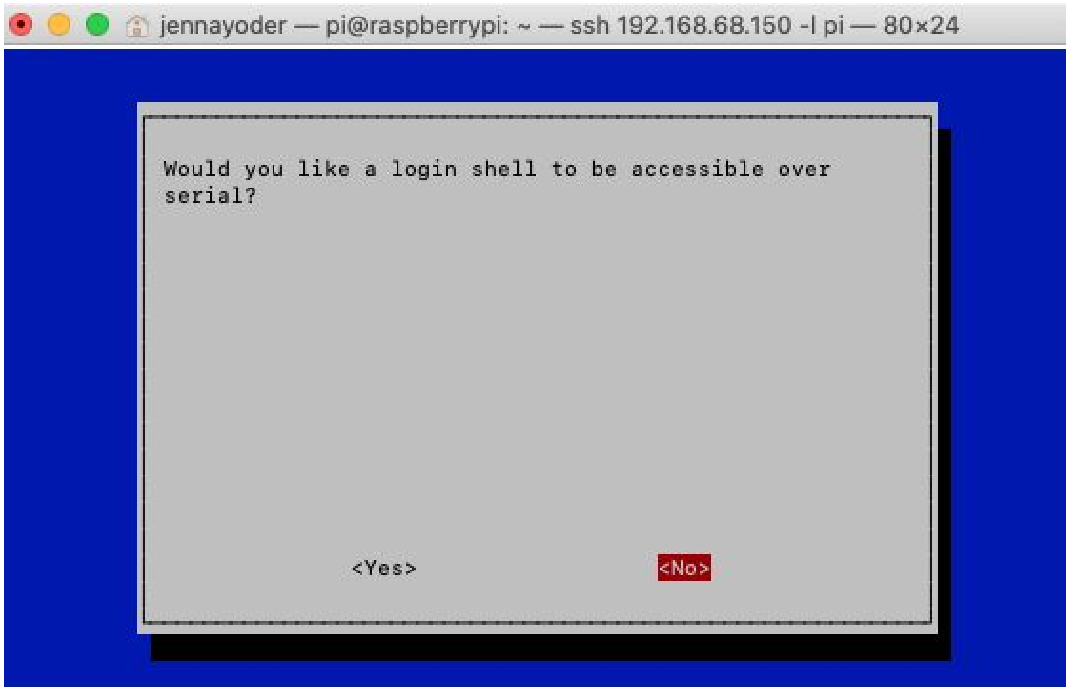
Figure 7: Raspberry Pi configure menu



```
jennayoder — pi@raspberrypi: ~ — ssh 192.168.68.150 -l pi — 80×24
Raspberry Pi 4 Model B Rev 1.1

          Raspberry Pi Software Configuration Tool (raspi-config)

 1 Change User Password Change password for the 'pi' user
 2 Network Options      Configure network settings
 3 Boot Options         Configure options for start-up
 4 Localisation Options Set up language and regional settings to match your
 5 Interfacing Options  Configure connections to peripherals
 6 Overclock            Configure overclocking for your Pi
 7 Advanced Options     Configure advanced settings
 8 Update               Update this tool to the latest version
 9 About raspi-config   Information about this configuration tool

                  <Select>                      <Finish>
```

We select no to disable the login shell from being accessible over the serial connection.

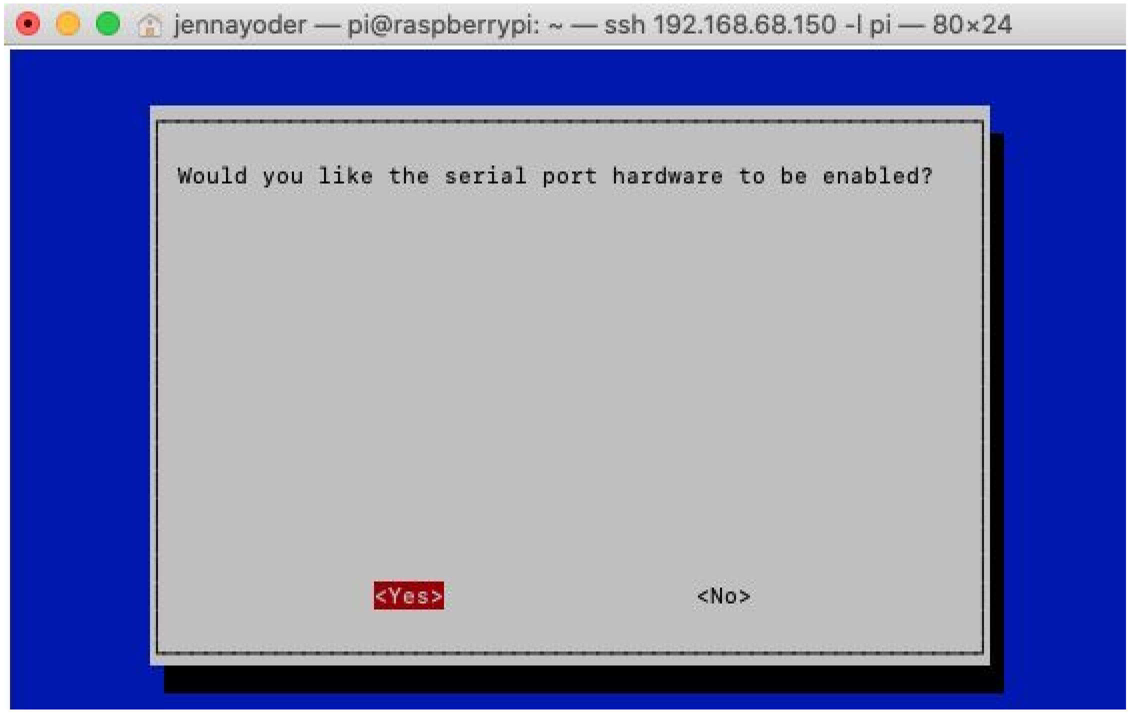
Figure 8: Disable login shell



```
jennayoder — pi@raspberrypi: ~ — ssh 192.168.68.150 -l pi — 80×24

Would you like a login shell to be accessible over
serial?

          <Yes>                    <No>
```

We then enable the serial port hardware.

Figure 9: Enable serial port hardware



```
jennayoder — pi@raspberrypi: ~ — ssh 192.168.68.150 -l pi — 80×24

Would you like the serial port hardware to be enabled?

          <Yes>                    <No>
```

Once the Pi was set up for serial connection, we wrote a python script to read and record the data coming from the STM32 to a database. The data is stored into a .csv file on the pi name "database3.csv". The code in figure 10 is named WritetoDatabase3.py in the source files on bitbucket.

Figure 10: WritetoDatabase3.py

```
#!/usr/bin/env python3
import serial
import time
import csv

#detects the device and its baud rate
if __name__ =='__main__':
    ser = serial.Serial('/dev/ttyACM0', 115200,timeout=1)
    ser.flush()

    while True:     #infinite loop to append data to file as strings
        if ser.inWaiting() >0:   #while serial connection is detected
            line = ser.readline().decode('utf-8').rstrip()
            print(line)
            with open("database3.csv","a") as f:
                writer = csv.writer(f,delimiter=",")
                writer.writerow([line])
```

When the data was stored to the .csv file, we then plotted it to a graph and saved it as a .png file. The libraries matplotlib and numpy were used to make creating the figure easier. The code in figure 11 is named plotDatabase3.py in the source code on bitbucket.

Figure 11: plotDatabase3.py

```
import matplotlib.pyplot as plt
import csv
import numpy as np

x=[]
y=[]
#loop will read the data and convert it to an integer array
```

```
with open('database3.csv','r') as csvfile:
    points = open('database3.csv').read().splitlines()

    i = 0
    for row in points:
        nums = [int(k) for k in row.split()]
        y.append(nums)
        x.append(i)
        i = i + 1
flaty = []  #converts list of lists to single list
for elem in y:
    flaty.extend(elem)

print(flaty)

plt.plot(x, flaty)
plt.grid(True)

plt.ylim([-55,155])  #sets limits of y range
plt.title("Thermistor Reading")
plt.xlabel("Time")
plt.ylabel("Temperature")



plt.savefig('static/graph3.png')
plt.savefig('graph3.png')  #save in multiple places for safety
plt.savefig('templates/static/graph3.png')
plt.savefig('templates/graph3.png')
plt.savefig('templates/img/graph3.png')

plt.show()
```

We then created the code for the Flask webpage which displays the graph figure we created. In order to display the figure, we had to create the flask page and then create an HTML file to style the page. Figure 12 displays the python script for the page and figure 13 displays the HTML code.

Figure12: app.py

```
from flask import Flask, render_template, url_for
#import serial
import time
import datetime



#Get a thermistor reading to display on webpage
#ser = serial.Serial('/dev/ttyACM0', 115200, timeout = 1 )
#ser.flush()
#temp = ser.readline().decode('utf-8') #assigns the serial reading into the variable "temp"



app = Flask(__name__)
@app.route('/')  #navigates to main directory
def index():
    now = datetime.datetime.now()
    timeString = now.strftime("%m-%d-%Y  %H:%M")

    templateData = {
        'time' : timeString,
        #'sensor' : temp
    }
    return render_template('data2.html', **templateData)

if __name__ == '__main__':
    app.run(debug=True, port=5000, host='0.0.0.0')
```

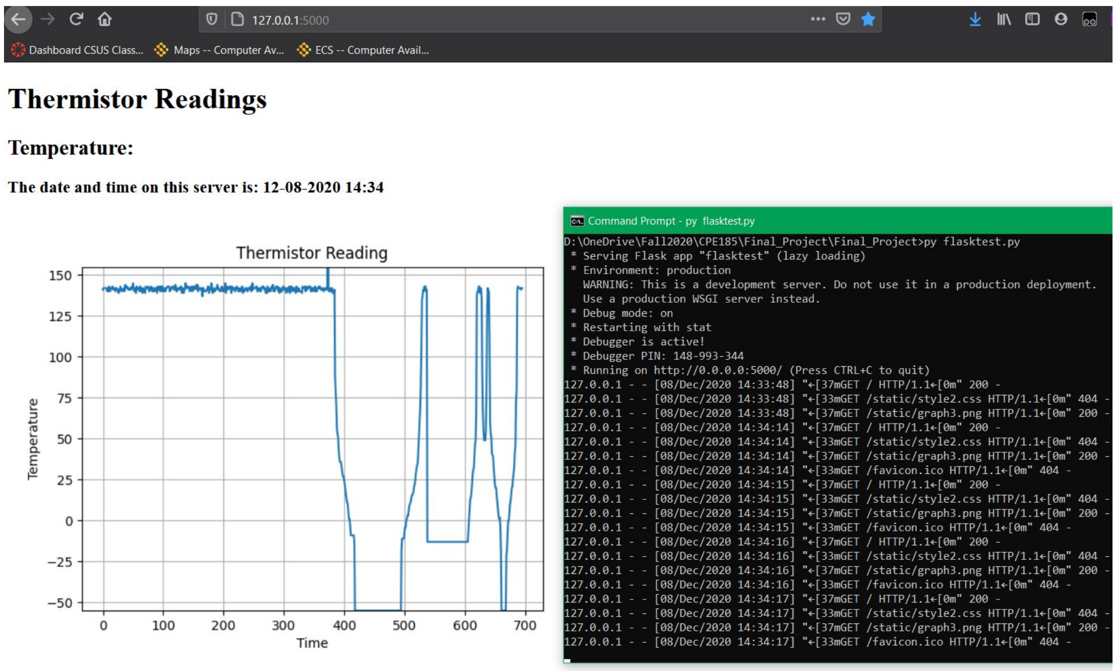
The function return render_template() allows the python script to pull the template data from the HTML script in figure 13.

Figure 13: data.html

```
<!DOCTYPE html>
    <head>
    <meta charset="UTF-8" >
        <title>{{ title }}</title>
        <link rel="stylesheet" href='/static/style2.css' />
    </head>
    <body>
        <h1>Thermistor Readings</h1>
        <h2>Temperature: {{ sensor }}<h2>
        <h3>The date and time on this server is: {{time}}</h3>

        <img src="/static/graph3.png" alt="Image Placeholder" height = "500">
        <img src="{{url_for('static',filename='graph3.png')}} >
    </body>
</html>
```

This project requires that each script is run sequentially. First we read the data from the STM32 and store it to a .csv file. Then the stored data is plotted to a graph which is saved as a .png file. Then script for the webpage is run which displays the image. Figure 14 displays the flask page.

Figure 14: Flask Page



Thermistor Readings

Temperature:

The date and time on this server is: 12-08-2020 14:34

```
D:\OneDrive\Fall2020\CPE185\Final_Project\Final_Project>py flasktest.py
 * Serving Flask app "flasktest" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 148-993-344
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [08/Dec/2020 14:33:48] "←[37mGET / HTTP/1.1←[0m" 200 -
127.0.0.1 - - [08/Dec/2020 14:33:48] "←[33mGET /static/style2.css HTTP/1.1←[0m" 404 -
127.0.0.1 - - [08/Dec/2020 14:33:48] "←[37mGET /static/graph3.png HTTP/1.1←[0m" 200 -
127.0.0.1 - - [08/Dec/2020 14:34:14] "←[37mGET / HTTP/1.1←[0m" 200 -
127.0.0.1 - - [08/Dec/2020 14:34:14] "←[33mGET /static/style2.css HTTP/1.1←[0m" 404 -
127.0.0.1 - - [08/Dec/2020 14:34:14] "←[37mGET /static/graph3.png HTTP/1.1←[0m" 200 -
127.0.0.1 - - [08/Dec/2020 14:34:14] "←[33mGET /favicon.ico HTTP/1.1←[0m" 404 -
127.0.0.1 - - [08/Dec/2020 14:34:15] "←[37mGET / HTTP/1.1←[0m" 200 -
127.0.0.1 - - [08/Dec/2020 14:34:15] "←[33mGET /static/style2.css HTTP/1.1←[0m" 404 -
127.0.0.1 - - [08/Dec/2020 14:34:15] "←[37mGET /static/graph3.png HTTP/1.1←[0m" 200 -
127.0.0.1 - - [08/Dec/2020 14:34:15] "←[33mGET /favicon.ico HTTP/1.1←[0m" 404 -
127.0.0.1 - - [08/Dec/2020 14:34:16] "←[37mGET / HTTP/1.1←[0m" 200 -
127.0.0.1 - - [08/Dec/2020 14:34:16] "←[33mGET /static/style2.css HTTP/1.1←[0m" 404 -
127.0.0.1 - - [08/Dec/2020 14:34:16] "←[37mGET /static/graph3.png HTTP/1.1←[0m" 200 -
127.0.0.1 - - [08/Dec/2020 14:34:16] "←[33mGET /favicon.ico HTTP/1.1←[0m" 404 -
127.0.0.1 - - [08/Dec/2020 14:34:17] "←[37mGET / HTTP/1.1←[0m" 200 -
127.0.0.1 - - [08/Dec/2020 14:34:17] "←[33mGET /static/style2.css HTTP/1.1←[0m" 404 -
127.0.0.1 - - [08/Dec/2020 14:34:17] "←[37mGET /static/graph3.png HTTP/1.1←[0m" 200 -
127.0.0.1 - - [08/Dec/2020 14:34:17] "←[33mGET /favicon.ico HTTP/1.1←[0m" 404 -
```

The next process was to use an FSM to activate the LEDs representing the heater and the AC. The FSM starts with both indicator LEDs off. The formula used to convert voltage readings to temperatures values converted them to Celsius so the values displayed in the FSM diagram are in Celsius. When the temperature reaches 20 degrees or lower, the heater LED indicator is activated. When it reaches 20 degrees the heater LED turns off. When temperatures reach 30 degrees or higher, the AC indicator LED is activated. It turns off when it reaches 25 degrees. When both LEDs are off, it is in an idle state.

Figure 15: FSM Diagram

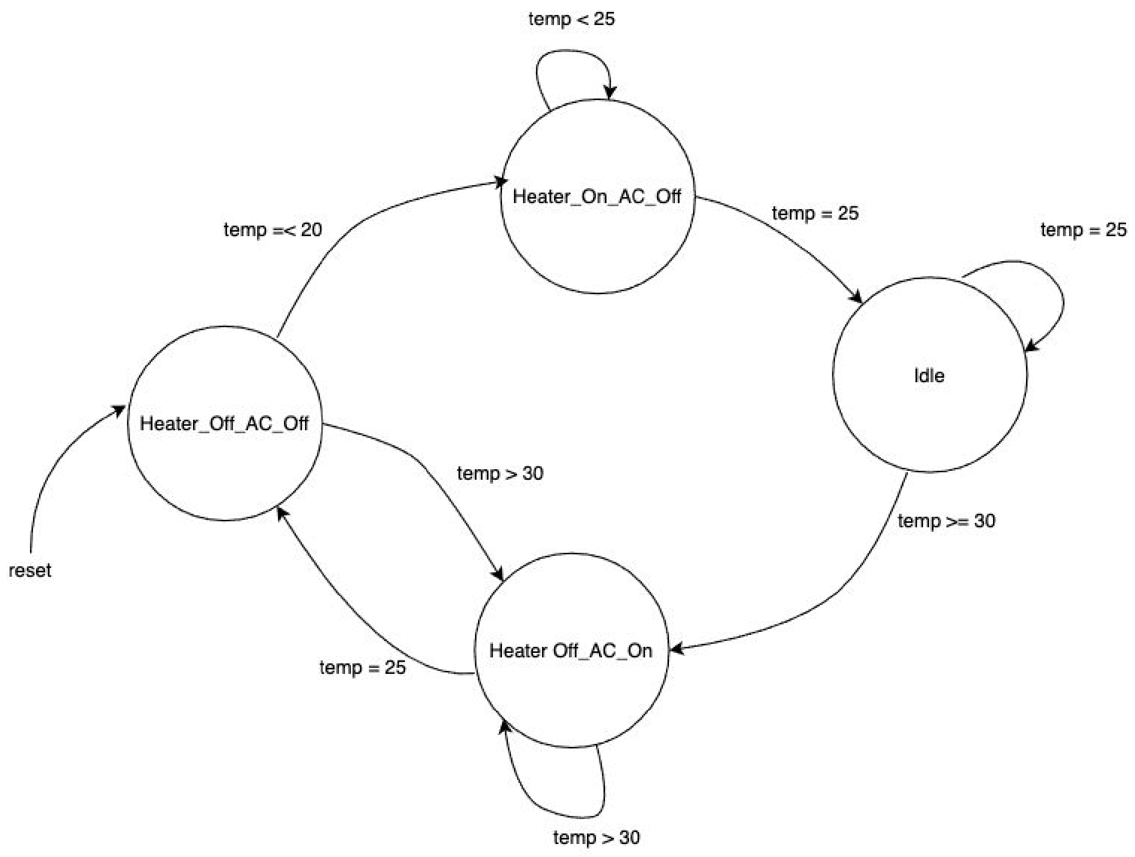


Figure 16: FSM Table

current state	Input	Next State	Output
Heater_Off_AC_Off	temp = 25	Heater_Off_AC_Off	Heater_LED = 0 AC_LED = 0
Heater_Off_AC_Off	temp <= 20	Heater_On_AC_Off	Heater_LED = 1 AC_LED = 0
Heater_On_AC_Off	temp <= 20	Heater_On_AC_Off	Heater_LED = 1 AC_LED = 0
Heater_On_AC_Off	temp = 25	Idle	Heater_LED = 0 AC_LED = 0
Idle	temp = 25	Idle	Heater_LED = 0 AC_LED = 0
Idle	temp >= 30	Heater_Off_AC_On	Heater_LED = 0 AC_LED = 1
Heater_Off_AC_On	temp >= 30	Heater_Off_AC_On	Heater_LED = 0 AC_LED = 1
Heater_Off_AC_On	temp = 25	Heater_Off_AC_Off	Heater_LED = 0 AC_LED = 0

Following the FSM, we created a python script to configure the LEDs on the Pi. Using a switch case would be the easiest for FSMs but python doesn't have case functions. Instead we decided to use Python's dictionary function.

Unfortunately we were unable to implement this function into the final project because the test's were unsuccessful. Figure 17 displays the python script used for the FSM.

Figure 17: fsmTest.py

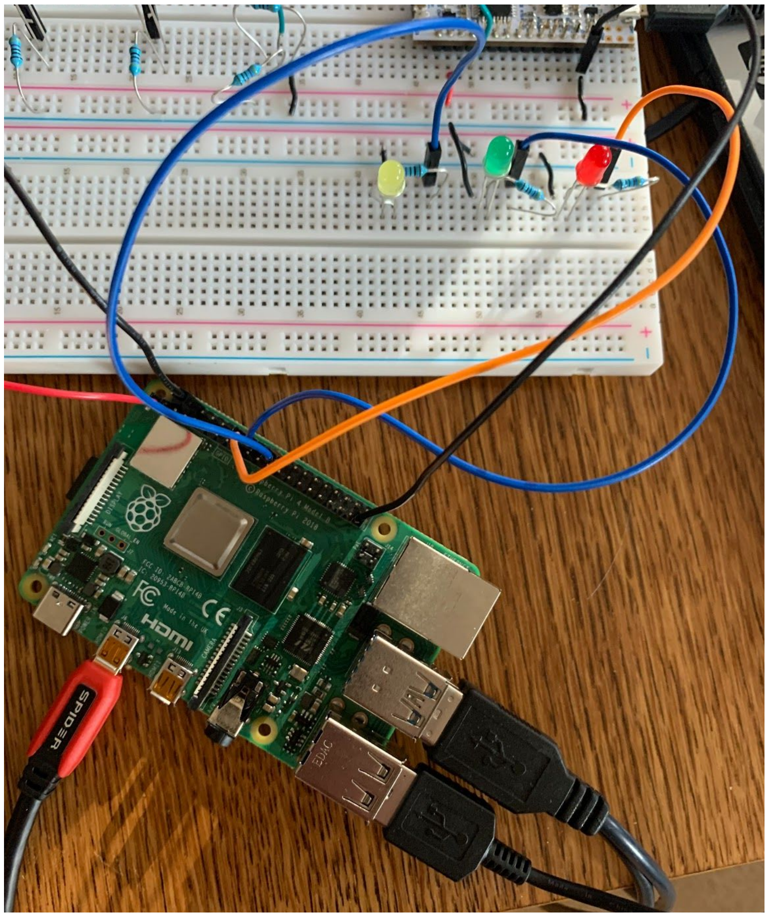
```python
import RPi.GPIO as GPIO

import time

from time import sleep

import serial


GPIO.setmode(GPIO.BCM)

GPIO.setwarnings(False)


ser = serial.Serial ('/dev/ttyS0', 115200, timeout=1)

temp = ser.readline().decode('utf-8') #decodes the serial data to readable values and puts it into a variable
```

```
named temp

# assign GPIO outputs
heater = 23
ac = 24
indicator = 25 # LED that detects data being read
dummy = "30" # The test variable to test different temp values. When testing is confirmed this variable will be
removed and replaced with temp as defined above

GPIO.setup(heater, GPIO.OUT)
GPIO.setup(ac, GPIO.OUT)
GPIO.setup(indicator, GPIO.OUT)

# defines the function and determines with GPIOs to activate
def heater_on():
    GPIO.output(heater, GPIO.HIGH)
    GPIO.output(ac, GPIO.LOW)

def nothing():
    return "nothing"



# Defines the switch cases. Case 4 is a test case to determine if while loop is reading the "nothing" function
defined above
def switch(ledStatus):
    switcher = {
        1: { GPIO.output(heater, GPIO.HIGH), GPIO.output(ac, GPIO.LOW)}, # heater on
        2: {GPIO.output(heater, GPIO.LOW), GPIO.output(ac, GPIO.LOW)}, # idle
        3: {GPIO.output(heater, GPIO.LOW), GPIO.output(ac, GPIO.HIGH)}, # ac on
        4: nothing #testing to see if it prints def nothing() function
    }
    #return switcher.get(ledStatus, "invalid")



if __name__=="__main__":
```

```
#while True:
#      GPIO.output(indicator, GPIO.HIGH)

    while dummy <= "20":
        ledStatus(1)
        print('heater on')

    while dummy == "25":
        ledStatus = 2
        print('idle')

    while dummy >= "30":
        ledStatus = 3
        print('ac on')

    else:
        ledStatus = 2
```

In figure 17 there is a variable defined "indicator" which is designed to turn on when the Raspberry Pi is reading data from the serial port. This portion was a success but since the rest of the testing did not work it is commented out in the code. Figure 18 shows the circuit design on the Raspberry Pi.

Figure 18: Raspberry Pi circuit design



Testing the FSM:

The python script used to implement the FSM is shown in figure 17. During testing the GPIO pins were unresponsive. The correct outputs were displayed on the terminal but the LEDs were not turning on. If we had more time to work on the issue we believe we would have been able to complete the testing and implement this function into our final project. Since both of us were not as experienced with python, we were unable to find a solution in time.

Conclusion

We were able to successfully collect accurate temperature readings, communicate between the STM32 and Raspberry Pi, and display the outputs to the Flask webpage. Even though we were unable to integrate the working FSM for the LEDs connected to the Pi, overall we were able to complete the main functions of the thermistor HVAC system.

Throughout the course of our project we had to modify several of our original plans. We wanted to display live temperature readings but were unable to fix the problems with the SQLite libraries. We also wanted to use a DHT11 sensor instead of a thermistor but also had problems with the timing analysis for the DHT11. Even with these modifications to our original plans, we believe we built the most efficient system given the amount of time we had and the learning curve we had to overcome in working with new software and hardware.

References

1. https://www.jameco.com/Jameco/workshop/TechTip/temperature-measurement-ntc-thermistors.html

2. https://www.allaboutcircuits.com/technical-articles/introduction-temperature-sensors-thermistors-thermocouples-thermometer-ic/

3. https://product.tdk.com/info/en/documents/data_sheet/50/db/ntc/NTC_Leaded_disks_K164.pdf

4. https://www.giangrandi.ch/electronics/ntc/ntc.shtml

5. https://matplotlib.org/gallery/index.html

6. https://www.geeksforgeeks.org/python-introduction-to-web-development-using-flask/