Final Report

Roastology

Fall 2024

Electrical and Computer Engineering

Auburn University

Members:

Alex Brown

Rick Cazenave

Abdullah Alsheri

Minmin Feng

Zayvier Hambrick

Yifan Zhu

Submitted to Dr. Geiger by December 9th, 2024

Table of Contents

Executive Summary - Rick Cazenave

The assistance of ChatGPT.com was used in the writing of this section

The Roastology project aims to create an innovative smart coffee bean roaster that combines remote monitoring and automated precision to ensure consistent, high-quality roasts. This comprehensive system integrates advanced temperature control, user-friendly interfaces, and robust hardware. The team—comprising Rick Cazenave, Alex Brown, Zayvier Hambrick, Yifan Zhu, Abdullah Alshehri, and Minmin Feng—leverages their expertise in electrical and computer engineering to refine and enhance the design.

Key components include a PT100 temperature sensor [2] paired with a MAX31865 amplifier [4] for accurate temperature monitoring, controlled via an Arduino UNO [1] and displayed in real time on an LCD1602 screen [3]. A 1500W heating element [5], modified for automated power modulation through a PID-controlled solid-state relay (SSR) [6], provides precise heat regulation. A Greartisan 12V DC motor that rotates a tin can and agitates the beans to ensure even roasting, and a stainless-steel mesh cooling tray rapidly cools the beans post-roast using Arduino-controlled [1] DC fans. These features collectively ensure optimal roast uniformity and flavor enhancement.

The project has achieved significant milestones, including successful integration of temperature monitoring, motorized rotation, and heating element regulation. Testing of roast profiles (light, medium, and dark) demonstrates promising results, though further adjustments are needed to optimize heating efficiency and improve consistency in larger batches. Remote monitoring via an ESP8266 web server enables users to control and monitor the roast process through an intuitive interface.

Future enhancements include increasing roast batch sizes, improving chaff management, refining the can design for better heat retention, and supporting additional roast profiles. The Roastology project showcases an innovative fusion of engineering principles and practical design, offering a scalable, cost-effective solution for coffee enthusiasts seeking precision roasting capabilities.

Introduction - Rick Cazenave

The assistance of ChatGPT.com was used in the writing of this section

The Roastology project is a comprehensive coffee roasting system designed in Auburn University's ECE program. Its primary aim is to automate the coffee roasting process by integrating advanced temperature monitoring, motorized bean rotation via a DC motor, and user-friendly control capabilities. The system utilizes a PT100 temperature sensor [2] for precise heat measurement, a DC motor to ensure even bean rotation within a tin can, a fan-powered cooling system, and a microcontroller to manage these components. With features like real-time temperature feedback and remote functionality, Roastology seeks to deliver a consistent, high-quality coffee roast. This approach addresses the need for precise control in roasting, and combining engineering and technology to enhance small-scale coffee bean roasting. As a reference for each block of circuit used throughout this report, Fig.1 shows the full circuit diagram.

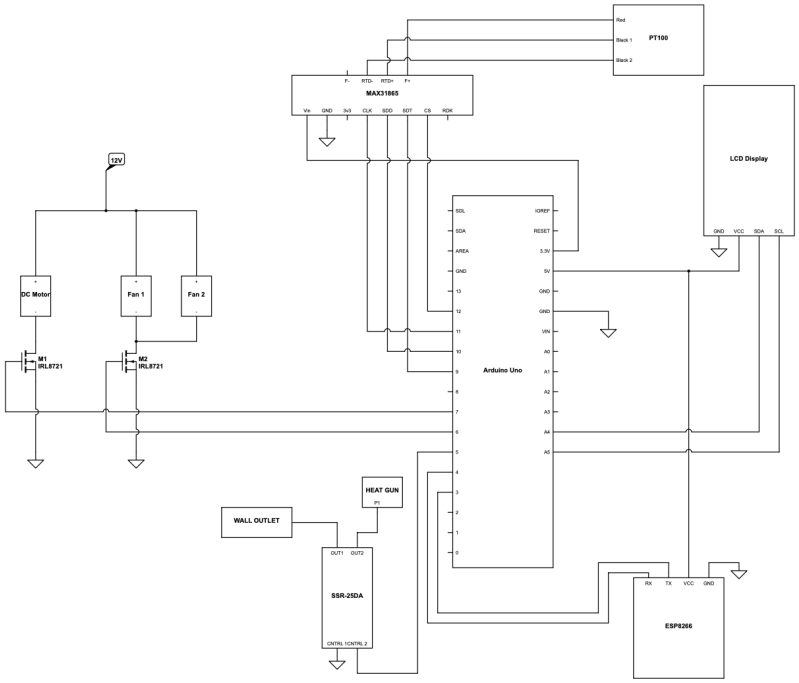

Figure 1: Project Circuit Diagram

Main Housing - Rick Cazenave

The housing design of this project incorporates many elements of traditional drum roasters and DIY projects. Figure 2 resembles the final design of the main housing. The main key features and components are as follows:

- 1500W dual temperature heating element [5]
- Modified tin can
- 12V DC motor [7]
- Cooling System
    - Cooling tray
    - 2 12V DC fans
- LCD Display [3]
- Wifi Shield [9,10]
- Box to contain electrical components
- Detachable end-piece for bean extraction
- Hinge that allows tin can to be rotated 180+ degrees from the heating element to the cooling system

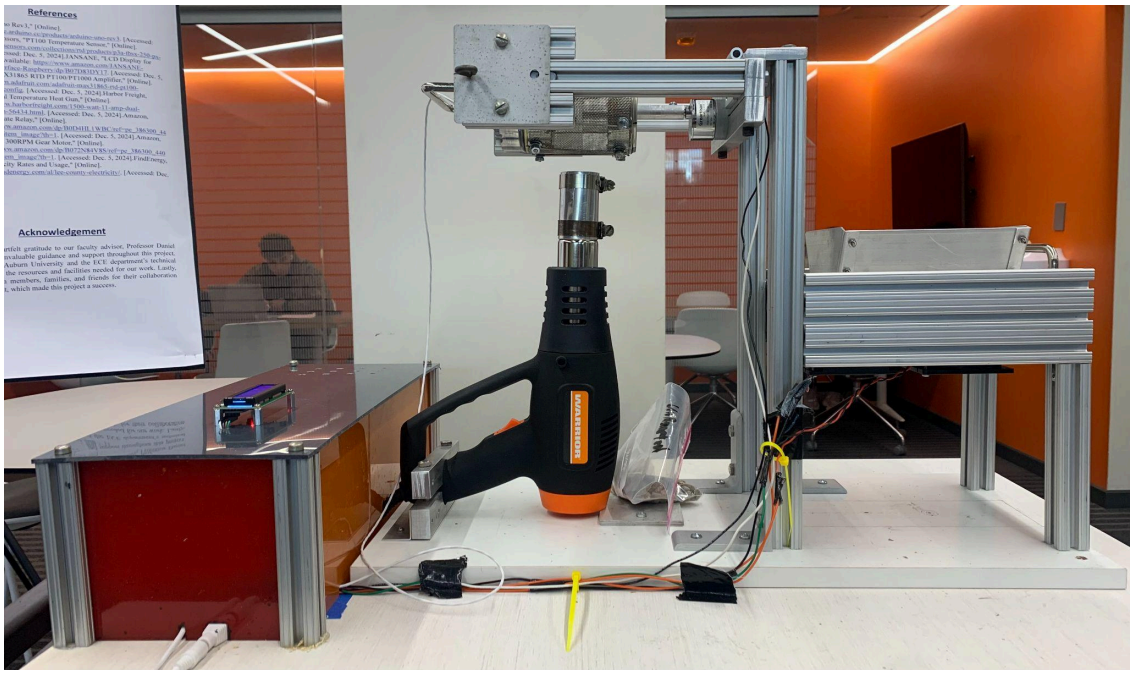

Figure 2a: Angle 1, side view

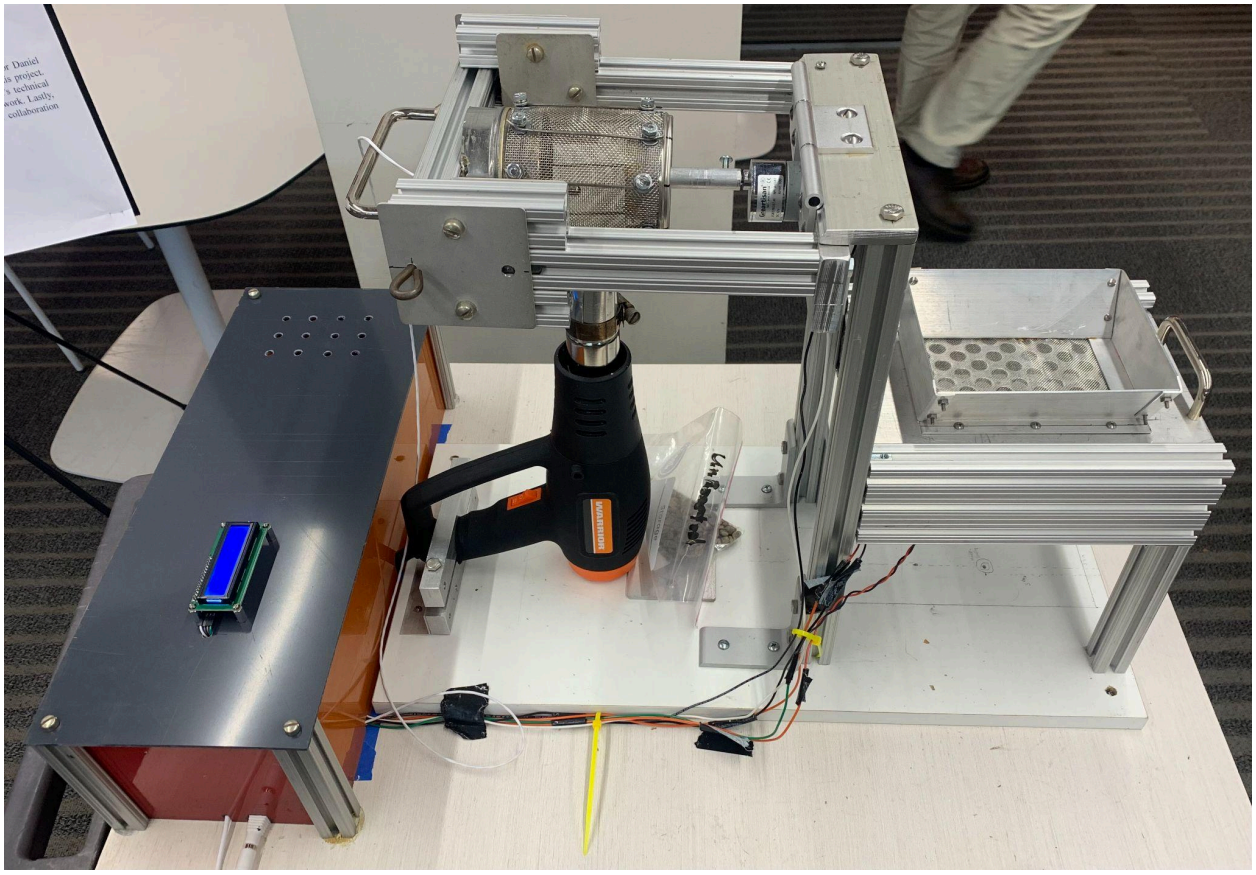

Figure 2b: Angle 2, upper side view

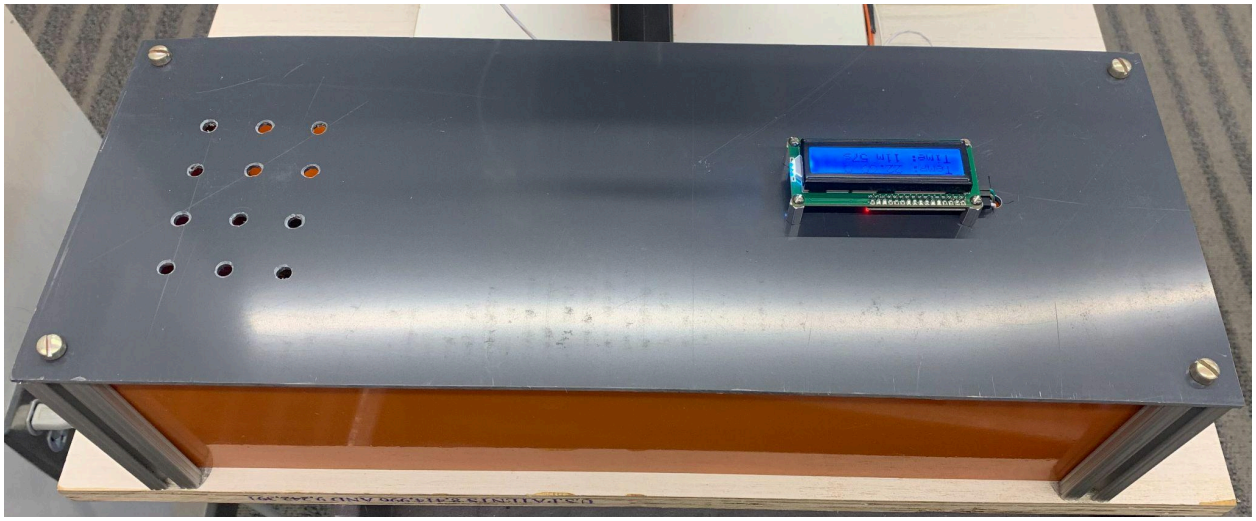

Figure 2c: Angle 3, casing for circuit

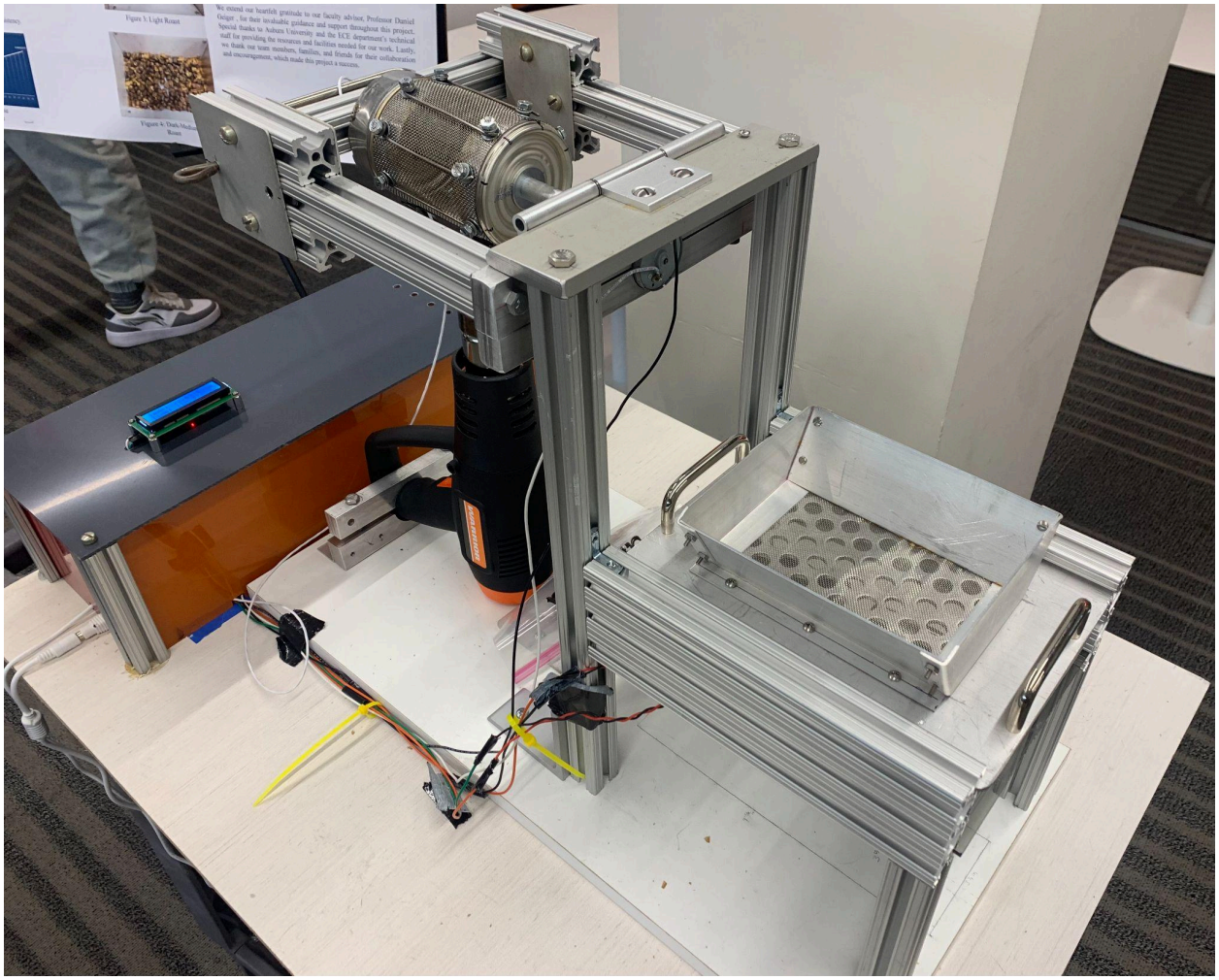

Figure 2d: Angle 4, cooling rack focus

Heating Element & Control Loop - Abdullah Alshehri

Heating Element: Adaptation and Performance

The 1500-watt heat device [5] was repurposed as the primary heating element for the coffee roasting system, providing the thermal energy required to achieve precise roasting profiles. The heating device's original factory controls [5], which allowed only limited manual temperature modulation, were removed to enable integration with an Arduino-based [1] control system. A Solid State Relay (SSR-25DA) [6] was employed for real-time power modulation,

allowing dynamic adjustment of the heat output. This modification, shown in the circuit diagram Fig. 3, was critical in achieving consistent temperature control and aligning the system with the design objectives.

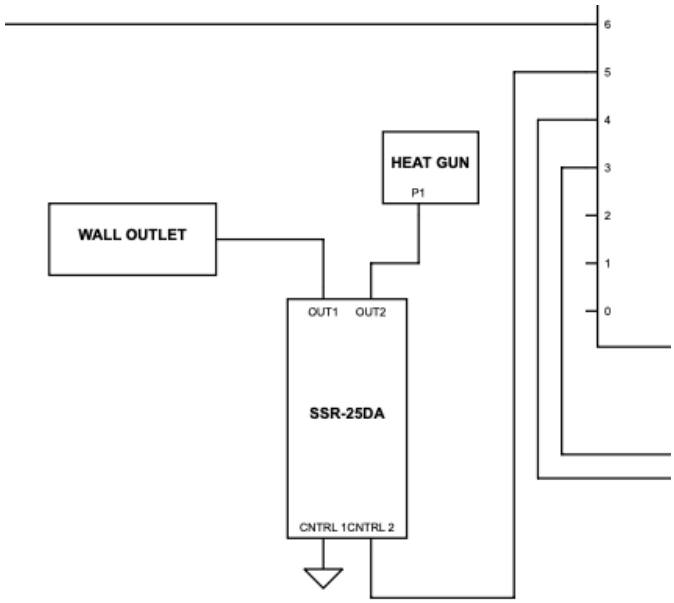

Figure 3: SSR & Heating Element

Initial testing of the heating element [5] revealed issues related to SSR [6] overheating, particularly during extended roasts at higher power levels. Overheating caused temporary performance dips and inconsistent heat delivery. To address this, a heat sink was integrated with the SSR [6], Fig.4, significantly improving its thermal dissipation and ensuring reliable operation over long roasting cycles. The upgraded design maintained consistent heat output, as observed in the temperature profiles. However, heat dissipation from the roasting chamber resulted in a maximum temperature cap of 189.5°C. This limitation affected the ability to achieve medium and dark roast profiles, as indicated in Table 1, which outlines the target versus achieved temperatures for each roast level.

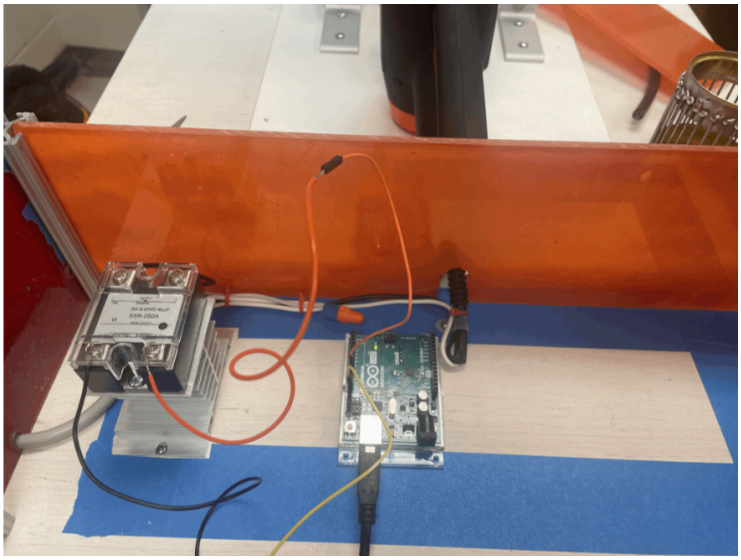

Figure 4: SSR & Heat Sink

Table 1: Target versus achieved temperatures for each roast profile

Roast Profile	Target Temperature (°C)	Actual Temperature (°C)	Roasting Time (Minutes)	Observations
Light	195	189.5	12	Even roast, good quality
Medium	210	189.5	18	Slight under-roast
Dark	225	189.5	23	Dark medium roast

The heat gun's [5] integration with the control system also included safety measures, such as temperature cutoffs and real-time monitoring, to prevent overheating or system failure. These measures ensured operational reliability and user safety, contributing to the project's overall success.

Control Loop: Design and Tuning

Temperature regulation in the coffee roasting system was implemented using a Proportional-Integral-Derivative (PID) control loop. The PID algorithm dynamically adjusted the power delivered to the heating element [5] to maintain the desired temperature throughout the roasting process. Early iterations of the control loop experienced issues such as significant overshoot and prolonged settling times, which disrupted roast consistency. Through iterative tuning of the PID parameters—proportional gain (Kp), integral gain (Ki), and derivative gain (Kd)—these issues were mitigated, resulting in stable and responsive temperature control.

The PID tuning process involved balancing the system's responsiveness and stability. The initial aggressive tuning caused oscillations, whereas overly conservative settings led to slow recovery from temperature disturbances. The final tuning achieved minimal overshoot and a recovery time of less than 10 seconds. The enhanced control loop minimized deviations from the target temperature and provided a smooth transition between power states. This performance is evident in the roast results in the next section, which demonstrate stable temperature control across multiple roast profiles.

One of the significant challenges during implementation was power cycling delays caused by the SSR [6]. These delays were addressed by increasing the sampling frequency and optimizing the control logic. Additionally, the PID controller’s performance was validated against various roast profiles, ensuring compatibility with light, medium, and dark roasts, even though the medium and dark profiles were limited by the system's maximum temperature.

Achievements and Challenges

The integration of the heating element [5] and PID control loop was pivotal in achieving the project’s objectives. Key achievements include the successful automation of the roasting process, stable temperature regulation, and consistent light roast results. The combination of the SSR-25DA [6] and PID algorithm allowed the system to dynamically respond to temperature

fluctuations with minimal manual intervention, demonstrating the feasibility of low-cost automation for small-scale coffee roasting.

Despite these successes, several challenges were encountered. The primary limitation was the system's inability to achieve target temperatures for medium and dark roasts due to heat dissipation from the roasting chamber. This limitation highlighted the need for enhanced thermal insulation or an enclosed chamber design to reduce heat loss. Furthermore, while the PID tuning significantly improved performance, future iterations could explore adaptive PID algorithms for even greater precision under varying load conditions.

These results and observations suggest opportunities for further improvement. For instance, refining the physical design of the roasting chamber, as well as exploring higher-capacity SSRs [6], could expand the system's capabilities to accommodate larger batches and more robust roast profiles. Such advancements would enhance the system's scalability and performance, aligning it more closely with commercial roasting solutions.

Roasting Results

The roasting results demonstrated the system's capability to produce consistent light roasts, while medium and dark roasts were constrained by the heating element's [5] maximum temperature. Testing was conducted using 50-gram batches of coffee beans, chosen for their optimal roast quality compared to larger batch sizes. Table 1 summarizes the target and achieved temperatures, along with the corresponding roast times for light, medium, and dark roast profiles.

Light Roast

The light roast achieved the best results, with the system maintaining a stable temperature of approximately 189.5°C for a 12-minute roasting cycle Fig. 5. Although the target temperature was slightly higher at 195°C, the flavor profile of the beans was consistent and uniform. This roast demonstrated the system's potential for precision and stability in achieving desired roast characteristics.

Figure 5: Light Roast

Medium Roast

For the medium roast, the target temperature was 210°C, requiring a roasting duration of 18 minutes. However, the system's maximum temperature limit of 189.5°C resulted in under-roasting. This is reflected in the physical appearance of the beans, which lacked the desired caramelization and flavor depth. While the PID control loop maintained stable temperature regulation within the achievable range, the inability to reach the target temperature affected the final quality of the roast Fig. 6.

Figure 6: Medium Roast

Dark Roast

The dark roast required a target temperature of 225°C over a 23-minute cycle. Similar to the medium roast, the system was unable to exceed 189.5°C, leading to an incomplete roast profile, Fig.7. The heat dissipation from the roasting chamber significantly impacted the system's performance at this level. The dark roast results highlight the limitations of the current design, emphasizing the need for improved thermal retention strategies, such as an enclosed roasting chamber or enhanced insulation materials.

Figure 7: Dark Roast

Analysis and Observations

Across all roast profiles, the system demonstrated precise temperature control within its operational range. Light roast results were consistent with expected standards, showcasing uniform bean color and optimal flavor. In contrast, the medium and dark roasts were impacted by temperature capping, resulting in suboptimal roast characteristics. In later sections of this report, power dissipation and cost will be discussed showing the cost per roast showing in the consumer's electricity bill per Lee County's rates.

Key Challenges

- **Temperature Limitation:** The maximum temperature of 189.5°C was insufficient for achieving medium and dark roast profiles, as detailed in Table I.
- **Heat Dissipation:** Significant heat loss from the roasting chamber reduced the system's overall efficiency, particularly for extended roast cycles.
- **Roast Quality:** While the light roast achieved satisfactory results, medium and dark roasts lacked the depth and uniformity required for their respective profiles.

Future Recommendations

To overcome the identified challenges, future iterations should focus on enhancing the thermal retention of the roasting chamber. This could include using food-grade insulated materials or redesigning the chamber to minimize heat loss. Additionally, exploring higher-capacity heating elements or increasing the efficiency of the current system through advanced PID tuning could improve performance across all roast profiles.

Temperature Display System - Yifan Zhu (Evan)

In this project, we design and implement a precise temperature monitoring and display system for a coffee bean roasting machine. Accurate temperature control is a critical aspect of the coffee roasting process, as it directly impacts the flavor profile and quality of the roasted beans. The main objective of this system is to provide accurate temperature input to other subsystems, ensure safety with an emergency shutdown feature when the temperature exceeds a critical threshold, and enable intuitive monitoring through real-time temperature and time displays. Additionally, the system supports remote monitoring via a web interface, offering convenience and enhanced functionality for users. This temperature system serves as a reliable and scalable foundation to achieve consistent roasting results while prioritizing user safety and operational efficiency.

Cost Analysis

- **PT100 Sensor**: The PT100 [2] is a high-precision temperature sensor, and it is more expensive than alternatives such as thermistors. However, its cost is justified given its stability and accuracy in high-temperature environments, which is crucial for precise temperature monitoring during coffee bean roasting.
- **MAX31865 Chip**: This chip is used to convert the PT100's [2] analog signal into a digital one, offering excellent precision. However, the MAX31865 [4] is relatively costly, especially in small-scale projects, where it can represent a significant portion of the budget.
- **LCD1602 and I2C Module**: The LCD1602 display [3] is cost-effective and easy to implement. The I2C module simplifies wiring and reduces hardware complexity. This combination offers good cost control and is well-suited for prototype development and small-scale production.
- **Arduino UNO Microcontroller**: The Arduino UNO [1] is an economical solution for such prototype projects. It is a powerful microcontroller capable of supporting various complex functions while keeping the costs reasonable.

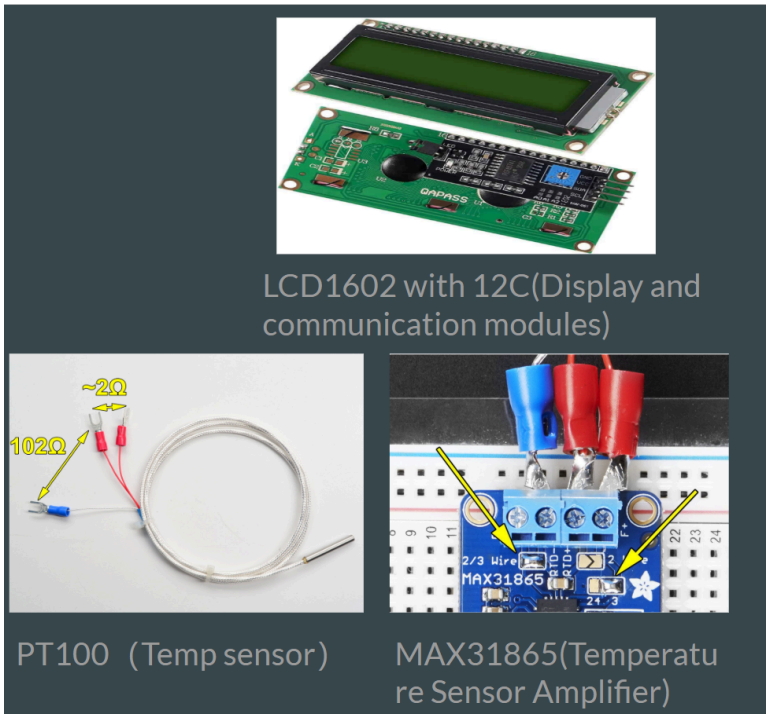

Figure 8: Temperature Display System Components Diagram

Scalability Analysis

- **Hardware Scalability**: The design uses I2C module and LCD1602 display [3], providing modular hardware interface and good scalability. More sensors (such as humidity or additional temperature sensors) can be added to expand the functionality. We have integrated wireless modules such as Wi-Fi or Bluetooth to achieve remote monitoring and control. In the future, if necessary, it can be integrated.
- **Software Scalability**: Based on the Arduino [1] development environment, the software can be easily expanded in the future. For example, we have used a PID control loop for more precise temperature control. In the future, cloud storage can be integrated for data analysis and recording. In addition, the software can be expanded to support customized smart reminder functions in the future. For example, when the temperature reaches the set threshold, it can not only trigger the buzzer alarm, but also send real-time notifications or alarm information through mobile applications.

Risk Analysis

- **Hardware Risks**:
    - **Overheating Risk**: Working in high-temperature environments over extended periods may overload the MAX31865 [4] or the PT100 [2]. It's important to design overheat protection mechanisms to prevent damage to the system.
- **Cost Risks**:
    - For mass production, designs based on PT100 [2] and MAX31865 [4] may be too costly and uncompetitive. If you plan to mass produce, you can consider lower-cost alternatives. ADS1115 is an alternative component to MAX31865 [4], which requires more hardware design but has a lower overall cost.

Implementation Method

In our project design, the temperature sensor component is crucial. The system will use the PT100 temperature sensor [2], MAX31865 analog-to-digital converter [4], LCD1602 display [3], and I2C module to form a complete temperature monitoring system. Below are the detailed implementation steps.

All Components Connection

- PT100 [2] is a resistive temperature sensor whose resistance changes as the temperature rises. We use the MAX31865 [4] chip to convert the analog signal of PT100 [2] into a readable digital signal. Since a 3-wire PT100 [2] is used, the red and blue wires should be properly connected to the RTD+, RTD-, and F+ pins of MAX31865 [4]. The red wire is connected to the F+ (Force Positive) pin of MAX31865 [4] to provide a stable excitation current for PT100 [2]. The two blue wires are connected to the RTD+ (Sensor Positive lead) and RTD- (Sensor Negative lead) pins respectively to detect the resistance change of PT100 [2].
- **MAX31865 Operation**: The MAX31865 [4] provides a stable current to the PT100 [2], measures the resistance, and converts it to temperature data through its internal ADC. The data is then transmitted to the microcontroller (e.g., Arduino [1]) for processing.
- **LCD1602 Display**: This is used to display temperature information. Due to its multiple pins, directly controlling the LCD1602 [3] would consume many of the microcontroller's

I/O resources. To simplify the wiring, we use an I2C expansion module to drive the LCD1602 [3].

- **I2C Module Connection**: The SDA and SCL pins of the I2C module are connected to the corresponding SDA and SCL pins on the Arduino [1], with VCC to power and GND to ground. Through the I2C communication protocol, temperature data can be sent to the LCD1602 [3] for real-time display.
- In the Arduino [1] programming, first initialize the MAX31865 [4] and read the PT100's [2] temperature data via SPI communication.
- The read temperature data is then transmitted to the LCD1602 [3] via I2C protocol and displayed in real time on the screen.

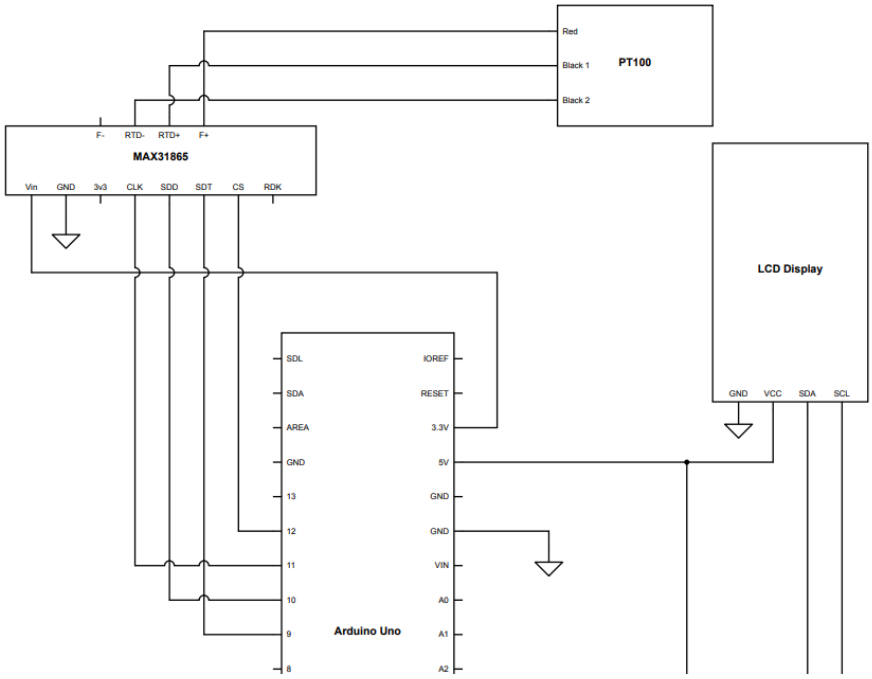

Figure 9: Temperature Display System Diagram

Temperature Sensor（Test Part）- Minmin Feng

The assistance of ChatGPT.com was used in the writing of this section

1. Introduction

To ensure the accuracy and reliability of the temperature sensor used in the smart coffee bean roasting machine, various tests were conducted across different temperature ranges. These tests aimed to validate the sensor's performance under low, medium, and high-temperature conditions.

---

2. Testing Equipment and Environment

- **Test Subject**: Temperature sensor
- **Reference Device**: Thermometer
- **Test Environment**: Indoor conditions, including cold water, hot water, body temperature simulation, and a 1500W heating element [5] for high temperatures.
- **Theoretical Temperature Range**:
    - Celsius: -70°C to 260°C
    - Fahrenheit: -90°F to 500°F
- **Data measured using a thermometer**:
    - Cold water: 5.7°C/7.0°C
    - Body temperature: 23.1°C
    - Room temperature: 35.6°C
    - Hot water: 72.6°C
- **Data measured using temperature sensors**:
    - Cold water: 6.01°C/7.08°C
    - Body temperature: 23.37°C
    - Room temperature: 35.36°C

- Hot water: 72.38°C

---

3. Testing Methodology

1. **Cold Water Test**: The sensor was immersed in cold water, and its reading was recorded.
2. **Room Temperature Test**: Sensor readings were recorded under ambient conditions.
3. **Body Temperature Test**: Simulated body temperature to test the sensor's accuracy.
4. **Hot Water Test**: Sensor performance was evaluated in medium temperature conditions using hot water.
5. **High-Temperature Test**: A heat gun was used to simulate extreme high temperatures, and the sensor's responsiveness was observed.

---

4. Data and Analysis

The temperature readings, sensor values, and relative errors are summarized below:

Temperature Point (°C)	Sensor Temperature (°C)	Relative Error (%)
7.0	7.08	1.14
23.1	23.37	1.17
35.6	35.36	0.67

5.7	6.01	5.44
72.6	72.38	0.30

Table 2: Temperature Accuracy Table

Temperature point (° C)	Temp sensor temperature (° C)	Relative error (%)
7	7.08	1.142857143
23.1	23.37	1.168831169
35.6	35.36	0.674157303
5.7	6.01	5.438596491
72.6	72.38	0.303030303

Theory Temperature Range (F)	Temp Sensor 1 (heat gun max) (C)	Thermometer temp (heat gun max)(C)
-90 to 500	196	207-260
Theory Temperature Range (C)	Temp Sensor 2 (heat gun max) (C)	
-70 to 260	203	

The relative error was calculated using the formula:

Error = (Temperature point - Temperature sensor)/ Temperature point %

---

5. Test Conclusion

- The temperature sensor demonstrated high accuracy within the tested range of 7°C to 72.6°C, with relative errors ranging from 0.30% to 5.44%.
- The sensor performed well during high-temperature testing, closely matching the reference device readings, making it suitable for the high-temperature requirements of coffee bean roasting.
- The higher relative error in cold water tests (5.44%) suggests the need for calibration in low-temperature environments.

- I used two temperature sensors, and the measured temperatures were 5.7 degrees Celsius and 7 degrees Celsius respectively. We can see that there are differences between different sensors.
- According to the data, it is good to measure the temperature at which our project coffee bean roaster actually works.

---

6. Recommendations

- Calibrate the sensor for improved accuracy in cold environments.
- Conduct further testing across a wider temperature range (e.g., -70°C to 260°C) to verify its stability and reliability under extreme conditions.

---

7. Experimental Images

The following image illustrates the experimental setup, showcasing the hot water temperature measurement process and equipment used:

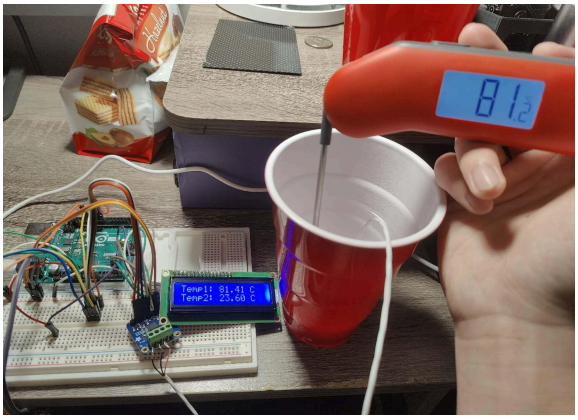

Figure 10: Live image of the temperature meter and two temperature sensors measuring

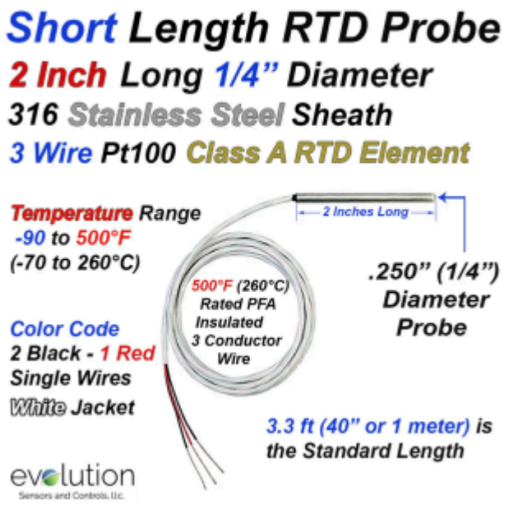

Figure 11: PT100 Temperature Sensors

The Digital Display - Minmin Feng

1. Introduction

In the smart coffee bean roasting machine, a digital display plays a critical role in monitoring the roasting process. It provides real-time data on temperature and roasting time, enabling users to manage and control the roasting process effectively. This report highlights the function and testing of the digital display.

---

2. Purpose of the Digital Display

The digital display serves the following purposes:

1. **Real-Time Monitoring**: Displays the current temperature inside the roasting chamber.
2. **Roasting Progress**: Tracks and shows the elapsed roasting time, helping users to assess the progress and stop roasting at the optimal point.
3. **Control and Decision-Making**: Allows users to identify critical parameters such as the roasting time limit and actual temperature.

---

3. Display Features

The digital display used in this project shows two primary metrics:

- **Temperature Readings**: Dual-temperature display, providing data from two sensors (e.g., Temp1: 81.41°C and Temp2: 23.60°C).
- **Time Tracking**: Shows elapsed time during the roasting process, e.g., "Time: 0m 55s."

These features ensure precise monitoring and better control of the roasting process.

---

4. Testing and Results

The digital display was tested to validate its functionality under different conditions:

1. **Temperature Measurement**: The display accurately captured and showed temperature data from two separate sensors.
2. **Time Tracking**: The elapsed time counter functioned as expected, updating in real-time during the roasting process.
3. **High-Temperature Testing**: The display performed well under high-temperature conditions, such as readings exceeding 180°C, matching the requirements for coffee roasting.

The setup involves connecting the sensors and the display to a microcontroller, ensuring real-time data transmission.

---

5. Observations

- The dual-display functionality allows users to compare two temperature readings simultaneously, which is helpful for monitoring different zones in the roasting chamber.

- The temperature and time readings are clear and easy to read, enhancing user experience and operational efficiency.
- The integration of the digital display with the system aids in identifying critical parameters, such as temperature thresholds and optimal roasting times.
- The JANSANE 16x2 1602 LCD display [3] plays a vital role in **data visualization, user monitoring, system debugging, and user experience enhancement**, making it a key component of your intelligent roasting system. By optimizing its features and incorporating interactive designs, it can further improve the system's intelligence and user-friendliness.

---

6. Experimental Images

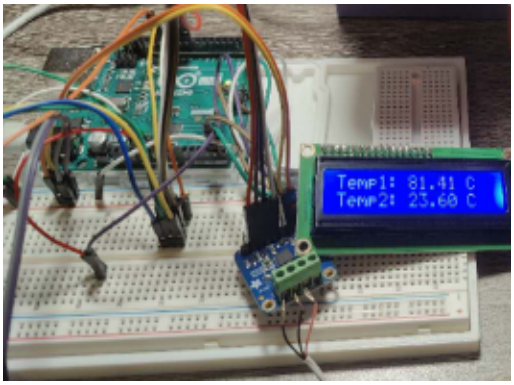

Figure 12: Dual temperature sensors displaying readings on the screen.

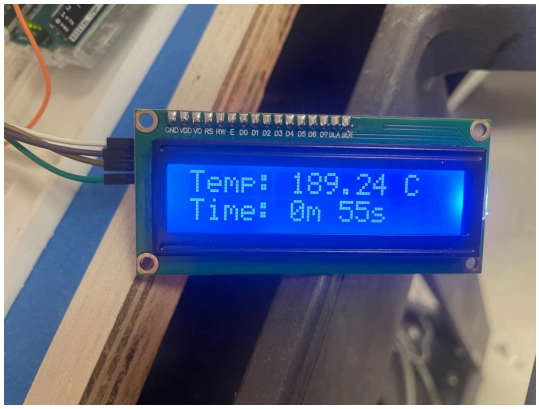

Figure 13: High-temperature reading (189.24°C) and elapsed time tracking (0m 55s).

Figure 14: JANSANE 16x2 1602 LCD Display

DC Motor - Rick Cazenave

The DC motor [7] plays a crucial role in ensuring the coffee beans are roasted evenly. It is mounted at the closed end of the tin can, where it rotates the can's endpiece in a manner similar to a lathe (Figure()). This continuous motion effectively cycles the beans, allowing them to receive a uniform distribution of heat throughout the roasting process. The way it is controlled is through a MOSFET switching circuit where a signal from the Arduino [1] turns on the MOSFET and allows current to flow through the motor [7] (Figure 15). The DC motor [7] will dissipate about 4.35W during a roast.

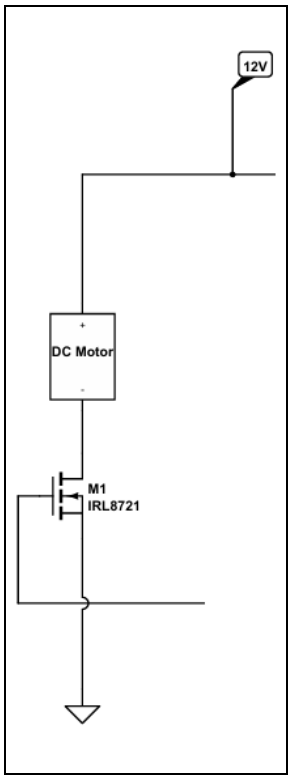

Figure 15: MOSFET switching circuit, Motor portion

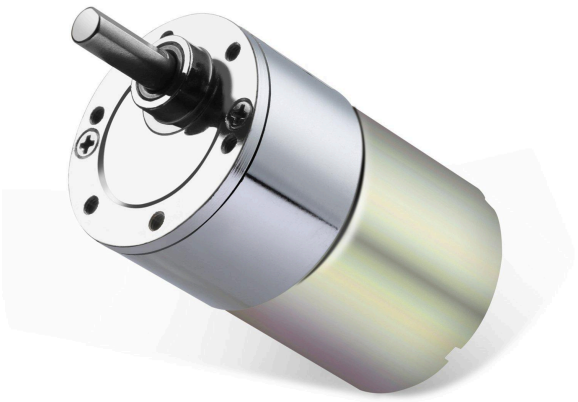

Figure 16: Greartisan 12V DC motor

Cooling System - Zayvier Hambrick

Freshly roasted coffee beans are incredibly hot and must be cooled rapidly to halt the roasting process and prevent them from overcooking or even burning, which could ruin their flavor. To achieve this, an effective cooling system is essential, as it ensures the beans are brought to an optimal temperature quickly and evenly. In this system, a specially designed structure, comprising strategically placed fans and a perforated tray, is utilized to provide consistent airflow and efficient cooling (Figure 17).

Figure 17: The Coffee Bean Roaster Cooling System

The cooling system was initially intended to be integrated into the heating element by modifying it to blow both hot and non-heated air. However, this approach was abandoned due to safety concerns, particularly the challenges associated with using AC power. Instead, it was redesigned as a separate component consisting of two main parts: a tray, to hold the hot coffee beans and fans, to cool them effectively. This design ensures safer operation while providing greater customization of airflow and temperature control.

Tray

The tray is a specially designed, heat-resistant, food-safe container capable of withstanding the intense temperatures of freshly roasted beans (Figure 18). The container features a removable stainless steel tray with a stainless steel mesh, allowing efficient cooling while also making it easy to pour the beans directly into any desired container for storage or use. As soon as the roasting timer is complete and the beans are finished roasting, they are emptied into the cooling tray and the fans are then activated.

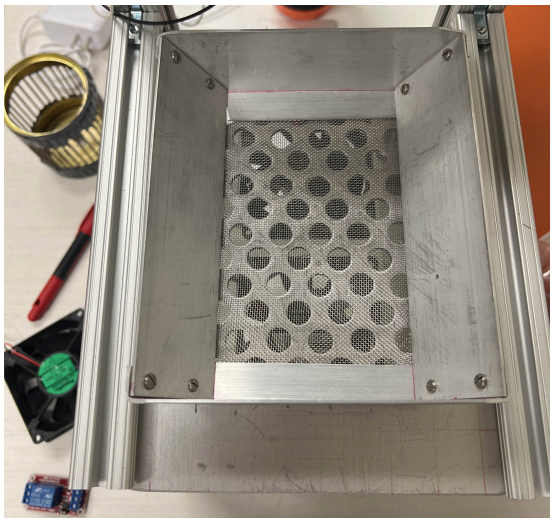

Figure 18: The Tray component within the Cooling System.

Fans

The cooling system has two 12V fans that pull air through the bottom of the beans, ensuring optimal cooling with room-temperature air, effectively halting the roasting process (Figure 19). These fans are controlled by a MOSFET switch in conjunction with an Arduino, which allows them to be turned on and off for set cooling time durations based on the roast type of the coffee beans, due to the varying coffee bean cooking temperatures (Figure 20). The fans stop the beans from continuing to cook, preserving their optimal flavor and preventing any burnt or bitter taste from developing. This system not only ensures that every bean cools evenly but also guarantees a high-quality finish , giving the beans the perfect balance of roast and flavor.

Ultimately, the cooling system is a critical component in delivering consistently great-tasting coffee.

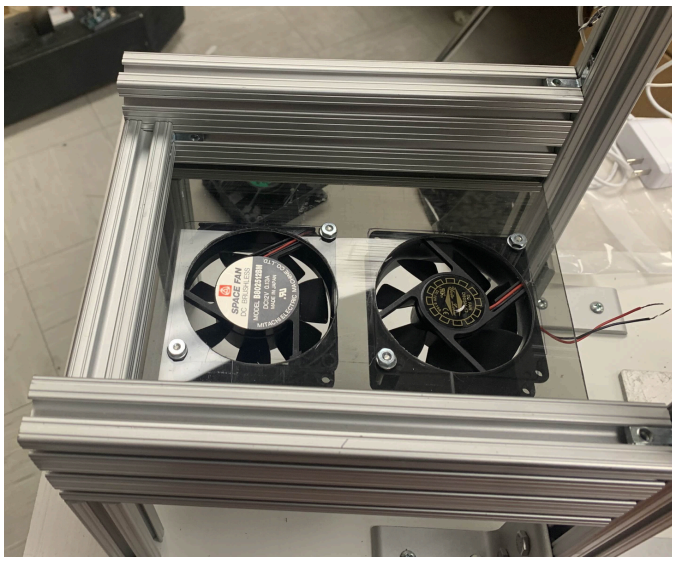

Figure 19: The two 12V fans secured into the system with plexiglass.

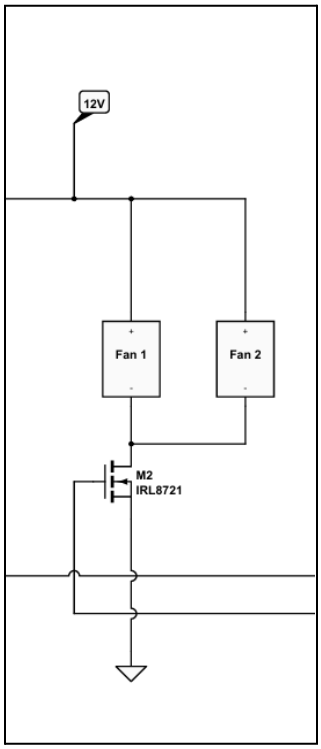

Figure 20: MOSFET switch circuit, fan portion.

Power Dissipation - Rick Cazenave

The total power dissipation and cost was an important consideration for the Roastology team. There are four main power drawing components: the heating element [5], the Arduino UNO [1], the DC motor [7], and the two DC fans. Table 3 represents the measured data of all power drawing components in the project. Using the energy equation in Figure 21, the energy used for a single 12 minute roast in kilowatt-hour is 0.261kWh. With this result, the cost equation is then used in Figure 22 along with the average electricity rate in Lee County Alabama [8] which equals approximately $0.04 or 4 cents per roast. Figure 23 represents a graph that corresponds to the time use versus the total cost.

Table 3: Power dissipation results

Component	Power Draw (Measured in Watts)
Heating Element	1297.7W
DC Motor	4.35W
DC Fans (Both)	1.97W
Arduino UNO	1.93W
Total	1305.95W

$$Energy(kWh) = \frac{Power(W) * Time(hours)}{1000}$$

Figure 21: Energy equation

$$Cost = Energy(kWh) * Rate(dollars/kWh)$$

Figure 22: Cost equation

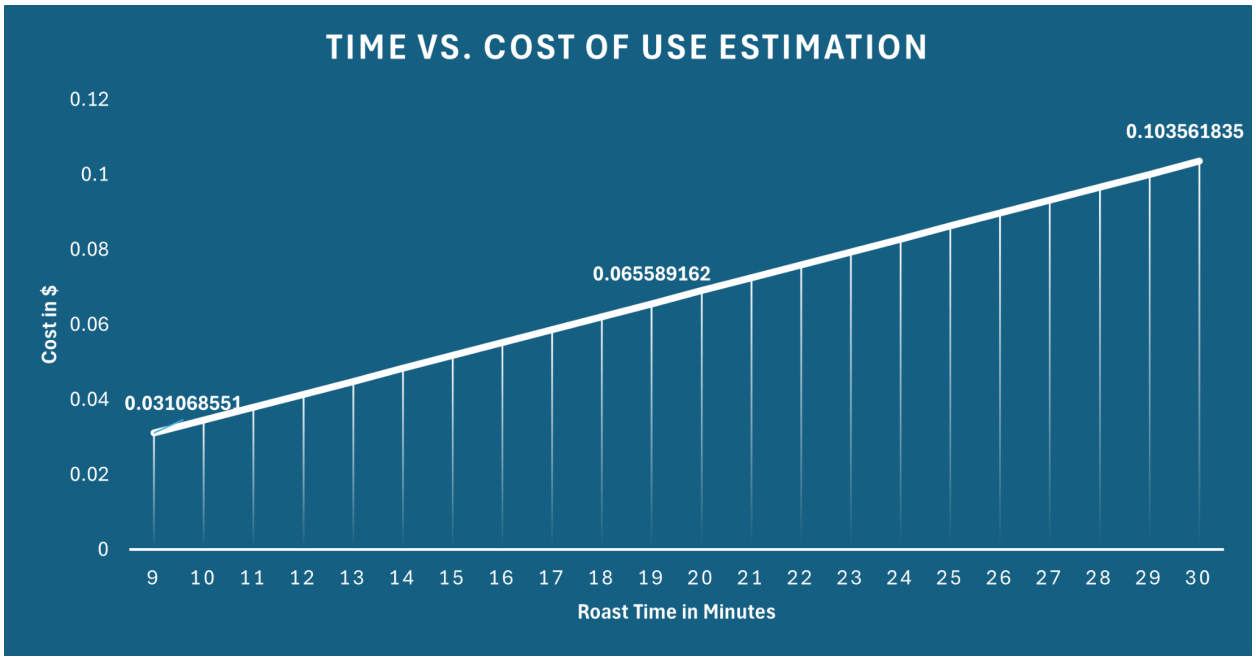

Figure 23: Cost of use per roast profile

Web Server & Wifi Shield - Alex Brown

The ESP 8266 ESP-01 Wifi Module [9] was utilized to create and host a webserver that could be accessed from any device to monitor and control the roaster. This module was connected to the Arduino Uno using an ESP-01 adapter module [10] to allow for use with the Arduino uno. The webpage can be seen in Figure 24.

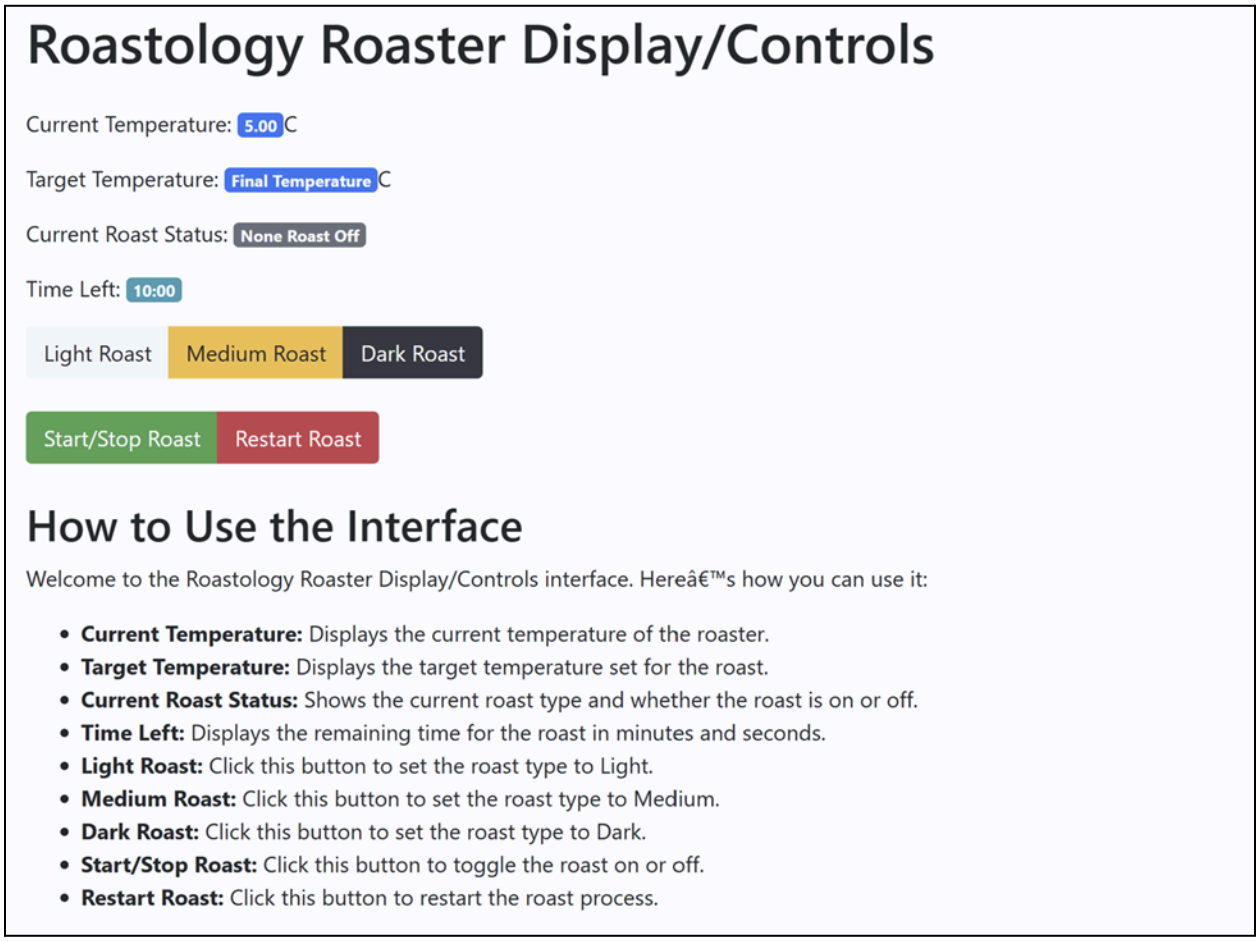

Figure 24: Roastology Web Page

The ESP8266 acts as the web server and handles WiFi connectivity, and it communicates with the Arduino Uno to receive updates as it manages the roasting process. Upon startup, the ESP8266 attempts to connect to a specific WiFi network. If the ESP8266 is unable to connect, it creates an Access Point for devices to connect to. Upon connecting to the WiFi network or creation of the Access Point, the IP address of the device is displayed on the LCD display.

The webpage is created using HTML to create displays from the temperature, time, and roast status, and to create buttons to control the roaster. JavaScript is used to allow the displays and buttons to receive and send HTTP requests. The webpage is updated each second with the current information received from the Arduino.

The ESP8266 and Arduino communicate using serial communication. The ESP8266 and Arduino send commands by sending strings of text though the serial port that they each can interpret and act upon. Each second, the ESP8266 send the GET_ALL_DATA command, and the Arduino will send a string with the current temperature, time, roasting profile, and if a roast is taking place. The ESP can also send commands to update roasting profiles and to start/stop the roasting process. Commands are sent in a que to ensure a response is received before executing another command.

Summary - Rick Cazenave

The assistance of ChatGPT.com was used in the writing of this section

The Roastology Project developed a smart coffee bean roasting system combining automation, real-time monitoring, and user-friendly controls. Designed by Auburn University's Electrical and Computer Engineering team, the project incorporates advanced technologies to improve roasting precision and quality.

Key Features

1. **Heating System**:
    - A 1500W heating element controlled by a PID system regulates temperature, limited to 189.5°C, affecting medium and dark roasts.
2. **Temperature Monitoring**:
    - A PT100 sensor with a MAX31865 amplifier ensures precise readings, displayed on an LCD and accessible via a web interface.
3. **Bean Rotation**:
    - A 12V DC motor rotates a modified tin can for even roasting, managed through a MOSFET circuit.

4. **Cooling System**:
    - A tray with DC fans quickly cools beans post-roast, preserving flavor and preventing overcooking.
5. **Web Server**:
    - An ESP8266 module provides remote control of roast profiles and real-time monitoring.

Achievements

- Automated roasting process successfully produces consistent light roasts.
- Cost-effective operation at $0.04 per roast.
- Integrated remote monitoring and control.

Challenges

- Limited temperature capacity prevents optimal medium and dark roasts.
- Heat loss from the chamber reduces efficiency.
- Scaling for larger batches and varied roast profiles requires further work.

Recommendations

- Improve thermal retention with better insulation or enclosed chambers.
- Upgrade heating elements for higher temperatures.
- Enhance PID control for precision and scalability.

The project demonstrates innovative engineering with practical applications, providing a strong foundation for future development in coffee roasting technologies.

References - Rick Cazenave

[1] Arduino, "Arduino Uno Rev3," [Online]. Available: https://store.arduino.cc/products/arduino-uno-rev3. [Accessed: Dec. 5, 2024].

[2] Evo Sensors, "PT100 Temperature Sensor," [Online]. Available: https://evosensors.com/collections/rtd/products/p3a-tbsx-250-px-2-pfxx-40-stwl. [Accessed: Dec. 5, 2024].

[3] JANSANE, "LCD Display for Arduino," [Online]. Available: https://www.amazon.com/JANSANE-Arduino-Display-Interface-Raspberry/dp/B07D83DY17. [Accessed: Dec. 5, 2024].

[4] Adafruit, "MAX31865 RTD PT100/PT1000 Amplifier," [Online]. Available: https://learn.adafruit.com/adafruit-max31865-rtd-pt100-amplifier/rtd-wiring-config. [Accessed: Dec. 5, 2024].

[5] Harbor Freight, "Warrior 1500W Dual Temperature Heat Gun," [Online]. Available: https://www.harborfreight.com/1500-watt-11-amp-dual-temperature-heat-gun-56434.html. [Accessed: Dec. 5, 2024].

[6] Amazon, "SSR-25DA Solid State Relay," [Online]. Available: https://www.amazon.com/dp/B0D4HL1WBC/ref=pe_386300_440135490_TE_simp_item_image?th=1. [Accessed: Dec. 5, 2024].

[7] Amazon, "Greartisan DC 12V 300RPM Gear Motor," [Online]. Available: https://www.amazon.com/dp/B072N84V8S/ref=pe_386300_440135490_TE_simp_item_image?th=1. [Accessed: Dec. 5, 2024].

[8] FindEnergy, "Lee County Electricity Rates and Usage," [Online]. Available: https://findenergy.com/al/lee-county-electricity/. [Accessed: Dec. 5, 2024].

[9] AITRIP, "AITRIP ESP8266 Serial Wireless Transceiver WiFi Module, Compatible with Arduino," Amazon. [Online]. Available:

https://www.amazon.com/AITRIP-ESP8266-Wireless-Adapter-Compatible/dp/B0872XG31W/. [Accessed: 09-Dec-2024].

[10] AITRIP, "AITRIP ESP8266 Serial WiFi Wireless ESP-01 Adapter Module, Compatible with Arduino," Amazon. [Online]. Available: https://www.amazon.com/AITRIP-ESP8266-Wireless-Adapter-Compatible/dp/B09FDNY8QV/. [Accessed: 09-Dec-2024].

Appendices

Appendix #1

Rick Cazenave - ELEC - DC Motor, Housing, Prototype Modeling

Alex Brown - ELEC - Remote Monitoring

Abdullah Alsheri - WREE - Heating Element

Minmin Feng - ELEC - Temperature Sensor

Zayvier Hambrick - CMPE - Cooling System

Yifan Zhu - CMPE - Microcontroler and Display

Weekly team meetings were conducted in-person on campus at Auburn University and virtually where team members shared their progress and discussed any ideas or improvements to be made. These meetings were led by Rick Cazenave, the team leader.

Appendix #2

```
//Current Instructions:

//After uploading to board, you have two options to control the roast
process:
//1.Look for Roastology in your WiFi settings and connect to it.
//Open a browser and type in 192.168.4.1 to access the web interface.
//2. Connect the ESP8266 to your local WiFi network and access the web
interface by typing in the IP address displayed on the serial monitor.
//The IP address will be displayed in the serial monitor after the ESP8266
has successfully connected to the network.

//You can now control the roast process from the web interface.

//Roasting profiles can only be changed when the roast is off.
//To change the roast profile, turn off or restart the roast, select the
desired profile, and turn the roast back on.

#include <Arduino.h>
#include <SoftwareSerial.h>
```

```
#include <Wire.h>
#include <Adafruit_MAX31865.h>
#include <LiquidCrystal_I2C.h>
// #include <PID_v1_bc.h>  // Import the PID library (Rick's version)
#include <PID_v1.h>  // Import the PID library(Alex's version)

// Pin assignments
SoftwareSerial ESP8266(3, 4); // RX, TX
int SSRPin = 5;                   // Pin connected to SSR
const int motorPin = 7;           // Pin connected to motor
const int fanPin = 6;             // Pin Connected to Fans
double TemperatureInput, Output;            // Temperature input and PID
output
double Kp = 2, Ki = 5, Kd = 1;    // PID constants for tuning
double targetTemperature;         // Target temperature for the roast
double maxTemperature = 235.0;    // Safety temperature limit in Celsius
unsigned long elapsedTime;
// unsigned long roast_type;
unsigned long roastTime;          // Roast time in milliseconds (set per
profile)
unsigned long startTime;          // To track the start time
unsigned long remainingTime = 0; // Remaining time to complete the roast
bool roast_on = false;            // Flag to control the roast process
bool roasting_restarted = true; // Flag to check if roasting is restarted
String roast_string = "none";             // Roast type string
String ipAddress = "Waiting..."; // IP address of the ESP8266
String AccessMode = "Waiting...";       // Access Point Mode

//============================== TEST
================================================================================
============================
//bool roastComplete = false;      // Set flag for roast completion
//================================================================================
================================================================

// PID object
PID myPID(&TemperatureInput, &Output, &targetTemperature, Kp, Ki, Kd,
DIRECT);
```

```
// MAX31865 settings for PT100
Adafruit_MAX31865 max2 = Adafruit_MAX31865(12, 9, 10, 11); // MAX31865:
CS, DI, DO, CLK
#define RREF 430.0  // Rref resistor value for the PT100

// Initialize LCD (I2C, default address is 0x27, 16 columns and 2 rows)
LiquidCrystal_I2C lcd(0x27, 16, 2);

// Roast type definitions
enum RoastType { LIGHT, MEDIUM, DARK };
RoastType currentRoast;

void setup() {
  Serial.begin(9600);  // Start serial communication for roast selection
  ESP8266.begin(9600); // Start serial communication with ESP8266

  pinMode(SSRPin, OUTPUT);   // SSR output control
  pinMode(motorPin, OUTPUT); // Motor output control
  pinMode(fanPin, OUTPUT);   // Fan Control
  digitalWrite(SSRPin, LOW); // Ensure SSR is off initially
  digitalWrite(motorPin, LOW); // Ensure motor is off initially
  digitalWrite(fanPin, LOW);   // Ensure fans are off initially

  lcd.init();                // Initialize LCD
  lcd.backlight();           // Turn on LCD backlight
  lcd.setCursor(0, 0);
  lcd.print("Temp: ");       // Display initial text

  max2.begin(MAX31865_3WIRE);  // Set MAX31865 to 3-wire mode
  myPID.SetMode(AUTOMATIC);    // Initialize the PID controller in
automatic mode

}

double readTemperature() {
  // Read temperature from the PT100 sensor using MAX31865
  float RTDtemp = max2.temperature(100, RREF);  // PT100, Rref=430.0
  return RTDtemp;
}
```

```
RoastType previousRoast; // Initialize previous roast type to Light

void setRoastProfile(RoastType roastType) {
  switch (roastType) {
    case LIGHT:
      targetTemperature = 195;       // Fill in your target temperature
for Light Roast

      //
------------------------------------------------------------------------
------------------------------------------------------------------
      // CHANGE MADE
      roastTime = 720000;            // Fill in the roast time for Light
Roast (in milliseconds)
      //
------------------------------------------------------------------------
------------------------------------------------------------------
      //check previous roast type
      if (previousRoast != roastType) {
        //reset roast time
        Serial.println("Profile: Light Roast");
      }
      //update previous roast type
      previousRoast = roastType;
      break;

    case MEDIUM:
      targetTemperature = 210;  // Fill in your target temperature for
Medium Roast
      roastTime = 900000;            // Fill in the roast time for Medium
Roast (in milliseconds)
      //check previous roast type
      if (previousRoast != roastType) {
        Serial.println("Profile: Medium Roast");
      }
      //update previous roast type
      previousRoast = roastType;
      break;

    case DARK:
```

```
      targetTemperature = 225;  // Fill in your target temperature for
Dark Roast
      roastTime = 1080000;            // Fill in the roast time for Dark
Roast (in milliseconds)
      //check previous roast type
      if (previousRoast != roastType) {
        Serial.println("Profile: Dark Roast");
      }
      //update previous roast type
      previousRoast = roastType;

      break;
  }
}

void StartRoast() {
  if (roasting_restarted) {        // Check if roasting begining and not
resuming from a pause
    startTime = millis();          // Record start time for roasting
    roasting_restarted = false;    // Reset the restart flag
  }
  digitalWrite(SSRPin, HIGH);    // Turn on SSR
  digitalWrite(motorPin, HIGH);  // Turn on the motor at the start of
roasting
  roast_on = true;               // Set roast_on flag to true
}

void StopRoast() {
  digitalWrite(SSRPin, LOW); // Turn off SSR
  digitalWrite(motorPin, LOW); // Turn off motor
  roast_on = false; // Set roast_on flag to false
}

void promptSerialRoastType() {
  Serial.println("Select Roast Type: ");
  Serial.println("1. Light");
  Serial.println("2. Medium");
  Serial.println("3. Dark");

  while (true) {
```

```
    if (Serial.available() > 0) {
      char selection = Serial.read();
      Serial.println(selection);

      switch (selection) {
        case '1':
          currentRoast = LIGHT;
          setRoastProfile(LIGHT);
          return;
        case '2':
          currentRoast = MEDIUM;
          setRoastProfile(MEDIUM);
          return;
        case '3':
          currentRoast = DARK;
          setRoastProfile(DARK);
          return;
        default:
          Serial.println("Invalid selection. Please try again.");
          break;
      }
    }
  }
}

void ESPComunication() {
  if (ESP8266.available()) {
    // Read the incoming string
    String incomingString = ESP8266.readStringUntil('\n');
    incomingString.trim(); // Remove any leading/trailing whitespace

    // Determine the command based on the incoming string
    if (incomingString == "GET_ALL_DATA") {
      Serial.println("Sending all data");
      ESP8266.println(TemperatureInput);
      // Serial.println(TemperatureInput);
      ESP8266.println(remainingTime);
      // Serial.println(remainingTime);
      ESP8266.println(roast_string);
      Serial.println(roast_string);
```

```
      ESP8266.println(roast_on?"true":"false");
      // Serial.println(roast_on?"true":"false");
      // //Target Temperature
      ESP8266.println(targetTemperature);
      // Serial.println(targetTemperature);
    } else if (incomingString == "UPDATE_ROAST_TYPE_LIGHT") {
      Serial.println("Updating roast type to Light");
      currentRoast = LIGHT;
      roast_string = "Light";
    } else if (incomingString == "UPDATE_ROAST_TYPE_MEDIUM") {
      Serial.println("Updating roast type to Medium");
      currentRoast = MEDIUM;
      roast_string = "Medium";
    } else if (incomingString == "UPDATE_ROAST_TYPE_DARK") {
      Serial.println("Updating roast type to Dark");
      currentRoast = DARK;
      roast_string = "Dark";
    } else if (incomingString == "TURN_ROAST_ON") {
      Serial.println("Turning roast on");
      StartRoast();
    } else if (incomingString == "TURN_ROAST_OFF") {
      Serial.println("Turning roast off");
      StopRoast();
    } else if (incomingString == "RESTART_ROAST") {
      StopRoast();
      roasting_restarted = true;
    } else if (incomingString.startsWith("IP_ADDRESS:")){ // Check if
ESP8266 has connected to a network{
      ipAddress = incomingString.substring(11); // Extract the IP address
from the incoming string
      Serial.println("Successfully connected");
      Serial.print("Received IP Address: ");
      Serial.println(ipAddress);
      AccessMode = "STA";
    } else if (incomingString.startsWith("ACCESS_POINT:")){ // Check if
ESP8266 has started an access point
      String SSID = incomingString.substring(13); // Extract the SSID from
the incoming string
      Serial.print("Access Point Started: ");
      Serial.println(SSID);
```

```
      incomingString = ESP8266.readStringUntil('\n'); // Read the next
line
      Serial.print("Access Point IP: ");// Extract the IP address from the
incoming string
      Serial.println(incomingString);
      ipAddress = incomingString;
      AccessMode = "AP";
    } else if (incomingString.startsWith("Trying to connect with:")){ //
Check if ESP8266 is trying to connect to a network
      String networkName = incomingString.substring(24); // Extract the
network name from the incoming string
      Serial.print("Trying to connect to network: ");
      Serial.println(networkName);
    }
    else {
      Serial.println("Invalid command");
    }
  }
}

void loop() {

  ESPComunication();  // Check for incoming commands from ESP8266

  if((millis() - startTime) % 1000 == 0){ //Perform every second

    // Read temperature from PT100 sensor
    TemperatureInput = readTemperature();
    // Serial.print("Current Temp: ");
    // Serial.print(TemperatureInput);
    // Serial.print(" Â°C, Elapsed Time: ");
    // Serial.print(elapsedTime / 1000); // Print elapsed time in seconds
    // Serial.println(" sec");
    if (roast_on == false) {
      //Perform when roasting is off
      remainingTime = ((roastTime > elapsedTime) ? (roastTime -
elapsedTime) : 0) / 1000;
      setRoastProfile(currentRoast); // Set the selected roast profile
      //Update LCD with Access mode
      lcd.setCursor(6,0);
```

```
      lcd.print("AccessMode: ");
      lcd.print(AccessMode);
      lcd.print(" ");
      // Update LCD with IP address
      lcd.setCursor(0, 1);
      lcd.print("IP: ");
      lcd.print(ipAddress);
      lcd.print(" ");

    } else{ // Perform when roasting is on
      // calculate mins or seconds
      elapsedTime = millis() - startTime;
      remainingTime = ((roastTime > elapsedTime) ? (roastTime -
elapsedTime) : 0) / 1000;
      int remainingMinutes = remainingTime / 60;
      int remainingSeconds = (remainingTime % 60);



      // Safety stop condition for max temperature
      if (TemperatureInput > maxTemperature) {
        Serial.println("Safety Stop: Temperature exceeded limit. Turning
off heat gun and motor.");
        StopRoast();
        roasting_restarted = true;
        digitalWrite(fanPin, HIGH);  // Turn on Fans
        lcd.setCursor(0, 1);
        lcd.print("Overheat STOP!"); // Display overheat stop message
      }

    //============================== Original Code as of 12/02/2024
=============================================================================


      // Stop condition based on roasting time
      if (elapsedTime >= roastTime) {
        Serial.println("Roast Complete: Time limit reached. Turning off
heat gun and motor.");
        digitalWrite(SSRPin, LOW);  // Turn off SSR
        digitalWrite(motorPin, LOW); // Turn off motor
        digitalWrite(fanPin, HIGH);  // Turn on Fans
```

```
        lcd.setCursor(0, 1);
        lcd.print("Time Limit STOP"); // Display time limit stop message
        while (true);  // Halt execution indefinitely
      }

    //
==========================================================================
============================================================

      // Run PID control loop to adjust SSR output if within time and
temperature limits
      myPID.Compute();
      analogWrite(SSRPin, Output); // Send calculated output to the SSR

      // Update LCD with current temperature
      lcd.setCursor(6, 0);
      lcd.print(TemperatureInput);
      lcd.print(" C   ");  // Clear trailing characters
      // update lcd with current spend time
      lcd.setCursor(0, 1);
      lcd.print("Time: ");
      lcd.print(remainingMinutes);
      lcd.print("m ");
      lcd.print(remainingSeconds);
      lcd.print("s ");
    }
  }
}
```

Appendix #3 (ESP8266 Code)

```
#include <DNSServer.h>
#ifdef ESP32
#include <WiFi.h>
#include <AsyncTCP.h>
#elif defined(ESP8266)
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#endif
#include "ESPAsyncWebServer.h"
```

```
#include <queue>
#include <string>



std::queue<String> commandQueue;

// DNSServer dnsServer;
AsyncWebServer server(80);

float temperature = 5;
int time_left = 600;
String roast_type = "None";
bool roast_on = false;

//Specifying the SSID and Password of the Local WiFi Network
bool usingSoftAP = false;
const char* ssid = "AlexIphone"; //"your_wifi_ssid"
const char* password = "12345678"; //"your_wifi_password"
uint8_t retries=0;
const char* AP_SSID = "Roastology";

enum State {
    IDLE,
    GET_ALL_DATA,
    PROCESS_RESPONSE,
    SEND_COMMAND
};

State currentState = IDLE;
unsigned long previousMillis = 0;
unsigned long stateStartMillis = 0;
const long interval = 1000; // interval at which to get data (1 second)
const long timeout = 900; // timeout for GET_ALL_DATA state (0.9 second)
int responseLine = 0;
int target_temperature = 200;


void setup(){
```

```
  Serial.begin(9600);
  delay(100);
  Serial.println();

  //Try and Connect to the Network
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid,password);
  Serial.print("Connecting to ");
  Serial.print(ssid);
  Serial.println("...");

  //Wait for WiFi to connect for a maximum timeout of 20 seconds
  while(WiFi.status()!=WL_CONNECTED && retries<20)
  {
    Serial.print(".");
    retries++;
    delay(1000);
  }

  Serial.println();
  //Inform the user whether the timeout has occured, or the ESP8266 is
connected to the internet
  if(retries==20)//Timeout has occured
  {
    WiFi.mode(WIFI_AP);
    Serial.print("ACCESS_POINT: ");
    Serial.println(AP_SSID);
    Serial.println(WiFi.softAPIP());
    usingSoftAP = true;
  }

  if(WiFi.status()==WL_CONNECTED)//WiFi has succesfully Connected
  {
    Serial.print("IP_ADDRESS: ");
    Serial.println(WiFi.localIP());
  }

  const char* PARAM_MESSAGE = "message";

  // Serve the HTML page
```

```
  server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
      request->send(200, "text/html", R"rawliteral(
          <!DOCTYPE html>
          <html>
          <head>
              <title>Roastology Roaster Display/Controls</title>
              <script>
                  function updateRoast(message) {
                      const xhr = new XMLHttpRequest();
                      const url = "/roast";
                      xhr.open("POST", url, true);
                      xhr.setRequestHeader("Content-Type",
"application/x-www-form-urlencoded");
                      xhr.onreadystatechange = function () {
                          if (xhr.readyState === 4 && xhr.status === 200)
{
                              alert(xhr.responseText);
                          }
                      };
                      xhr.send("message=" + encodeURIComponent(message));
                  }

                  function updateTemperature() {
                      const xhr = new XMLHttpRequest();
                      xhr.open("GET", "/temperature", true);
                      xhr.onreadystatechange = function () {
                          if (xhr.readyState === 4 && xhr.status === 200)
{

document.getElementById("temperature").innerText = xhr.responseText;
                          }
                      };
                      xhr.send();
                  }

                  function updateTimeLeft() {
                      const xhr = new XMLHttpRequest();
                      xhr.open("GET", "/time_left", true);
                      xhr.onreadystatechange = function () {
```

```
                        if (xhr.readyState === 4 && xhr.status === 200)
{
                          const timeLeft = parseInt(xhr.responseText);
                          const minutes = Math.floor(timeLeft / 60);
                          const seconds = timeLeft % 60;
                          const formattedSeconds = seconds < 10 ? "0" +
seconds : seconds;
                          document.getElementById("time_left").innerText
= minutes + ":" + formattedSeconds;
                        }
                    };
                    xhr.send();
                }
                function toggleRoast() {
                  const xhr = new XMLHttpRequest();
                  xhr.open("POST", "/toggle_roast", true);
                  xhr.onreadystatechange = function () {
                    if (xhr.readyState === 4 && xhr.status === 200) {
                      alert(xhr.responseText);
                    }
                  };
                  xhr.send();
                }
                function restartRoast() {
                  const xhr = new XMLHttpRequest();
                  xhr.open("POST", "/restart_roast", true);
                  xhr.onreadystatechange = function () {
                    if (xhr.readyState === 4 && xhr.status === 200) {
                      alert(xhr.responseText);
                    }
                  };
                  xhr.send();
                }

                function updateRoastStatus() {
                  const xhr = new XMLHttpRequest();
                  xhr.open("GET", "/roast_status", true);
                  xhr.onreadystatechange = function () {
                    if (xhr.readyState === 4 && xhr.status === 200) {
```

```
                        document.getElementById("roast_status").innerText
= xhr.responseText;
                    }
                  };
                  xhr.send();
                }

                function updateRoastType() {
                  const xhr = new XMLHttpRequest();
                  xhr.open("GET", "/roast_type", true);
                  xhr.onreadystatechange = function () {
                    if (xhr.readyState === 4 && xhr.status === 200) {
                      document.getElementById("roast_type").innerText =
xhr.responseText;
                    }
                  };
                  xhr.send();
                }

                function updateTargetTemperature() {
                  const xhr = new XMLHttpRequest();
                  xhr.open("GET", "/target_temperature", true);
                  xhr.onreadystatechange = function () {
                    if (xhr.readyState === 4 && xhr.status === 200) {

document.getElementById("target_temperature").innerText =
xhr.responseText;
                    }
                  };
                  xhr.send();
                }

                function updateStatus() {
                    updateTemperature();
                    updateTimeLeft();
                    updateRoastStatus();
                    updateTargetTemperature();
                    updateRoastType();
                }
                setInterval(updateStatus, 1000);
```

```
            </script>
        </head>
        <body>
          <!-- Include Bootstrap CSS -->
          <link
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min
.css" rel="stylesheet">
          <!-- Include Bootstrap JS and dependencies -->
          <script
src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
          <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.2/dist/umd/popper.min
.js"></script>
          <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.j
s"></script>

          <div class="container">
              <h1 class="my-4">Roastology Roaster Display/Controls</h1>
              <div class="mb-3">Current Temperature: <span
id="temperature" class="badge badge-primary">Loading...</span>C</div>
              <div class="mb-3">Target Temperature: <span
id="target_temperature" class="badge
badge-primary">Loading...</span>C</div>
              <div class="mb-3">Current Roast Status: <span
id="roast_status" class="badge badge-secondary">Loading...</span></div>
              <div class="mb-3">Roast Type: <span id="roast_type"
class="badge badge-secondary">Loading...</span></div>
              <div class="mb-3">Time Left: <span id="time_left"
class="badge badge-info">Loading...</span></div>
              <div class="btn-group" role="group">
                  <button class="btn btn-light"
onclick="updateRoast('LIGHT')">Light Roast</button>
                  <button class="btn btn-warning"
onclick="updateRoast('MEDIUM')">Medium Roast</button>
                  <button class="btn btn-dark"
onclick="updateRoast('DARK')">Dark Roast</button>
              </div>
              <br><br>
```

```
                <div class="btn-group" role="group">
                    <button class="btn btn-success"
onclick="toggleRoast()">Start/Stop Roast</button>
                    <button class="btn btn-danger"
onclick="restartRoast()">Restart Roast</button>
                </div>
                <br><br>
                <h2>How to Use the Interface</h2>
                <p>Welcome to the Roastology Roaster Display/Controls
interface. Here's how you can use it:</p>
                <ul>
                  <li><strong>Current Temperature:</strong> Displays the
current temperature of the roaster.</li>
                  <li><strong>Target Temperature:</strong> Displays the
target temperature set for the roast.</li>
                  <li><strong>Current Roast Status:</strong> Shows the
current roast type and whether the roast is on or off.</li>
                  <li><strong>Time Left:</strong> Displays the remaining
time for the roast in minutes and seconds.</li>
                  <li><strong>Light Roast:</strong> Click this button to
set the roast type to Light.</li>
                  <li><strong>Medium Roast:</strong> Click this button to
set the roast type to Medium.</li>
                  <li><strong>Dark Roast:</strong> Click this button to
set the roast type to Dark.</li>
                  <li><strong>Start/Stop Roast:</strong> Click this button
to toggle the roast on or off.</li>
                  <li><strong>Restart Roast:</strong> Click this button to
restart the roast process.</li>
                </ul>
                <p>Roast type can only be changed when the roast is off.
Time left and target temperature are set by the roaster.</p>
            </div>
          </body>
          </html>
      )rawliteral");
  });

  // Handle POST request to update roast type
```

```
  server.on("/roast", HTTP_POST, [PARAM_MESSAGE](AsyncWebServerRequest
*request){
      String message;
      if (request->hasParam(PARAM_MESSAGE, true)) {
          message = request->getParam(PARAM_MESSAGE, true)->value();
          roast_type = message; // Update roast type
          commandQueue.push("UPDATE_ROAST_TYPE_" + message);
      } else {
          message = "No message sent";
      }
      request->send(200, "text/plain", "Roast type updated to: " +
message);
  });
  // Handle Post Request for toggling roast
  server.on("/toggle_roast", HTTP_POST, [](AsyncWebServerRequest
*request){
    roast_on = !roast_on;
    commandQueue.push(roast_on ? "TURN_ROAST_ON" : "TURN_ROAST_OFF");
    request->send(200, "text/plain", "Roast toggled to: " +
String(roast_on ? "On" : "Off"));
  });
  // Handle Post Request for restarting roast
  server.on("/restart_roast", HTTP_POST, [](AsyncWebServerRequest
*request){
    roast_on = false;
    time_left = 600;
    commandQueue.push("RESTART_ROAST");
    request->send(200, "text/plain", "Roast restarted");
  });

  // Handle temperature request
  server.on("/temperature", HTTP_GET, [](AsyncWebServerRequest *request){
      request->send(200, "text/plain", String(temperature));
  });

  // Handle time left request
  server.on("/time_left", HTTP_GET, [](AsyncWebServerRequest *request){
      request->send(200, "text/plain", String(time_left));
  });
  // Handle roast status request
```

```
  server.on("/roast_status", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(200, "text/plain", "Roasting:" + String(roast_on ? "On"
: "Off"));
  });

  //handle roast type request
  server.on("/roast_type", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(200, "text/plain", String(roast_type) + "Roast");
  });

  //Hanldle target temperature request
  server.on("/target_temperature", HTTP_GET, [](AsyncWebServerRequest
*request){
    request->send(200, "text/plain", String(target_temperature));
  });

  server.onNotFound([](AsyncWebServerRequest *request){
    request->send(404, "text/plain", "Not found");
  });

  server.begin();
  Serial.println("HTTP server started");
}


void loop() {
  unsigned long currentMillis = millis();

  // Check if ESP8266 is connected to WiFi network or using SoftAP
  if (WiFi.status()==WL_CONNECTED || usingSoftAP)
  {
    //EP8266 is connected to WiFi Access Point or using SoftAP, you can
proceed with your code execution
    switch (currentState) {
        case IDLE:
            // Check if it's time to get data
            if (currentMillis - previousMillis >= interval) {
                previousMillis = currentMillis;
                Serial.println("GET_ALL_DATA");
                currentState = GET_ALL_DATA;
```

```
            stateStartMillis = currentMillis;
            responseLine = 0;
        }
        // Check if there are any commands to send
        else if (!commandQueue.empty()) {
            String command = commandQueue.front();
            commandQueue.pop();
            Serial.println(command);
            currentState = SEND_COMMAND;
        }
        break;

    case GET_ALL_DATA:
        if (currentMillis - stateStartMillis >= timeout) {
            // Timeout exceeded, go back to IDLE
            currentState = IDLE;
            break;
        }
        while (Serial.available()) {
            String response = Serial.readStringUntil('\n'); // Read
the response until newline
            response.trim(); // Remove any leading/trailing whitespace
            switch (responseLine) {
                case 0:
                    temperature = response.toFloat();
                    break;
                case 1:
                    time_left = response.toInt();
                    break;
                case 2:
                    roast_type = response;
                    break;
                case 3:
                    roast_on = (response == "true");
                    break;
                case 4:
                    target_temperature = response.toInt();
                    currentState = PROCESS_RESPONSE;
                    break;
            }
```

```
                responseLine++;
            }
            break;

        case PROCESS_RESPONSE:
            // Process the response here if needed
            // For now, just go back to IDLE
            currentState = IDLE;
            break;

        case SEND_COMMAND:
            // After sending the command, go back to IDLE
            currentState = IDLE;
            break;

        default:
            currentState = IDLE;
            break;
    }
  }
  else
  {
    //ESP8266 is not connected to any WiFi network. You need to wait for
the internet connection before you start interacting with any web server
    Serial.print("Trying to connect with ");
    Serial.print(ssid);
    while(WiFi.status()!=WL_CONNECTED)
    {
      Serial.print(".");
      delay(1000);
    }
    Serial.println();
    Serial.print("Sucessfully Connected to ");
    Serial.println(ssid);
    Serial.print("IP Address: ");
    Serial.println(WiFi.localIP());
  }
  //   // Other code that runs continuously
  // long new_interval = 5000;
  // if (currentMillis - previousMillis >= new_interval) {
```

```
  //   previousMillis = currentMillis;
  //   //flip between roast types
  //   if (roast_type == "LIGHT") {
  //     roast_type = "MEDIUM";
  //   } else if (roast_type == "MEDIUM") {
  //     roast_type = "DARK";
  //   } else {
  //     roast_type = "LIGHT";
  //   }
  // }
}
```