Purdue University Fort Wayne

Mechanical Engineering Program

(ME 487 – ME 488)

Capstone Senior Design Project

Report #2

Project Title:	**Extruder for Production of 3-D Printing Filament**
Team Members:	**Mason Averill**
	Kade Bontrager
	Chandler Schoeff
Faculty Advisor:	**Dr. Donald Mueller**
Date:	**December 3rd, 2021**

Table of Contents

Acknowledgements *i*

Abstract *ii*

Section I: Building Process *1*

Section 1.1: Initial Changes to Preliminary Design *2*

Section 1.2: Refurbishing the Cart *2*

Section 1.3: 3-D Printed Parts *3*

Section 1.4: Material Input Subsystem *9*

Section 1.5: Extrusion Subsystem Building Process *10*

Section 1.6: Heating Subsystem Building Process *12*

Section 1.7: Cooling Subsystem Building Process *12*

Section 1.8 Spooling Subsystem Building Process *13*

Section 1.9: Wiring and Electrical Components *15*

Section 1.10: Temperature Control Arduino (1) Functions and Code Logic *34*

Section 1.11: Motor Control Arduino (2) Functions and Code Logic *36*

Section II: Testing and Evaluation *37*

Section 2.1: Testing Plan *38*

Section 2.2: Testing Results *38*

Section III: Cost Analysis *46*

Conclusion *50*

References *51*

Appendices *52*

Acknowledgements

The design team would like to extend our gratitude to the Purdue Fort Wayne Civil and Mechanical Engineering Department for providing the project and financial investment necessary for the completion of this project.

We would like to thank Jason Moyer for the technical assistance that was provided throughout the construction of the device.

Finally, we would like to thank Dr. Donald Mueller for acting as our advisor for the project, and all of the assistance throughout the design and development process.

Abstract

The Civil and Mechanical Engineering Department at Purdue University Fort Wayne has requested a device that is able to convert failed 3-D prints into reusable filament for the 3-D printing lab. The purpose of this document is to communicate the building process, testing, and cost analysis of this machine.

The first section discusses the construction of each of the subsystems, both mechanically and electrically. This begins with the modifications that were made to the sheet metal cart that was provided and moves into the mechanical construction of each of the subsystems in the following order: material input, extruding, heating, cooling, then finally spooling. Next, a detailed explanation of the extensive amount of electrical components and wiring is outlined before finishing up with an outline of the Arduino codes that were used to control the motor functions and heating functions.

The second section details the results of testing the device to comply with the three requirements that were established previously. The nozzle temperature requirements were tested simultaneously and were verified using a thermal imaging camera as well as thermocouple capabilities of a multimeter. The 1.75 mm filament diameter requirement was tested by extruding filament and measuring the diameter with calipers. Both of the temperature requirements were met, as the nozzle temperature was adjustable and could be heated to above 300 °C. However, the filament diameter requirement wasn't fully achieved—the machine is capable of producing filament at a diameter of 1.75 mm, but the correct combination of device settings has not been discovered to maintain a tolerance of ± 0.05 mm. The final section is a detailed cost analysis of the cart, including the prices and quantities of each item that was purchased.

The appendix contains the Arduino codes that are used to control the functions of the device.

Section I: Building Process

Section 1.1: Initial Changes to Preliminary Design

Originally, the device was designed to be mounted to a piece of aluminum sheet metal, but before beginning the construction process, we were approached about using an old caster wheel cart that no longer had a use in the department. We decided to use this, as it greatly improved the transportability of the device, and allowed for everything to be mounted to one transportable piece of equipment. Since the cart was used, the mounting designs from the preliminary design were also changed to be compatible with the cart.

Another change from the original preliminary design was the removal of the lateral guide and drawing systems within the spooling subsystem. Researching the topic of spooling allowed us to determine that these were not a complete necessity to spool the filament properly.

Section 1.2: Refurbishing the Cart

To start assembly of the extruder, the cart needed to be extended to fit the entire device. The cart was extended by fastening 80/20 aluminum supports to its sides. The cart after adding the extension is shown in Figure *1*.

Figure 1: Cart After the 80/20 Extension was Added

Once the cart was extended, it was spray painted to make it more aesthetically pleasing and to provide protection against future corrosion. Figure 2 shows the cart after the sheet metal was fastened to the extended frame and the cart was painted.

Figure 2: Cart After the Addition of Sheet Metal and Spray Paint

Section 1.3: 3-D Printed Parts

Many parts were 3-D printed for easy customization. Using 3-D printed parts was avoided for the extrusion and heating sections of the device because of the high temperatures they reach, but multiple parts were designed and printed for the cooling subsystems.

The largest 3-D printed part was the duct for the cooling section. The duct was split into two identical prints and fastened together by bolting the fans to each of the halves. Figure *3* and Figure *4* display each of the pieces of the duct.

Figure 3: One Half of PLA Duct

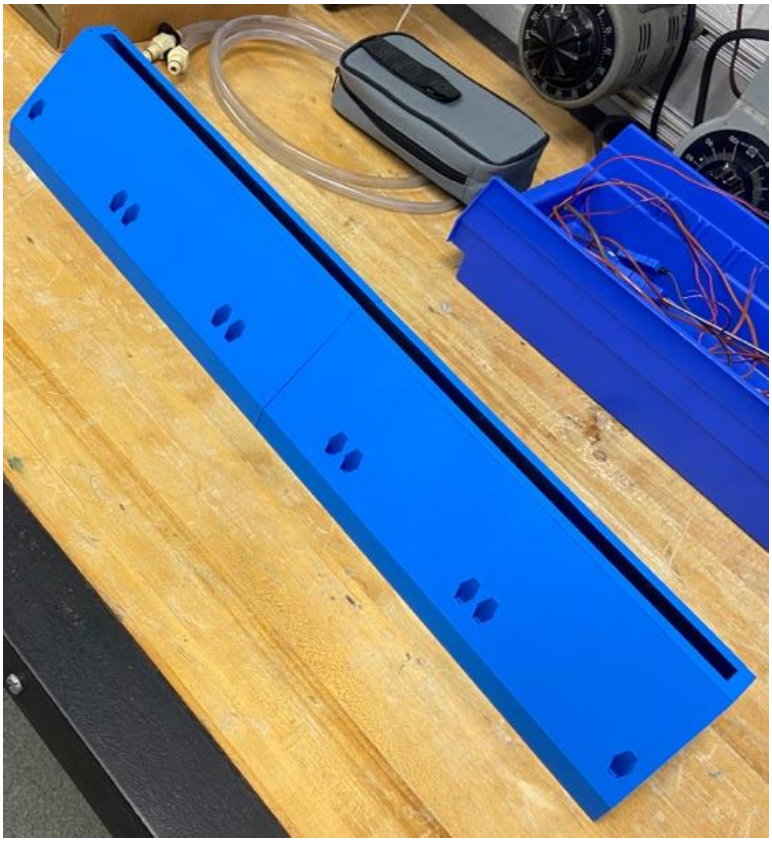

Figure 4: Full PLA Duct

For the spooling section, the spool holder was 3-D printed using the university's Stratasys machine with ABS filament. Figure *5* and Figure *6* shows a couple views of the spool holder attachment.

Figure 5: Spool Holder

Figure 6: Top View of Spool Holder

As shown above in Figure *6*, there is a slot for a nut to slide in, and a hole from the exterior of the shaft. A ¼-20" bolt is able to thread through the nut and tighten onto the shaft, acting as a set screw to fix the rotation of the spool holder with the shaft.

Figure 7: Spool Holder with Nut Attachment

The motor mount for the spooling motor was also 3-D printed, shown in Figure *8* and Figure *9*. The spooling motor was one being repurposed from the university stock, and a custom mount was designed in SolidWorks. More detail on the spooling pieces and their functions will follow in the spooling build process subsection.

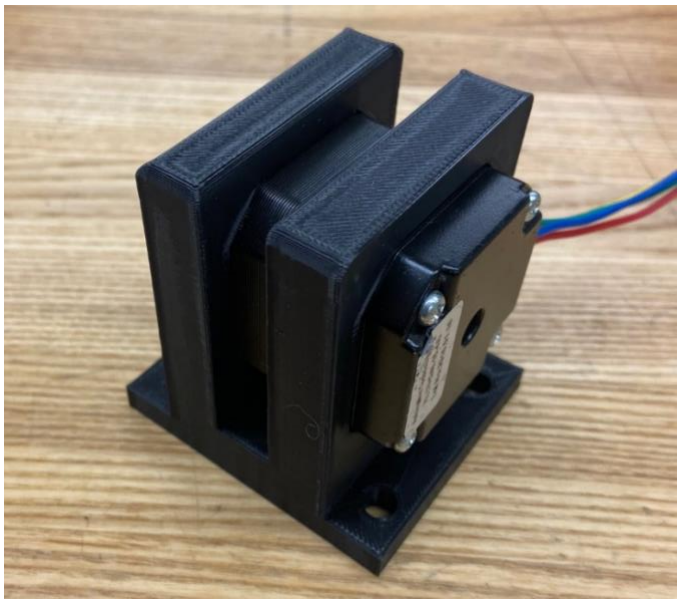

Figure 8: Back View of Spooling Motor Mount

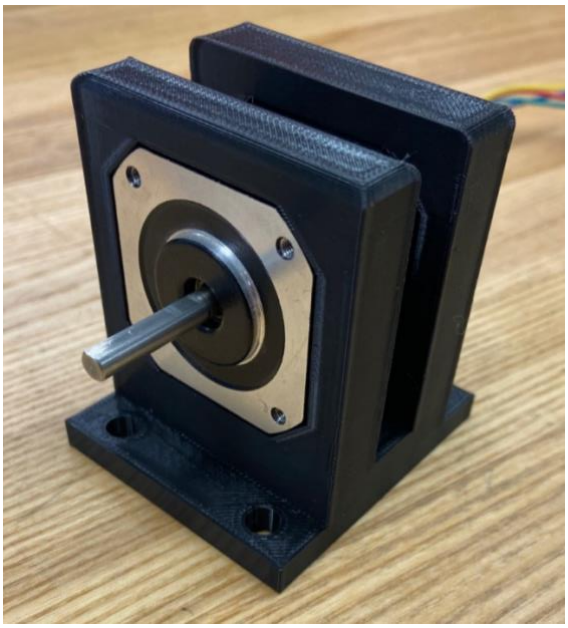

Figure 9: Front View of Spooling Motor Mount

Next, a mount was designed in SolidWorks for the LCD screen in order to fasten it to the cart and allow for exposed wires to be protected. The final mount is shown below in Figure *10*.

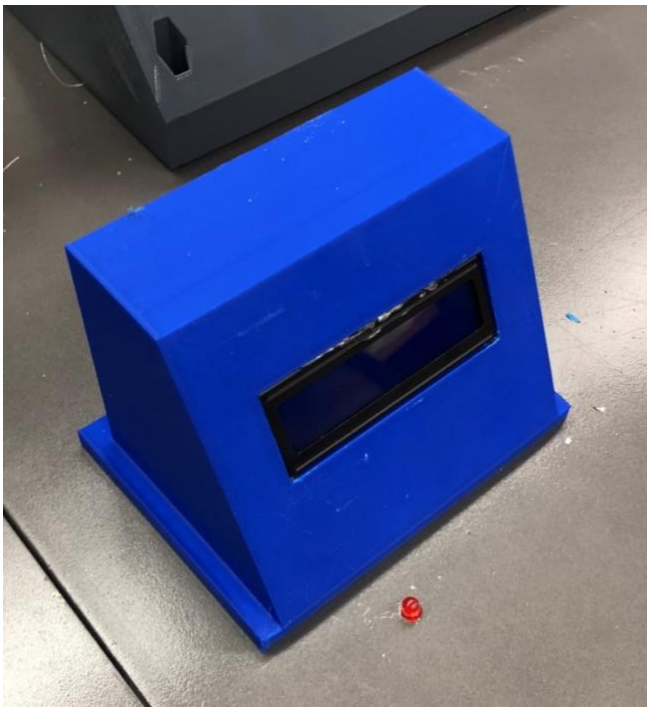

Figure 10: LCD Screen Mount

Finally, the part used to mount the break beam sensors was 3-D printed, shown in Figure *11*.

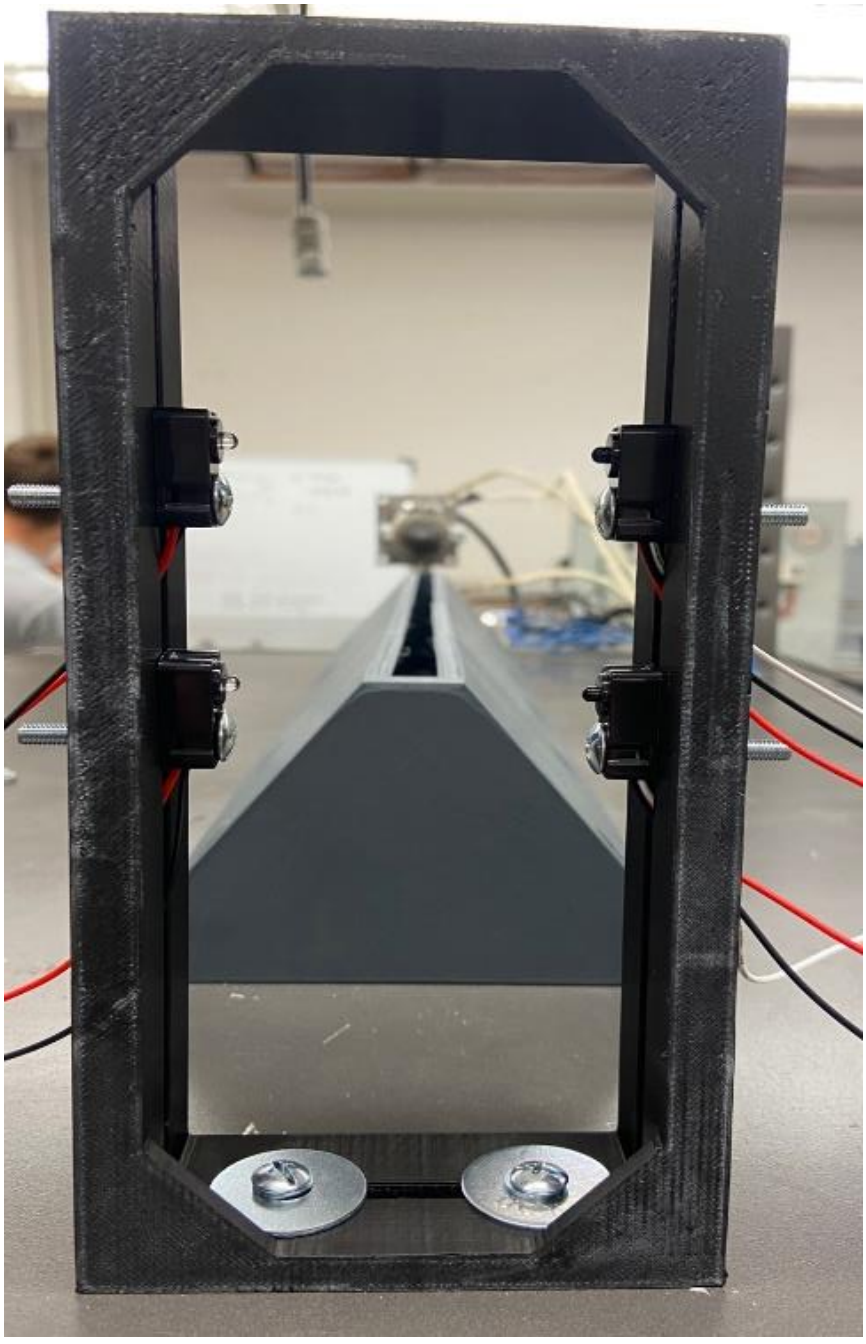

Figure 11: Break Beam Sensor Mount

Section 1.4: Material Input Subsystem

The adapter to connect the hopper to the extruder barrel was made using a CNC mill. The part was designed in SolidWorks, drawings were prepared, and the CNC code was generated for the machine. Figure *12* shows the part being milled.

Figure 12: Machining of the Hopper Adapter

The flats for the hopper were cut from sheet metal using a laser cutter, as shown in *Figure 13.*

Figure 13: Hopper Flats Being Cut from Sheet Metal

After the hopper and hopper connecter were finished, they were welded together as needed to fill the open seams. Figure *14* displays the complete hopper assembly after it had been welded and powder coated.

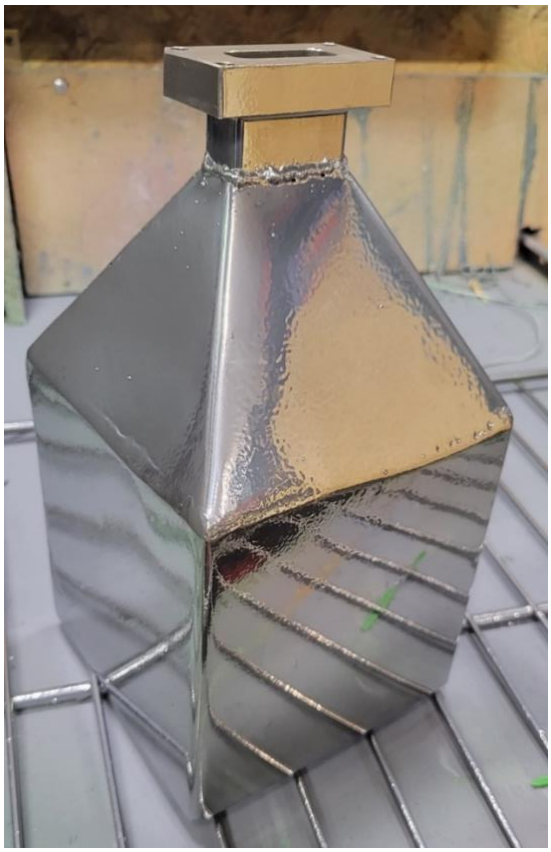

Figure 14: Hopper Assembly

Section 1.5: Extrusion Subsystem Building Process

First, the extruder barrel, motor, auger screw, and nozzle were all ordered and delivered. Holes were drilled through the top surface of the sheet metal extension and the extruder barrel was mounted using clamps that are attached to threaded rods that run through the holes. Thermal insulation padding was wedged between the clamps and the barrel to ensure that the large amount of heat from the heaters is not transferred to the cart itself. The threaded rods are secured by tightening a nut on the top and bottom sides of the cart's surface. Each rod runs through the cart and its corresponding flange, and the flanges are tightly secured to the sheet metal with hex bolts. The height of the barrel is controlled by the adjusting the nuts on the threaded rods. The same fixing method was done for the auger motor. Figure *15* and Figure *16* show the barrel and extruder motor mounted to the cart.

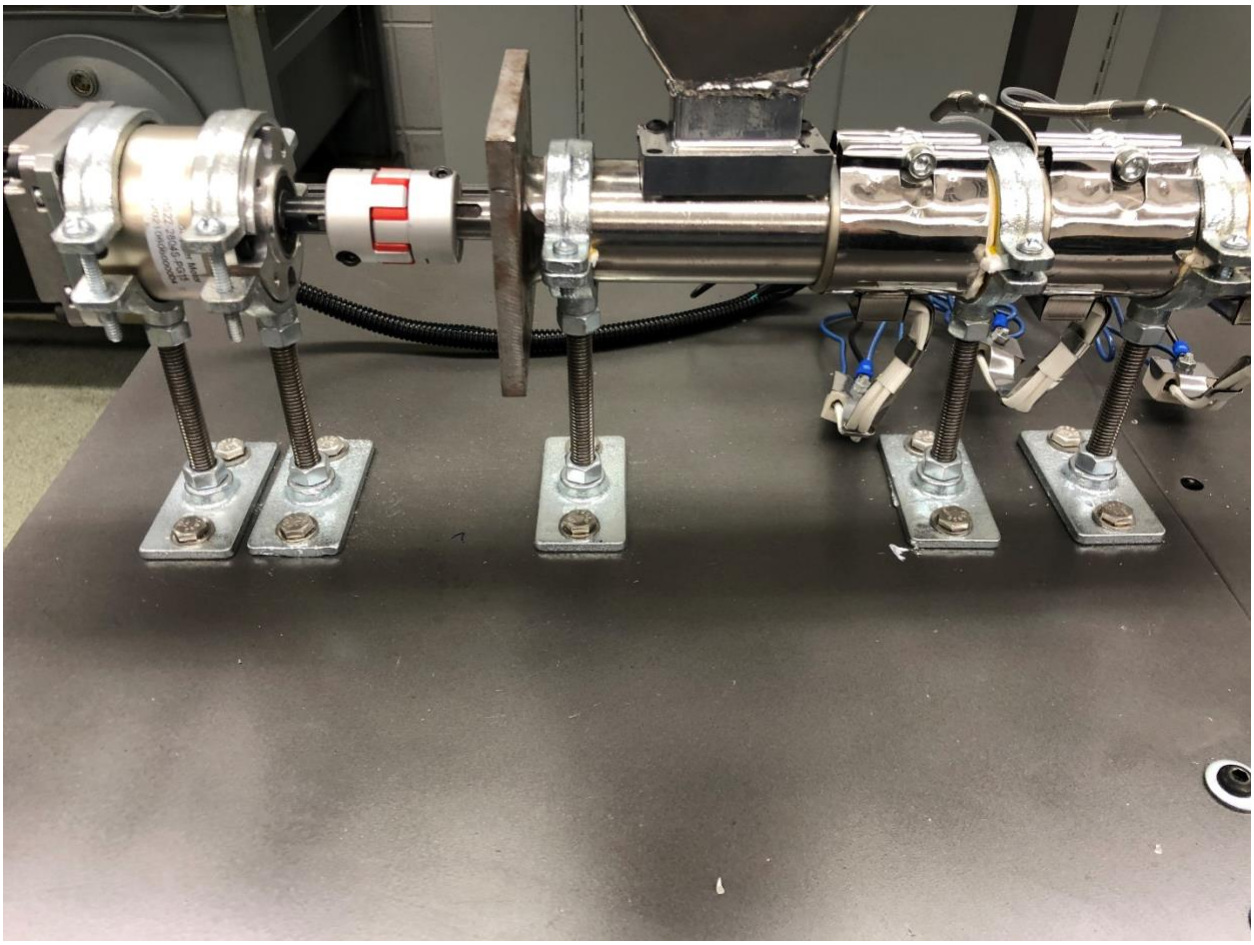

Figure 15: Top View Detailing the Mounting of the Extrusion Subsystem

Figure 16: Bottom View of the Extrusion Mounting Method

Section 1.6: Heating Subsystem Building Process

The heating subsystem acts in unison with the extrusion subsystem, and the building of this was no different. Once the band heaters were ordered and delivered, they were simply fitted onto the barrel of the extruder. Space was left between each band heater for the mounting clamps. Two thermocouples were also fitted between the clamps and the barrel to allow control over the temperature of the barrel. The third and final thermocouple was attached to the nozzle using a hose clamp. The heating setup is shown below in Figure 17.

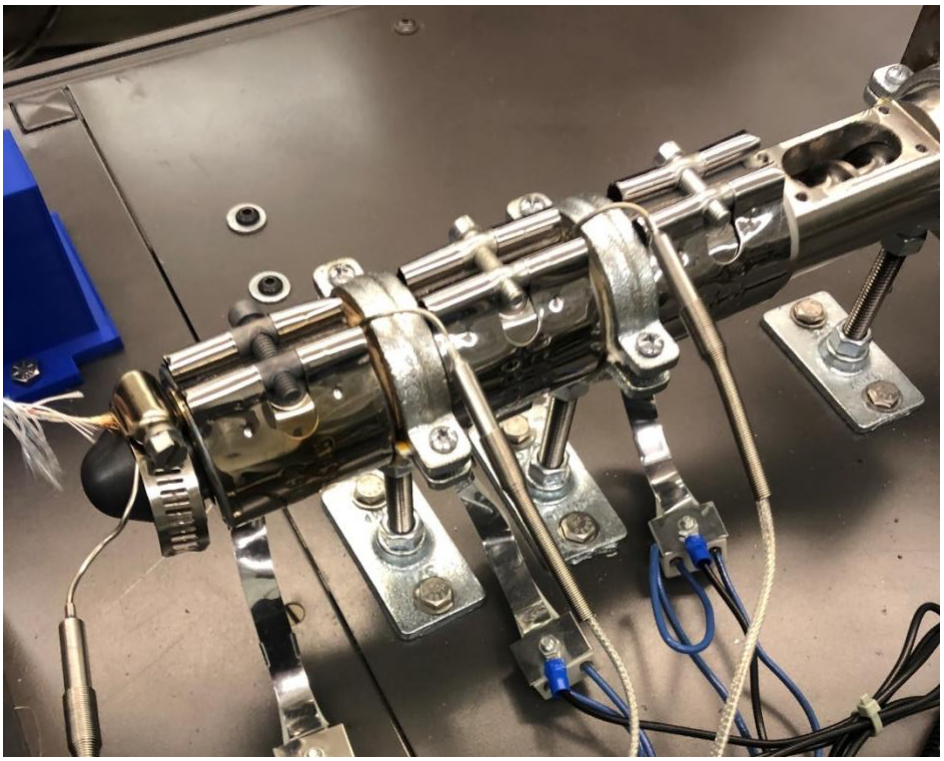

Figure 17: Close-up View of Thermocouples and Band Heaters on the Barrel

Section 1.7: Cooling Subsystem Building Process

The 3-D printed duct was designed so that it could be directly attached to the cooling fans through the use of M4 shoulder bolts and locknuts. This is shown below in Figure 19. The duct with the attached fans were slotted into a cut of the rectangular profile matching the lined-up fans in the cart. Doing so allowed for the fans to be hidden under the cart and for the duct to sit flush with the cart, all while allowing full airflow from the fans. The filament guides were inserted into drilled holes along the top of the duct and created out of aluminum wire molded by hand into the guiding shape. These, as well as the top view of the mechanical aspects of the cooling subsystem are shown below in Figure 18.

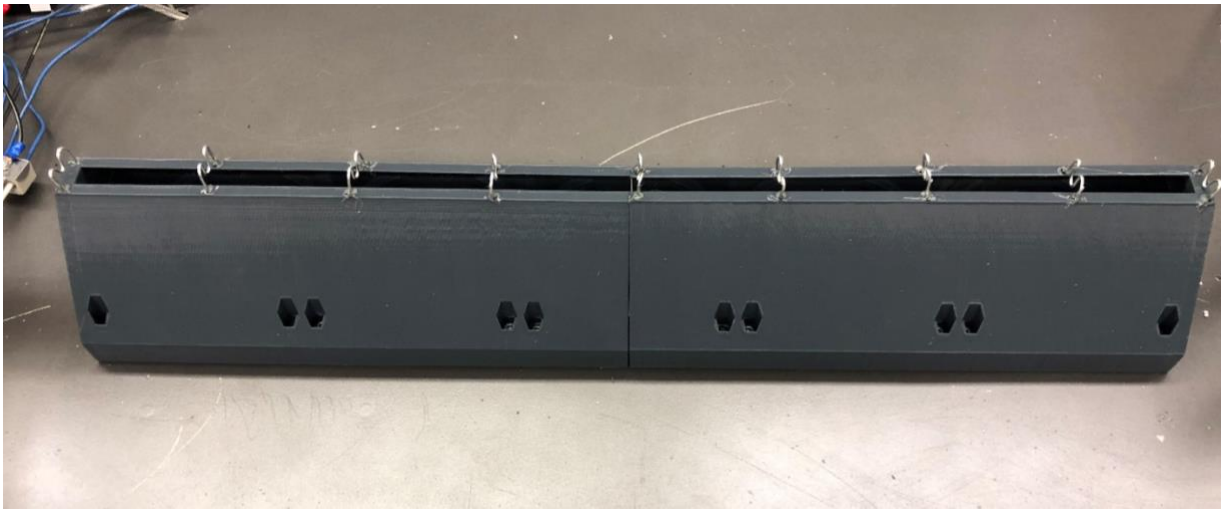

Figure 18: Top View of Duct Inserted in the Cart

Figure 19: Bottom View of Duct and Fans Inserted in the Cart

Section 1.8 Spooling Subsystem Building Process

In order to neatly spool the material without compromising the height of the duct and barrel, another extension was made to the cart using a piece of sheet metal, hex bolts, and L-brackets. The 3-D printed motor mount from Figure *8* was then bolted to the extension along with a bearing mount. A custom-made shaft coupler was attached to the motor on one end, and the 5/16” shaft on the other end. The spool holder was then fit onto the shaft. The electrical

configuration and integration of this can be viewed in Figure *23*. Figure *20* and Figure *21* display the mechanical components of the spooling subsystem.

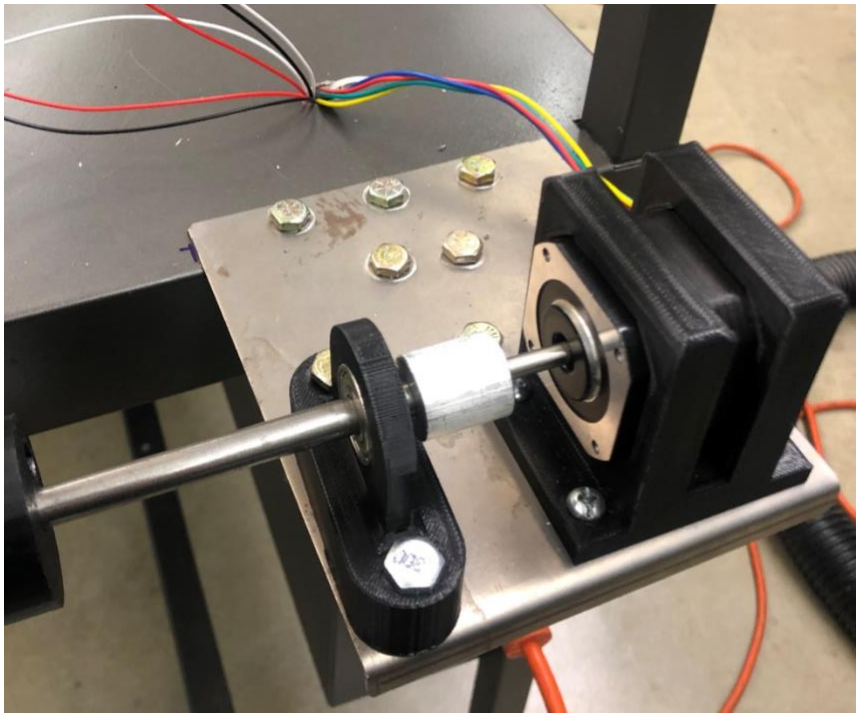

Figure 20: Spooling Motor and Bearing Mounted to the Sheet Metal Extension

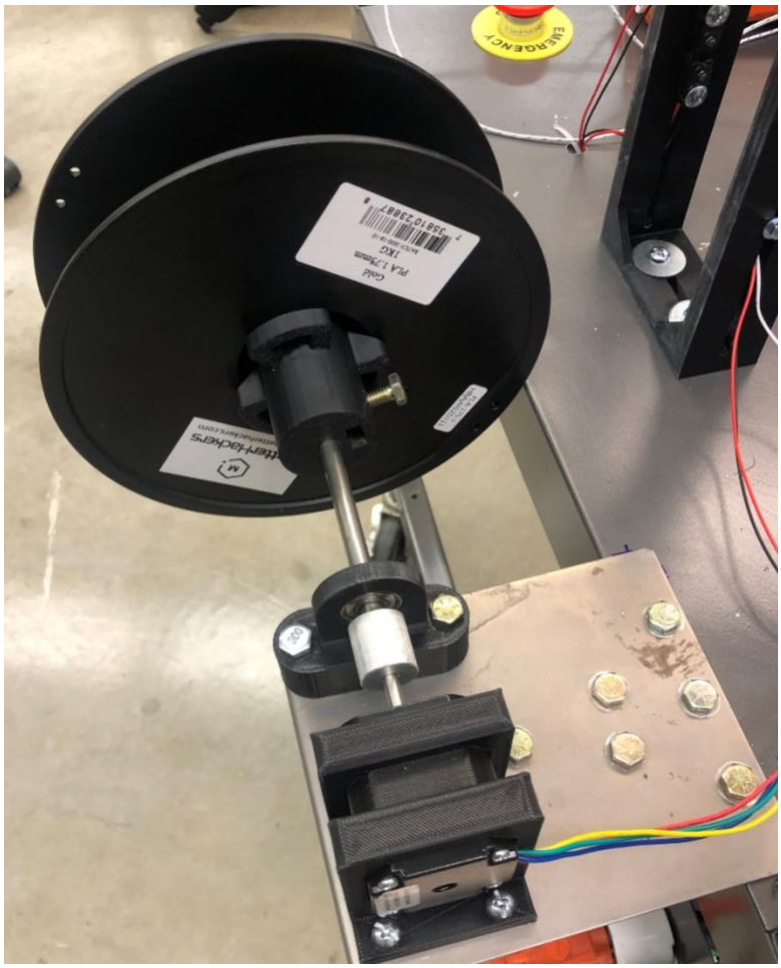

Figure 21: Full Mechanical Components of the Spooling Subsystem

Section 1.9: Wiring and Electrical Components

This project involved a great deal of wiring, all of which was performed without assistance from anyone outside the group. The components that required wiring were as follows:

- 1 Nema-23 stepper motor to drive the auger screw
- 1 Nema-17 stepper motor to drive the spool holder
- 1 stepper motor driver for each of the stepper motors (2x total)
- 5 120mm 120V AC fans for the cooling section
- 3 ungrounded Type K thermocouples
- 3 thermocouple amplifying boards (MAX31855)
- 2 Arduino Mega 2560’s (one for driving the stepper motors, one for nearly everything else)
- 4 5K linear potentiometers
- 1 rocker switch

- 3 latching push buttons
- 1 emergency stop button
- 1 LCD display
- 3 250W band heaters
- 1 4 channel 5V relay module
- 1 12V AC-DC power supply
- 1 0-48V AC-DC power supply
- 1 9V AC-DC power supply for the Arduino used with the MAX31855 boards (very low electrical noise required, output voltage cannot fluctuate significantly)

Figure *22* displays all of these listed components.

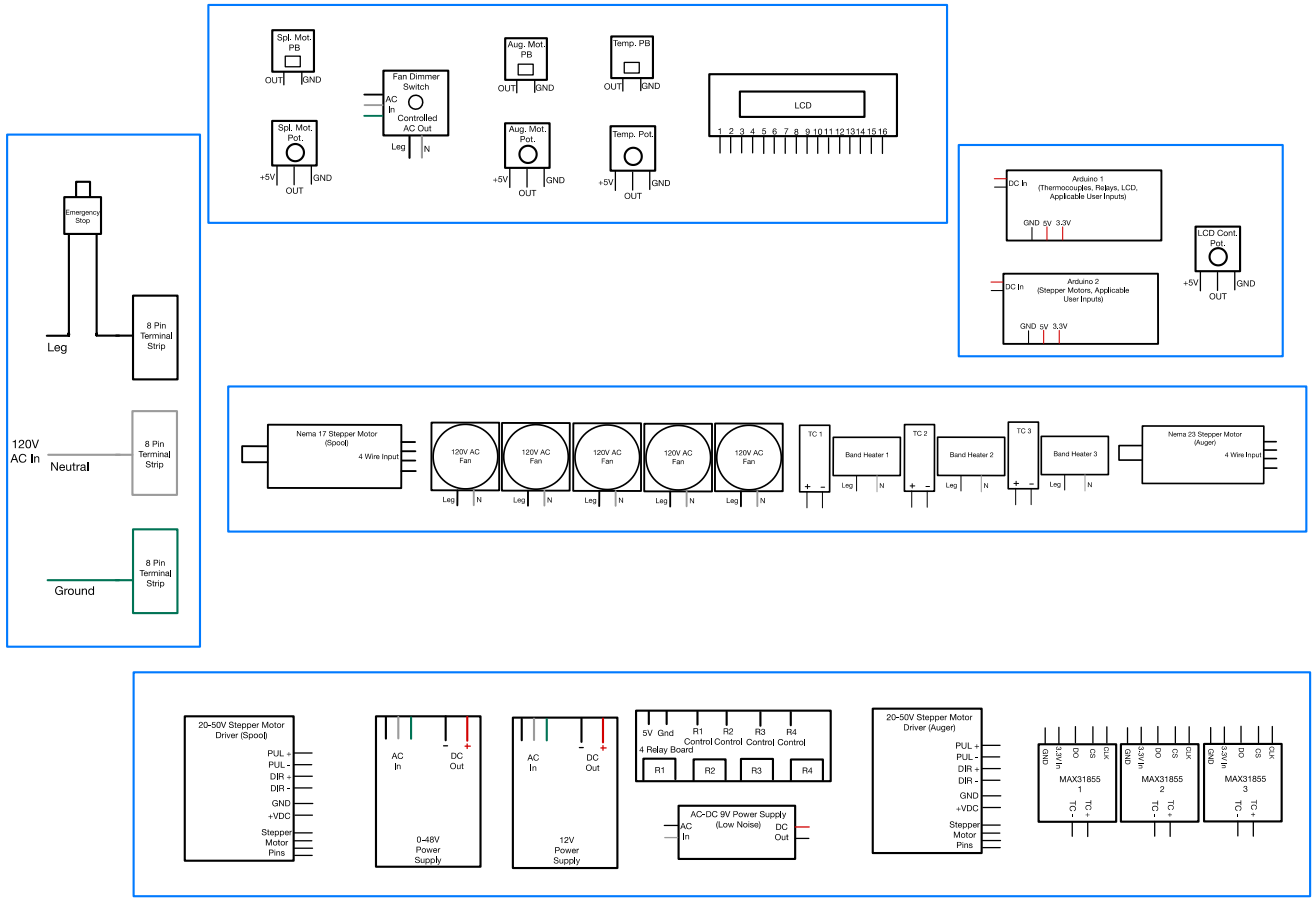

Figure 22: Electrical Components

Figure *23* shows the wiring diagram for all power supplies, stepper motor drivers, stepper motors, thermocouples, thermocouple amplifying boards, Arduinos, band heaters, and the 4 channel 5V relay module.

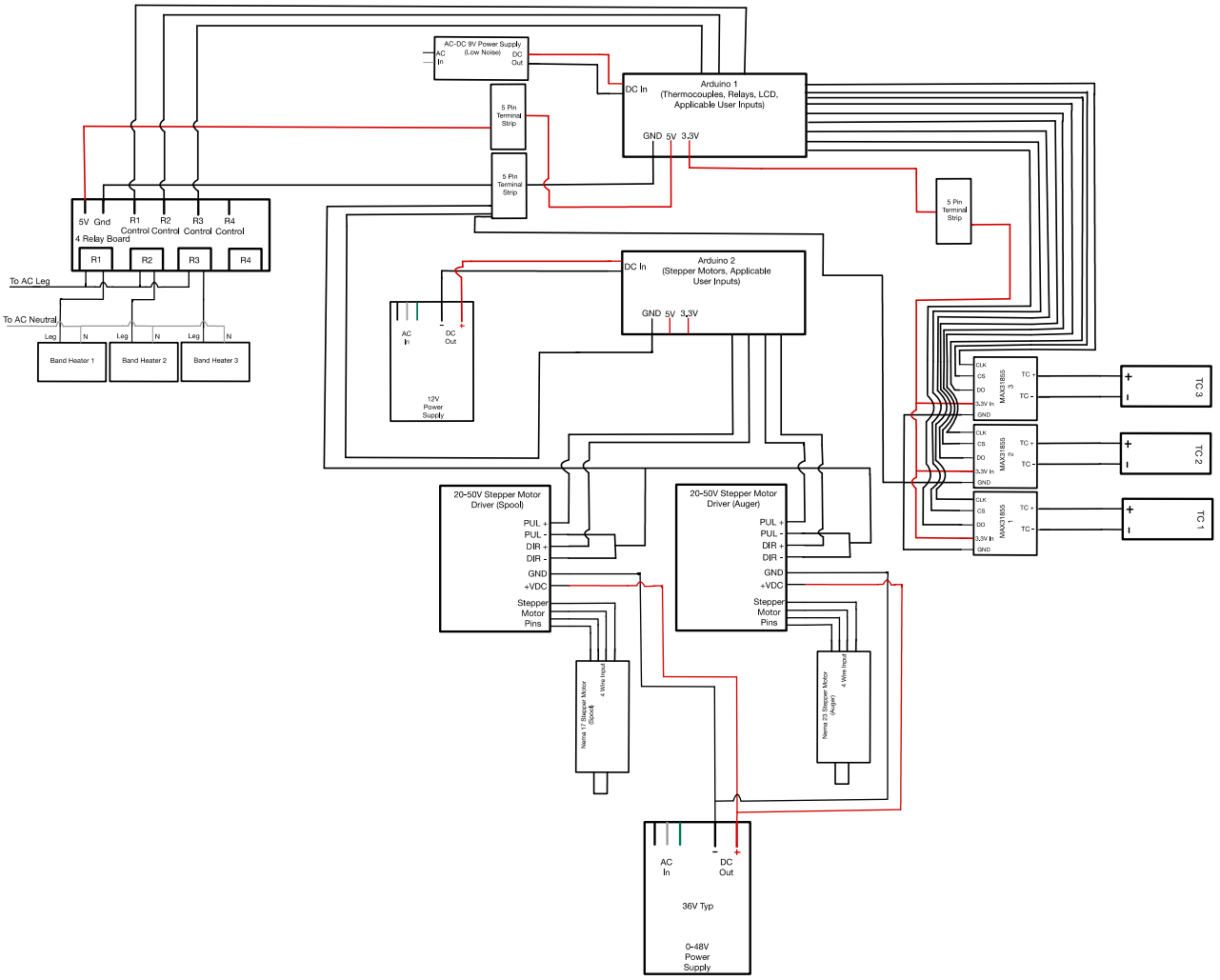

Figure 23: Partial Electrical Wiring Diagram

Figure *24*-Figure *31* displays the physical wiring completed and discusses pertinent connections shown in each Figure.

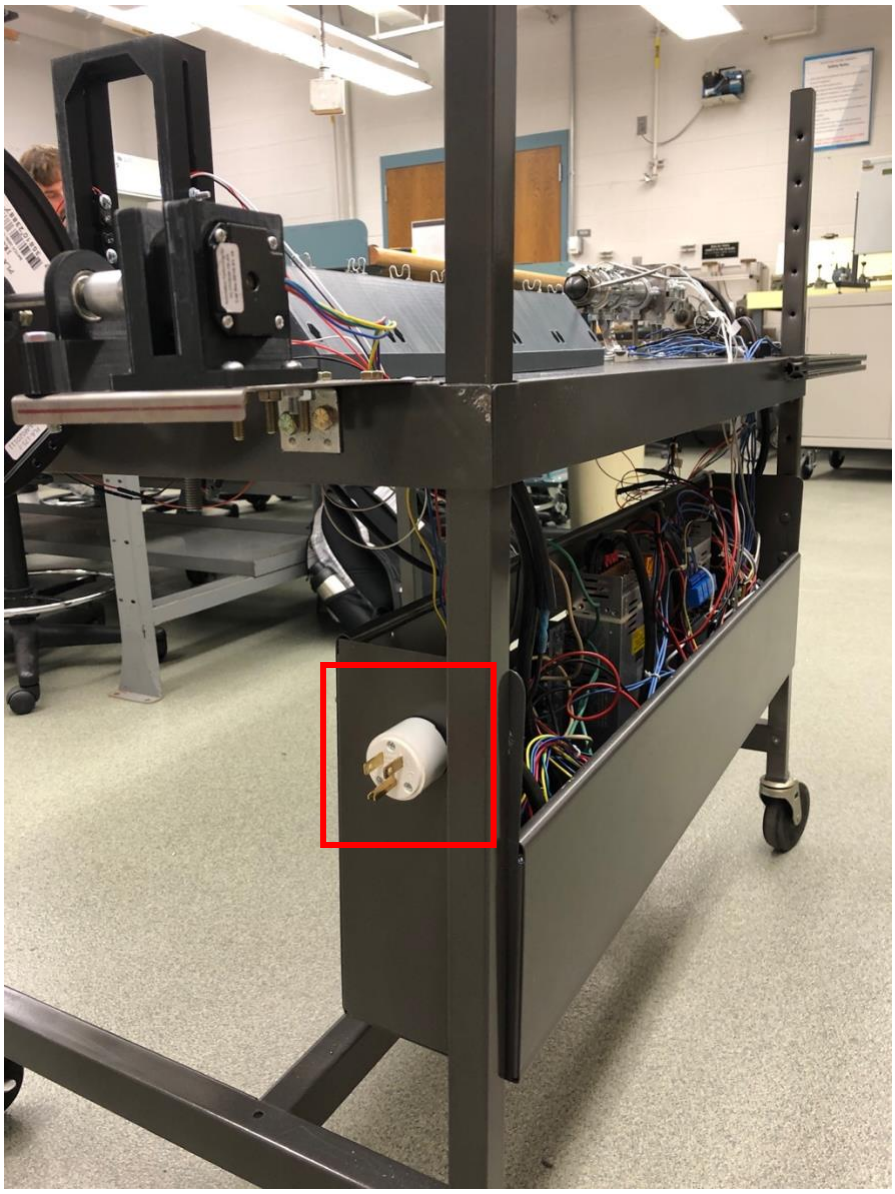

Figure 24: Power Entry

Power input into the cart is very simple. An extension cord is plugged into any typical 120V receptacle nearby and the opposite end directly plugs into the connection shown in Figure *24*.

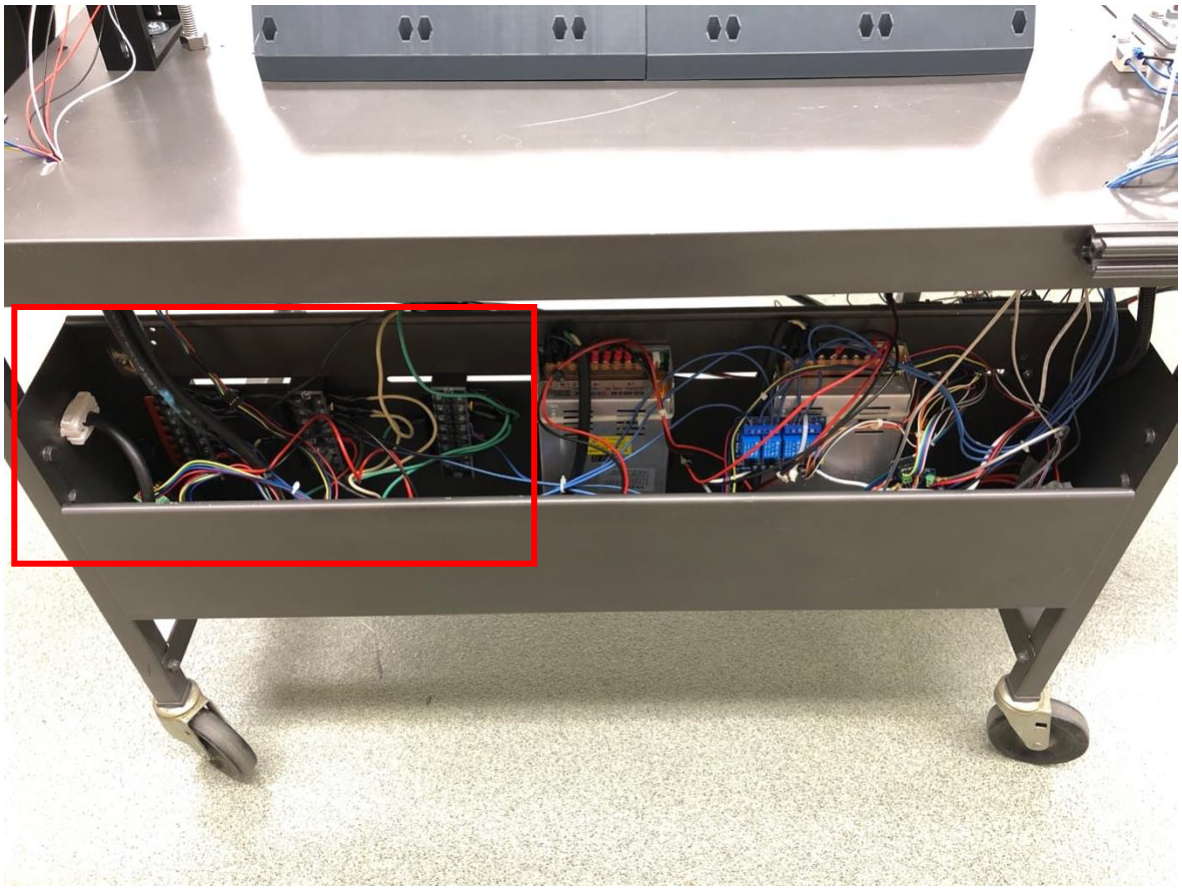

Figure 25: Power Entry Distribution

After 120V AC power enters the rear tray of the cart, it is split off into its 3 components: leg (or hot), neutral, and ground. The ‘hot’ wire immediately runs to a normally closed emergency stop button at the front of the cart, shown by Figure *26*, and then back to the entrance of the tray. Each component (leg, neutral, and ground) then runs to its own 8-position terminal strip shown in Figure *25*.

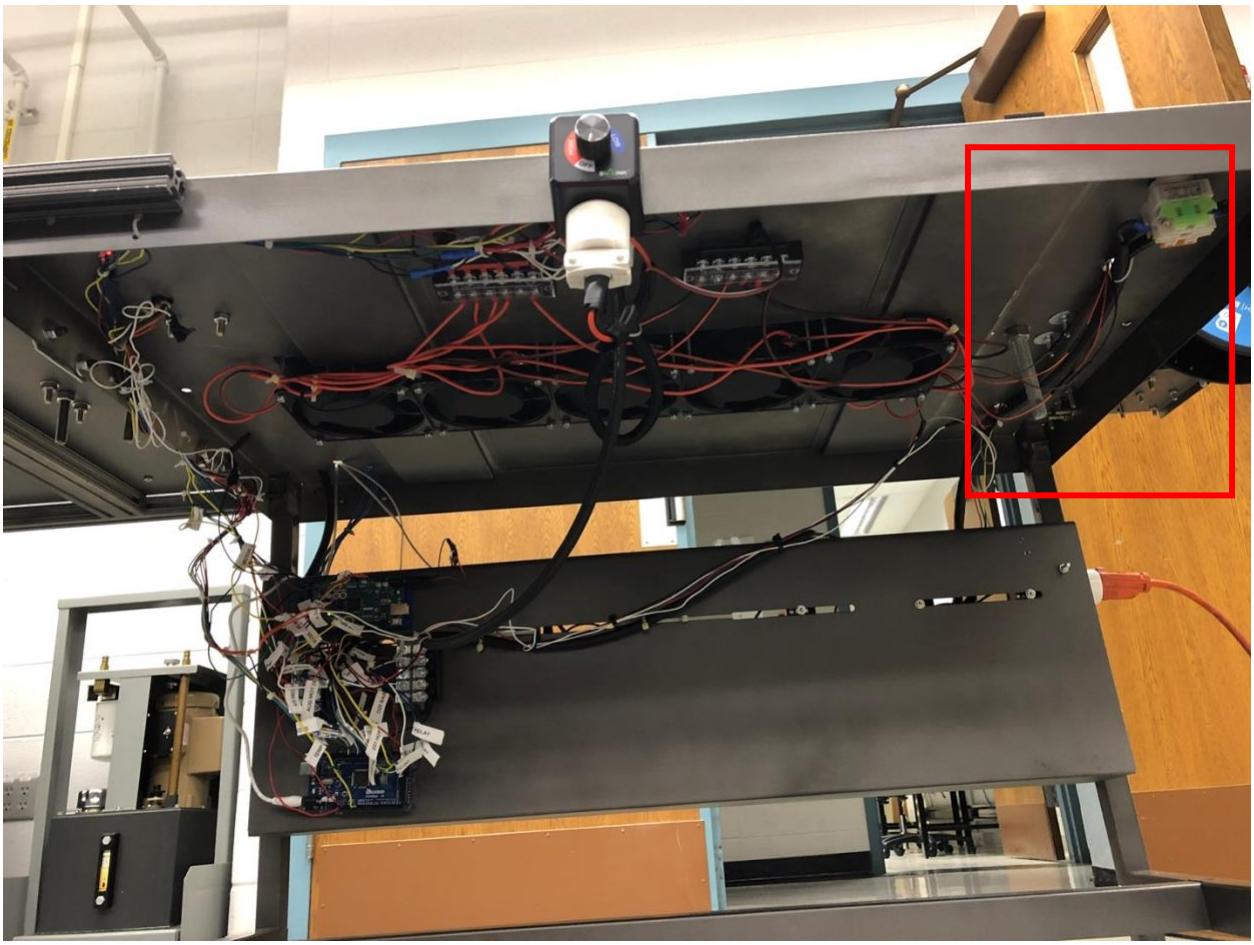

Figure 26: Emergency Stop Button Wiring

Next, the 3 8-position terminal strips are used to power all AC devices. The first item that will be discussed is the fan dimmer switch and associated fan wiring, shown by Figure *27*.

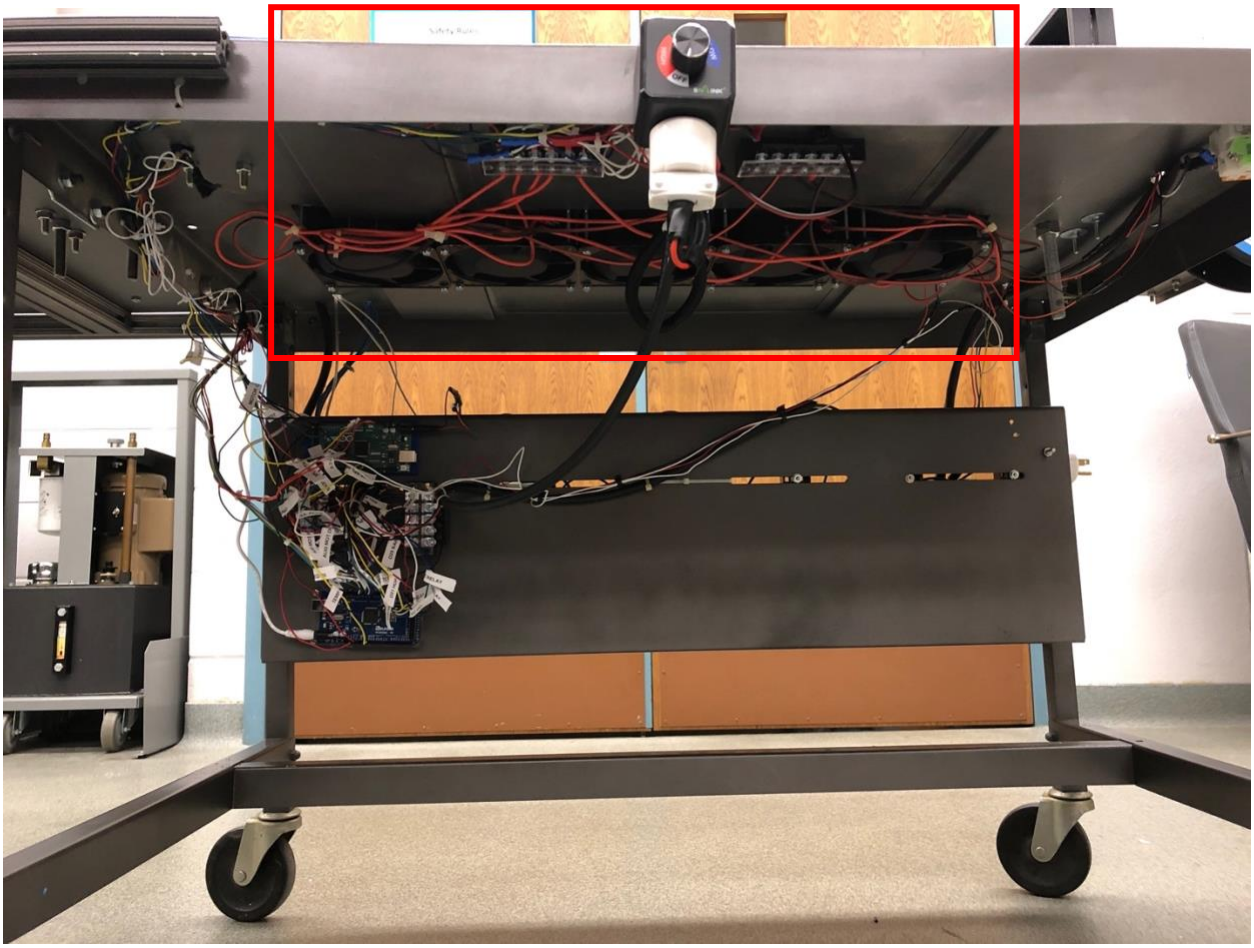

Figure 27: Fan Wiring

The 8-position terminal strips were used to run one wire from each component of the input power to the input of the fan dimmer switch. The output of the fan dimmer switch then splits into its ‘hot’ and neutral components and are ran to 2 6-position terminal strips, which can be seen in Figure *27*. Each fan then had its own ‘hot’ and neutral wire ran to it from the 6-position terminal strips.

The remainder of the AC wiring can be seen in Figure *28*.

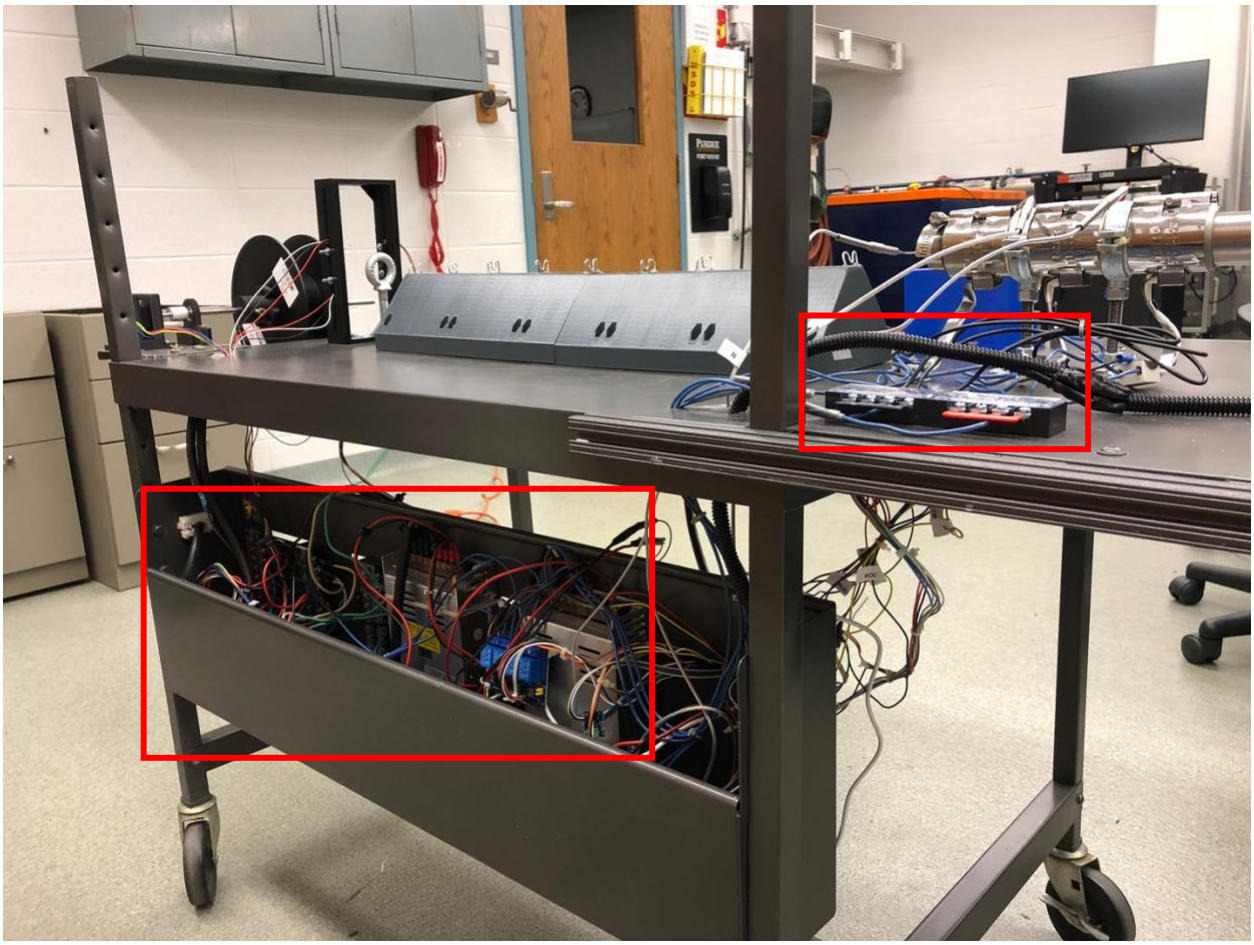

Figure 28: Remaining AC Wiring

The 8-position terminal strips were used to deliver input power into all 3 AC-DC power supplies. In addition, 3 ‘hot’ wires were ran from the ‘hot’ 8-position terminal strip into 3 separate relays on the 4 relay module board. The output from these relays were then wired to the ‘hot’ input on each of the 3 band heaters. Next, wiring was ran from each of the neutral and ground 8-position terminal strips to 2 separate 4-position terminal strips located near the band heaters. The 4-position neutral and ground terminal strips located near the band heaters then had wiring ran into each band heater. All of this wiring can be seen in Figure *28*.

Next, all DC power used will be discussed. Figure *29* shows 2 of the 3 AC-DC power supplies used, the 3rd AC-DC power supply is located at the bottom of the tray between the two other power supplies.

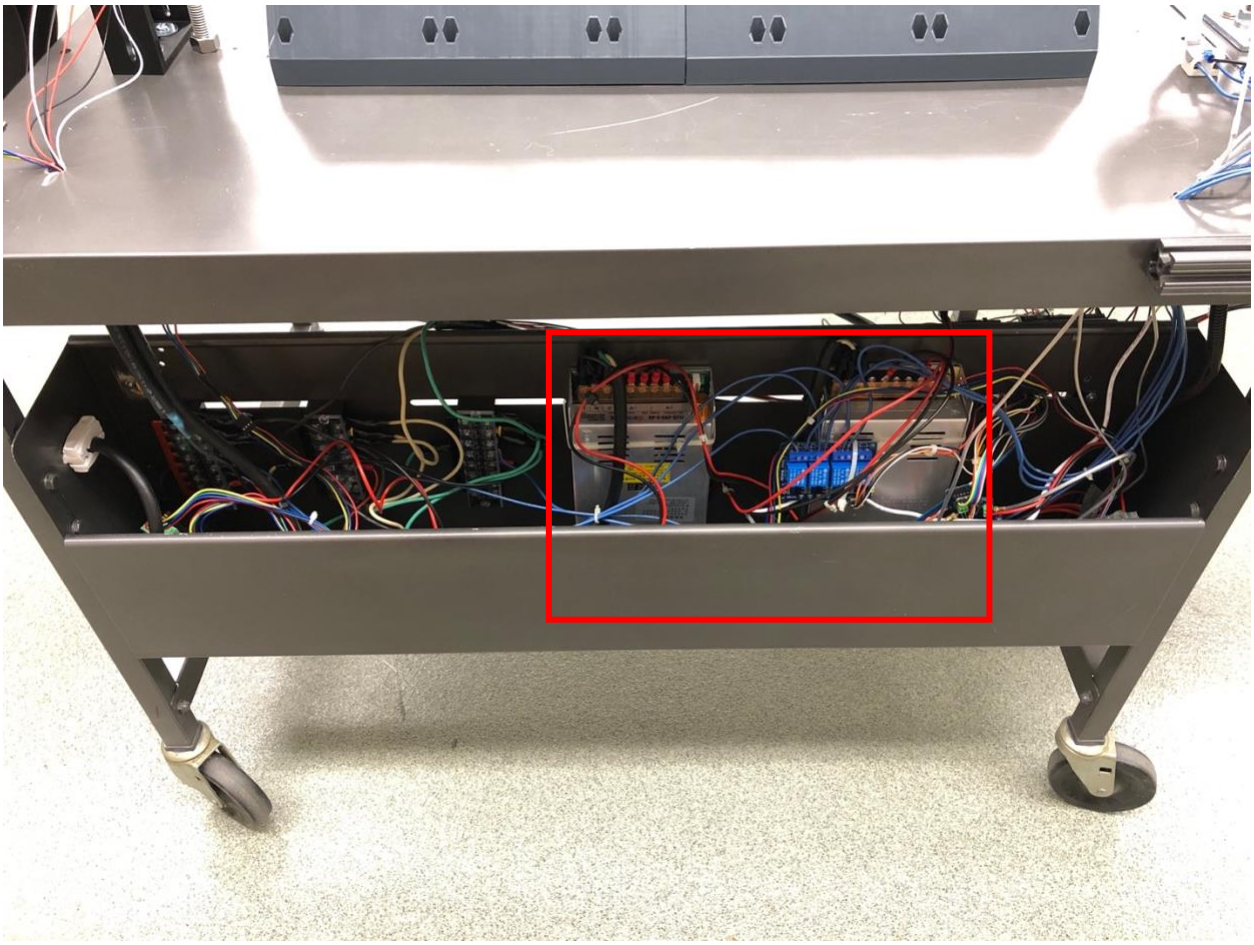

Figure 29: AC-DC Power Supplies

The power supply shown on the left (0-48V) in Figure *29* was used to power both stepper motor drivers, which in turn power the stepper motors themselves. This power supply was chosen to operate at 36V but can be varied between 20-48V for the stepper motor drivers. The power supply shown on the right (12V) in Figure *29* was used to power the Arduino responsible for driving the motors and reading associated inputs. The third power supply (low electrical noise 9V), which is not visible in Figure *29*, but is located at the bottom of the tray, was used to power the Arduino responsible for all other I/O functions.

An example of the stepper motor drivers used can be seen in Figure *30*.

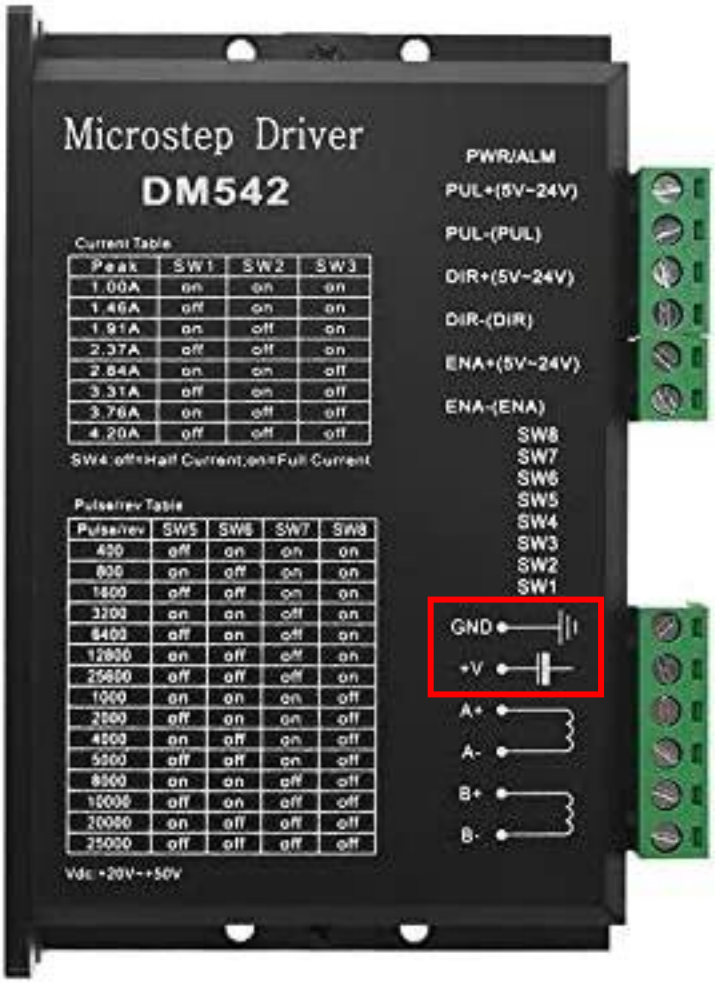

Figure 30: Stepper Motor Driver

Figure *30* shows how DC power is input into the stepper motor drivers. Wiring was ran from the 0-48V power supply into each of the stepper motor drivers in this manner.

Figure *31* shows the DC power input into each of the Arduinos.

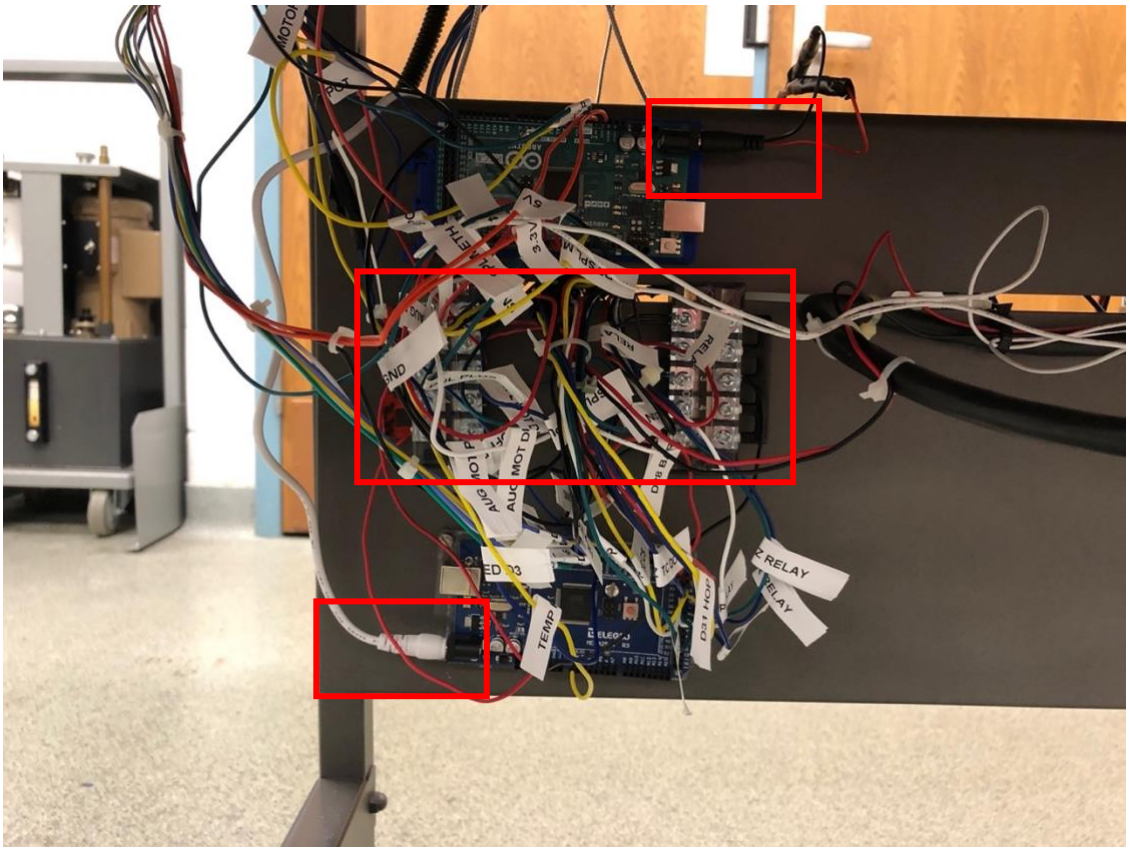

Figure 31: Arduino Input and Output Power

In Figure *31*, the upper Arduino is responsible for all motor-related functions, the bottom Arduino is responsible for all other functions (the rationale as to why two Arduinos had to be used will be covered in the troubleshooting section). The red/black power cord, running to the top Arduino, is from the 12V power supply. The white power cord, running to the lower Arduino, is a low electrical noise 9V power source (meaning it outputs 9V± *small* V). Initially we were using the 12V power supply for both Arduinos, but this caused interference with the readings from the sensitive thermocouple amplifying boards. Upon hooking the 12V supply up to an oscilloscope we found significant noise from the 12V power supply (meaning the output was 12V± *relatively-large* V). While this typically wouldn't adversely affect the Arduino's capability, the thermocouple amplifying boards require a very steady 3.3V for accurate readings. With the input power varying relatively significantly in voltage, a lot of noise was also generated in the 3.3V output of the Arduino, which caused issues reading the thermocouple values.

An alternative solution that was considered was constructing a PI-filter for the 12V power supply, but since we already had a lower noise 9V supply on hand, we decided to just use the available supply.

Figure *31* also shows the DC power output from the Arduinos. Many electrical components used to gather user input, display current conditions, and perform other functions rely on a 5V or 3.3V input (from the Arduinos). Since there are a limited number of 5V and 3.3V output pins on an

Arduino, 2 5-position terminal strips were used to power all relevant components from the Arduinos. The top 3 rungs of the left terminal strip in Figure *31* were used for 5V power, the bottom two rungs were used for 3.3V power. The terminal strip on the right was used for ground. The grounds of both Arduinos were also connected to one another. This concludes all power distribution used for the machine.

Figure *32*-Figure *33* display all user inputs and outputs added to the machine.

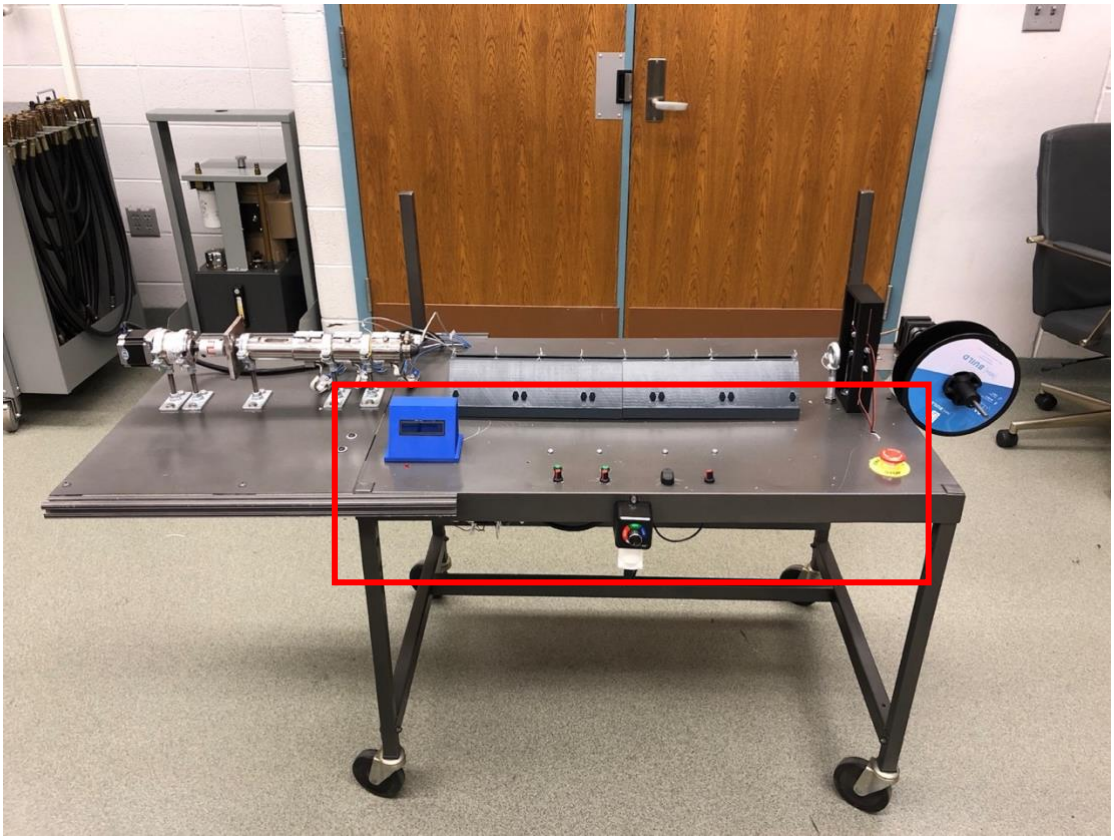

Figure 32: User Inputs and Outputs

The user inputs include:

- On/Off control of all band heaters using a latching push button (far left push button)
- Temperature set point control for the nozzle using a potentiometer (far left potentiometer)
- Auger motor on/off control using a latching push button (next to band heater push button)
- Auger motor speed control using a potentiometer (next to temperature control potentiometer)
- Mode switching rocker switch. One mode uses the IR break-beam sensors to control the spool motor speed, the other mode uses the potentiometer input for the spool motor speed
- Spool motor speed control using a potentiometer (far right potentiometer)
- LCD contrast adjustment using a potentiometer (see Figure *33*)
- Fan speed control using fan dimmer switch (mounted at the front and center of the cart)
- Emergency stop button that disables all input power into the cart (far right side of cart)

The user outputs include:

- Red LED that turns on when the extruder reaches a temperature of over 40°C (far left of the cart, below the LCD)
- 2x16 character LCD (2 rows, each at 16 characters) mounted on the far-left side of the cart. The LCD is currently configured to display nozzle set point temperature and thermocouple readings

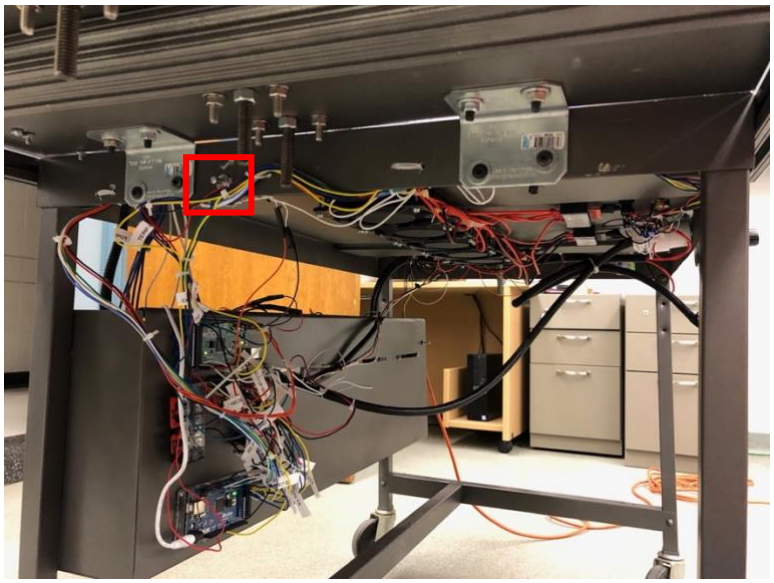

Figure 33: LCD Contrast Adjustment

In order to connect all user inputs, outputs, and remaining electrical devices, harnesses, such as that shown in Figure *34*, were constructed. The harnesses connected the devices to appropriate power supplies and Arduino pins. The harness shown in Figure *34* was used for the LCD wiring.

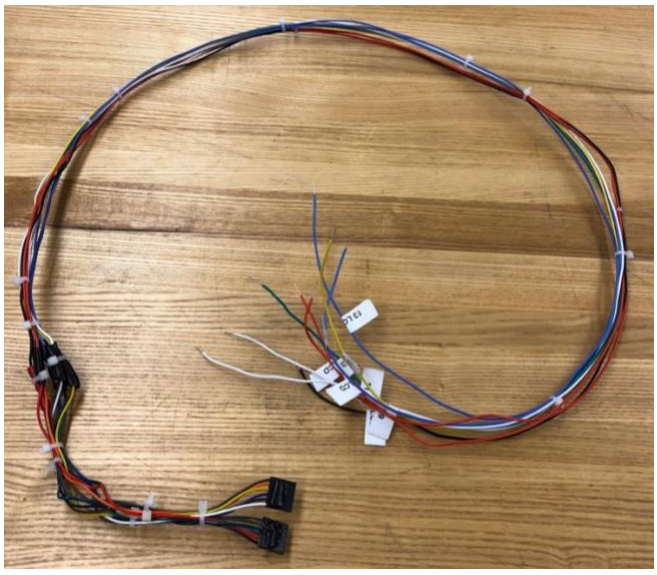

Figure 34: Harness Example

The pin-outs for both Arduinos are shown below in Table *1*.

Table 1: Arduino Pin-Outs

Arduino 1 (All Functions Except Motor Driving)		
Item	Pin	Type
Nozzle Thermocouple Clk	D25	Input
Nozzle Thermocouple CS	D26	Input
Nozzle Thermocouple D0	D27	Input
Barrel Thermocouple Clk	D28	Input
Barrel Thermocouple CS	D29	Input
Barrel Thermocouple D0	D30	Input
Hopper Thermocouple Clk	D31	Input
Hopper Thermocouple CS	D32	Input
Hopper Thermocouple D0	D33	Input
ON/OFF Band Heater	D2	Input
Set Temperature Potentiometer	A0	Input
Band Heater Nozzle Relay	D34	Output
Band Heater Barrel Relay	D35	Output
Band Heater Hopper Relay	D36	Output
LCD 4	D8	Output
LCD 6	D9	Output
LCD 11	D6	Output
LCD 12	D5	Output
LCD 13	D7	Output
LCD 14	D4	Output
Temperature LED Light	D3	Output
Arduino 2 (Motor Driving)		
Item	Pin	Type
Upper Break Beam Sensor	D23	Input
Lower Break Beam Sensor	D24	Input
Auger Motor On Off	D53	Input
Spool Method Switch	D12	Input
Auger Potentiometer	A0	Input
Spool Potentiometer	A1	Input
Auger Motor Pul+	D2	Output
Auger Motor Dir+	D3	Output
Spool Motor Pul+	D4	Output
Spool Motor Dir+	D5	Output

At this point, all wiring was completed, and we were able to move on to the next phase of the electrical design—coding the Arduinos.

Overview of Coding Process and Troubleshooting

Code was developed throughout the semester for each of the electrical devices used. Figure *35* displays the initial wiring and coding development for reading a potentiometer's analog value, formatting the reading, and displaying the information to an LCD display. Similar testing and development procedures were used to create standalone functions for each of the electrical components to be used, including all user inputs/outputs, thermocouple readings, and relay control.

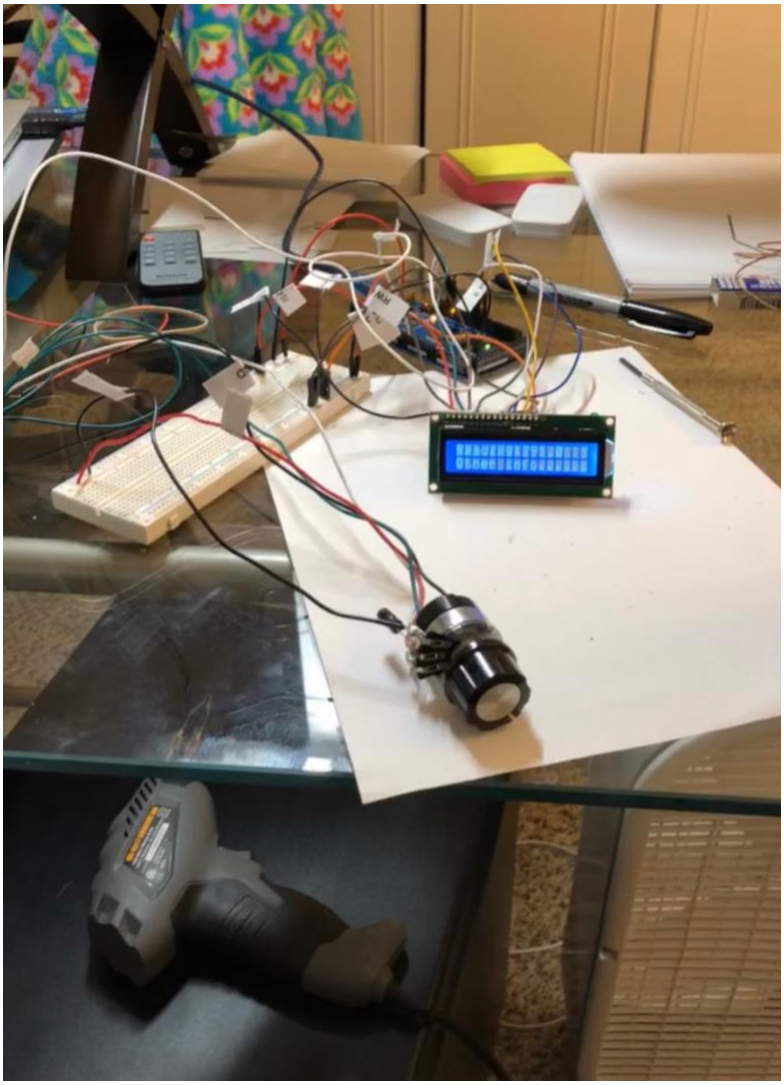

Figure 35: Code and Wiring Development

Once standalone code for each electrical component to be used was developed, the process of integrating all of the code into a single file began. The process of combining all of the code into a single file went smoothly, but quickly unearthed some unexpected issues.

The Thermocouple Conundrum

Using the standalone code for a single thermocouple reading, code to read three thermocouples was quickly developed. The readings from all three thermocouples were accurate, until they came into contact with a common electrically conductive surface—the extruder itself. In addition, the behavior exhibited by the thermocouple readings also seemed to be affected by the power supply for the Arduino. The readings would be accurate when the Arduino was solely connected to a computer (using it as the power source), but became quite inaccurate when the Arduino used the 12V power supply for its power source.

The solution to the first problem, when all three thermocouples were in contact with the extruder, had a relatively easy solution. Somehow, the connection between the thermocouples and the extruder had to be both electrically isolated (insulating) and thermally conductive. The issue is, a very small subset of materials offer both of these properties simultaneously, especially with the additional constraint of being able to maintain temperatures above 300°C. After doing some research and investigating various methods to achieve this goal, a solution was discovered. Among the thermocouple types (E, J, K, N, etc), there is a further distinction about how the connection is formed at the end of the thermocouple: ungrounded, grounded, and exposed, as shown by Figure *36*. Both the grounded and exposed variations offer an electrically and thermally conductive path to the object to be measured. However, the ungrounded variation offers an insulated electrical path and a thermally conductive path. This is achieved by separating the end of the thermocouple connection from a surrounding metal sheath using a material such as magnesium oxide (electrically insulating, thermally conductive). This allows a thermocouple to come into contact with an electrically noisy surface (such as the extruder barrel, with multiple band heaters and thermocouples) and still give an accurate reading. So, the first problem was resolved by replacing the existing exposed thermocouples with ungrounded thermocouples.

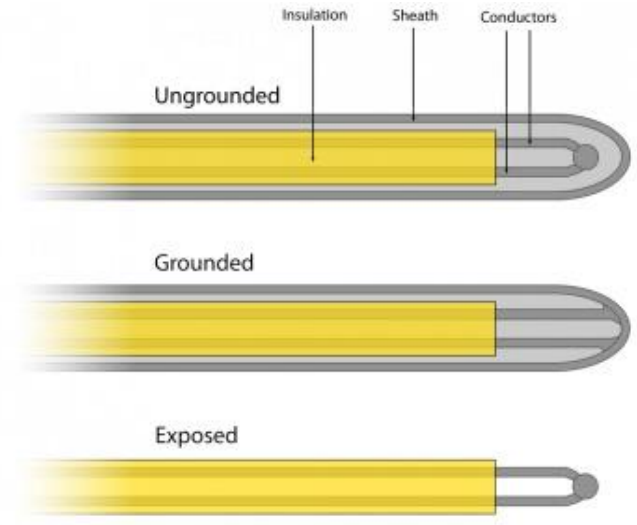

Figure 36: Thermocouple Connection Types

Next, the issue of getting different readings depending on the power source for the Arduino had to be resolved. The cart was brought to a laboratory in the ETCS building that had oscilloscopes and power supplies available, which is shown by Figure *37*. The existing 12V power supply was connected to an oscilloscope and the output voltage was inspected. The output from the power supply was quite noisy (meaning the voltage varied from 12V relatively significantly), especially in comparison to that of the power supply in the lab and another 9V power supply we had on hand. With confirmation that the 12V power supply was the source of this issue, we wired the 9V power supply to the Arduino.

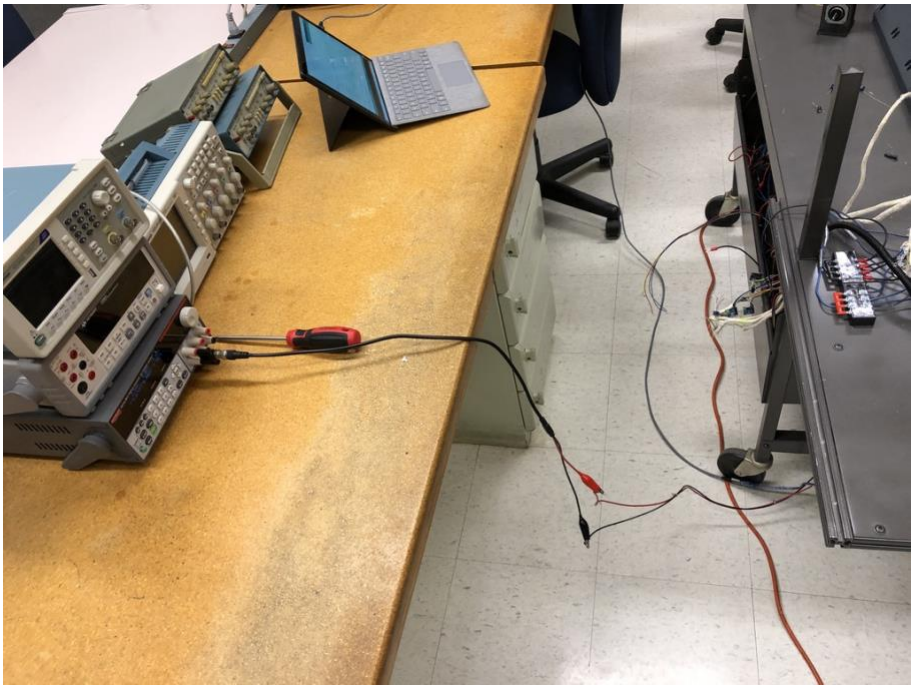

Figure 37: Power Supply Troubleshooting

After both solutions developed were implemented, the readings from the thermocouples were finally accurate, even with all three in contact with the extruder.

Why Two Arduinos Were Used

The next issue we discovered was related to the computational time for some of the functions used in the code. Figure *38* displays a portion of the datasheet for the MAX31855 amplifying boards that were used to gather thermocouple readings.

THERMAL CHARACTERISTICS (continued)

(3.0V ≤ V_{CC} ≤ 3.6V, T_A = -40°C to +125°C, unless otherwise noted.) (Note 4)

PARAMETER	SYMBOL	CONDITIONS	MIN	TYP	MAX	UNITS
Thermocouple Temperature Data Resolution				0.25		°C
Internal Cold-Junction Temperature Error		T_A = -20°C to +85°C (Note 3)	-2		+2	°C
		T_A = -40°C to +125°C (Note 3)	-3		+3	
Cold-Junction Temperature Data Resolution		T_A = -40°C to +125°C		0.0625		°C
Temperature Conversion Time (Thermocouple, Cold Junction, Fault Detection)	t_{CONV}	(Note 5)		70	100	ms
Thermocouple Conversion Power-Up Time	t_{CONV_PU}	(Note 6)	200			ms

Figure 38: MAX31855 Temperature Conversion Time

By reviewing Figure *38*, the temperature conversion time for the MAX31855 amplifying boards is typically 70 milli-seconds and is a blocking function—meaning that the next line of code will not compile until the function call finishes. While this may seem like a small amount of time, when it comes to driving a stepper motor 70 ms is far too significant of a delay. This is especially true when you consider that 3 amplifying boards were used, one for each thermocouple, meaning that the total time to take a single reading from each thermocouple was about 0.25 seconds. Figure 39 displays the applicable section of the AccelStepper library used for the stepper motors and indicates the frequency at which the runSpeed() function must be called.

Figure 39: AccelStepper runSpeed() Function Call Frequency

While the information in Figure *39* doesn't display numeric values for the frequency at which the function must be called, it can be calculated based on specifics to the stepper motor and stepper motor driver. Equation 1 indicates this conversion.

$$\Delta t = \alpha * \rho * \omega \quad (1)$$

With:
Δt=maximum allowable time between function calls (seconds)
α=steps per revolution selected on stepper motor driver (steps/revolution)
ρ=gear ratio of stepper motor (if applicable)
ω=desired angular velocity of the stepper motor (revolutions/second)

The following is an example of using Equation (1) for the auger motor.

Setting the stepper motor driver shown in Figure *30* to 400 steps per revolution (the lowest possible value), the stepper motor's gear ratio to 15.3:1, and the angular velocity to 1 revolution/6 seconds (10 RPM, which should be approximately the rate required to produce 1kg of filament in an hour), the maximum time allowable between function calls is approximately 1 ms (which is equal to the amount of time for one step interval). Comparing this result with the conversion time required for the thermocouple amplifying boards (70 ms each), it is clear why the stepper motors weren't working properly when attempting to read thermocouple values and drive the stepper motors from a single Arduino. In light of this issue, a separate Arduino was used to drive the stepper motors, leaving all temperature sensing to the other Arduino.

This type of issue was not limited to thermocouple measurements, even the time required to gather a reading from an analog pin connected to a potentiometer presented issues. This was especially the case when attempting to drive two stepper motors and read two different potentiometers (each one controlling one motor's speed). In order to combat this issue, the analogRead() function for the Arduino had to be customized to make it non-blocking. Fortunately, we were able to successfully code a non-blocking version of the analogRead() function to resolve this issue. In spite of all the issues faced with wiring and compiling all the standalone code into a single script, some of which have been explained above, we were able to successfully integrate all desired inputs, outputs, and other necessary electronics using both Arduinos.

Section 1.10: Temperature Control Arduino (1) Functions and Code Logic

There was a specific vision for the functionalities of the heating system when beginning the programming and build. The idea began with a potentiometer or knob that is able to control the temperature at which the nozzle is set, as well as a push button to control when the heaters functions are able to turn on. Due to the simulations in the design phase, we knew that each band heater would have to have different settings for when to turn on and off because of the nonlinear temperature distribution required, so the idea was to have “calibration constants” governing the regulation of the band heaters based on the readings of thermocouple closest to it. This is the principle in which the heating functions operate, controlled by the Arduino (1).

The first Arduino that was connected controls the temperature functions. There are three functions uploaded that work in unison: Temperature_Sensing, Band_Heaters, and HEATING_FUNCTIONS_ARD1.

HEATING_FUNCTIONS_ARD1

The heating functions class serves as the main class, calling the band heaters and temperature sensing functions at different times throughout the code. Before performing any actions, many global variables are declared and initialized in this class. This includes setting constants, Arduino pin locations, and other variables important to the logic. In this specific class, there are a few variables that can and should be changed by the user depending on the situation, to be explained further. The main class has two functions inside, one to execute the setup one single time, and a main loop function that is continuously running over and over. Inside the setup function is the initialization of the serial monitor that was used during the troubleshooting phase to print various values and verify the code was working as intended. Next, the pinModes were activated for the band heater push button, the LED warning light, and the pins connecting to each of the band heater relays so that the Arduino is able to send or receive the correct signal to each of the devices. Finally, the setup portion ends with initializing variables pertaining to the dissection of the analogRead function and initializes the setup of the LCD functions.

Moving onto the looped portion of the main class, this begins with the Temperature_Sensing function, which is described below. The purpose of running this first is to just gather the current and average temperatures (over the defined sample size) at each of the thermocouples. The next line is to read the status of the push button that is governing the on/off status of the band heaters. For the final reading in the loop, the function reads the value of the potentiometer and maps it according to a maximum potentiometer temperature that is initialized with the global variables. This value is free to be changed by the user by simply going into the code and changing it to the new desired value and reuploading the code. This temperature set by the potentiometer is referred to as the "Set Point Temperature”. Next, the main class prints the desired information to the LCD screen, and in the last iteration of this, the LCD prints on the top line: “Set Temp:

####", with the bottom line printing the current reading from each of the thermocouples with a space between. The order of the temperatures reads left to right, corresponding with the left to right thermocouples on the extruder. The code updates these readings once every second, due to the constant update causing visibility issues and blurriness. An example of the LCD reading these values is shown below in Figure *40*. Finally, the main class ends by checking the status of the on/off push button and activates the Band_Heaters function if it is on, which is explained further below.

Figure 40: Example LCD screen reading.

Temperature_Sensing Function

The first "side" function of the Arduino 1 is Temperature_Sensing. This code uses the library from the Adafruit_MAX31855 thermocouple amplifying boards in order to read the thermocouples in real time with a calibrated value. The code takes these readings and stores them in arrays of a designated "sample size", and outputs the average of this array to the main class. The function also outputs the reading of each of the three thermocouples to the main function. Knowing the real time temperature at each of the three points is imperative to calibrate and regulate the temperature of the entire barrel and when each of the band heaters is to be energized. There is a large amount of error handling done in this function. It is unlikely that every reading coming from each thermocouple is perfect, and there are infrequent NaN error readings thrown out from the thermocouples. In order for these to not mess with the average temperature or current readings, there is logic to filter out the rare error readings. This function ends by turning on the LED warning light if the current nozzle reading exceeds 40 ˚C.

Band_Heaters Function

Moving onto the next complementary function within the heating system, Band_Heaters. This function takes the temperature set by the potentiometer reading and sets regulation temperatures based on this value. The relays are then told to trigger based on a comparison to the readings of each of the thermocouples. The calibration constants were determined experimentally, and these describe the percentage of the set point temperature the regulation temperatures are to hold for each band heater. The goal was to set a nonlinear temperature distribution that is able to be maintained by the function being repeated over and over. There are tolerances built into the if statements that allow for some room for the temperatures to bounce between a small range but be mostly maintained at the set regulation temperature. The calibration constant of the nozzle is set to 1 so that the set point temperature can match the nozzle temperature reading.

Section 1.11: Motor Control Arduino (2) Functions and Code Logic

After determining that a second Arduino was necessary, the code governing both motors were swapped over. This code uses the AccelStepper library to allow both motors to move simultaneously. The motor control for the spooling motor and auger motor are contained in the same class. For each of the motors, there is a variable defined in the global variables section that sets the speed of the upper limit on the potentiometer. These speeds within the code do not have correlating units but were rather determined experimentally since there is not a conversion to a normal speed unit available from the arbitrary values that the library uses. Once all of the necessary variables and pinouts are initialized, the class moves to the setup function where the pinMode was set for the on/off auger motor button, and the spooling motor rocker switch. With the AccelStepper library, a "maximum speed" of each motor must be defined, so this is also done inside of the setup function.

The main loop function begins by reading the value of the auger potentiometer, converts it to a speed based on the previously defined upper limit, then does the same for the potentiometer governing the spooling motor. Next, the code takes those converted speeds and sets the speed for the motor and tells it to run. For the auger motor, the sign (+-) of the speed is swapped so that it defaults to the correct spinning direction. As the potentiometers are adjusted, the motor speeds change instantaneously, which was made possible by using multiple Arduinos.

Section II: Testing and Evaluation

Section 2.1: Testing Plan

In order to ensure that the device met the design requirements, the adjustability of the temperature at the nozzle, the capability of the nozzle to reach 300 °C, and the diameter of the produced filament had to be measured or tested. The nozzle temperature is adjusted by using a potentiometer and viewing the corresponding set point temperature requested on an LCD display. The nozzle temperature is measured by a thermocouple secured near the exit of the nozzle. After ensuring both the adjustability of the nozzle temperature and capability of the nozzle to reach a temperature of 300 °C, filament should be extruded and measured to ensure that its diameter is within the tolerance of 1.75mm ±0.05mm.

Section 2.2: Testing Results

Nozzle Temperature Adjustability and Capability

In order to test the nozzle temperature capability, three different temperature measurement methods were implemented:

1.) The ungrounded Type-K thermocouple mounted to the nozzle and read from the Arduino
2.) An exposed Type-K thermocouple connected to a Klein Tools MM700 Multimeter
3.) A FLIR E8-XT Infrared Camera

Six different temperatures were tested: room temperature, and a nozzle set point temperature of 100°C, 200°C, 250°C, 300°C, and 325°C.

Figure *41*-Figure *46* display the results of testing the nozzle temperature capability.

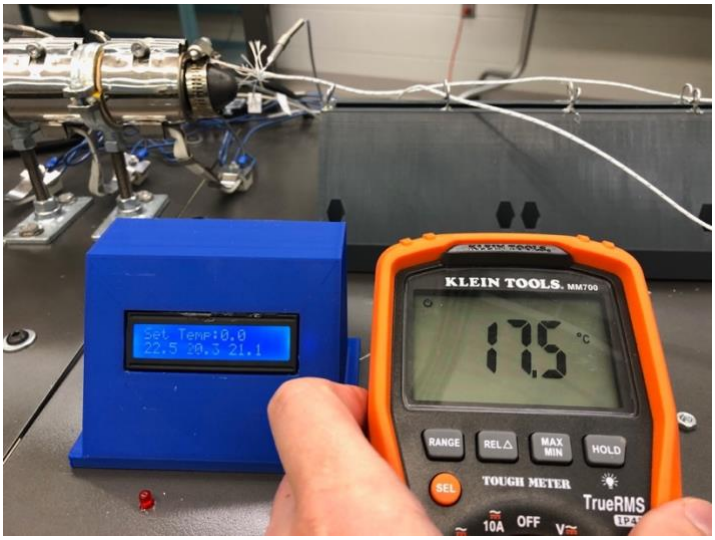

Figure 41: Room Temperature

The room temperature test was completed before energizing the band heaters at all. The LCD is displaying the current set point temperature for the nozzle in degrees Celsius and the readings

from each of the thermocouples attached to the extruder. The furthest left reading coincides with the thermocouple nearest the hopper, the middle reading coincides with the thermocouple attached between the last two band heaters, and the furthest right reading coincides with the thermocouple attached to the nozzle with a hose clamp (the thermocouple for the Klein multimeter is also attached to the nozzle using the hose clamp).

Figure 42: 100°C

Figure *42* displays the results for the nozzle set point temperature of 100°C. The multimeter and nozzle thermocouple indicated a reading of 100.3°C, the IR camera indicated a reading of 100.0°C.

Figure 43: 200°C

Figure *43* displays the results for the nozzle set point temperature of 200°C. The multimeter indicates a reading of 202.1°C, the nozzle thermocouple indicates a reading of 200°C (note that when displaying all three attached thermocouple values the precision of the nozzle temperature is limited by the character limit of the LCD), and the IR camera indicates a reading of 199°C.

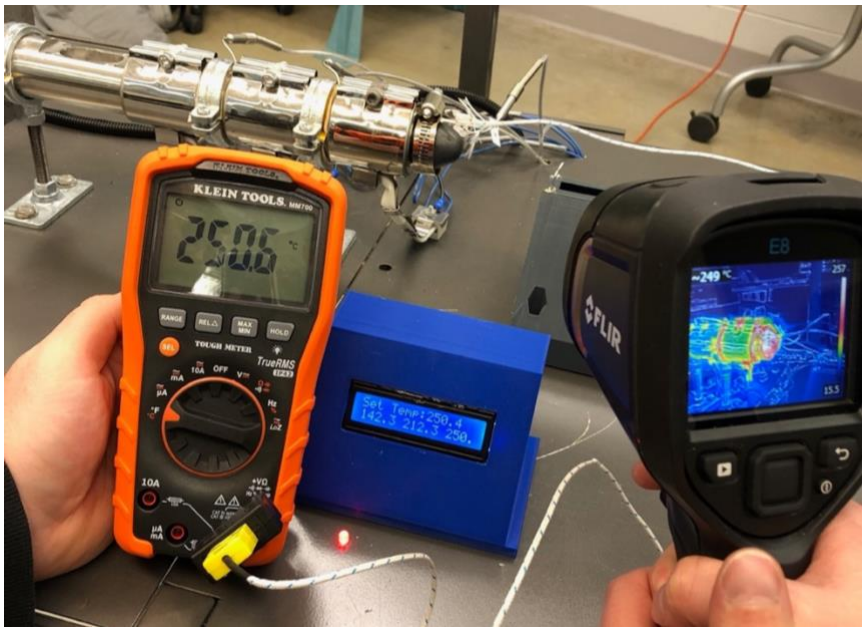

Figure 44: 250°C

Figure *44* displays the results for the nozzle set point temperature of 250 °C. The multimeter indicates a reading of 250.6 °C, the nozzle thermocouple indicates a reading of 250 °C, and the IR camera indicates a reading of 249 °C.

Figure 45: 300°C

Figure *45* displays the results for the nozzle set point temperature of 300 °C. The multimeter indicates a reading of 297.8 °C, the nozzle thermocouple indicates a reading of 301 °C, and the IR camera indicates a reading of 298 °C.

Figure 46: 325°C

Figure *46* displays the results for the nozzle set point temperature of 325°C. The multimeter indicates a reading of 322.5°C, the nozzle thermocouple indicates a reading of 327°C, and the IR camera indicates a reading of 326°C. Table 2 summarizes the results shown in Figure *41*-Figure *46*.

Table 2: Nozzle Temperature Adjustability and Capability Results

Nozzle Set Point Temperature (°C)	Nozzle Thermocouple Measurement from Arduino (°C)	Nozzle Thermocouple Measurement from Klein Tools MM700 Multimeter (°C)	Nozzle Thermocouple Measurement from FLIR E8-XT Infrared Camera (°C)
Room Temperature (~20)	21.1	17.5	N/A
99.9	100.3	100.3	100.0
200.1	200	202.1	199
250.4	250	250.6	249
300.0	301	297.8	298
325.4	327	322.5	326

By reviewing Figure *41*-Figure *46* and Table *2*, it is clear that the nozzle is capable of reaching temperatures in excess of 300 °C and is adjustable, meeting the following design requirements:

- The temperature of the plastic at the nozzle should be capable of reaching 300 °C to ensure compatibility with a wide variety of materials.
- The temperature of the molten material at the nozzle should be adjustable.

Filament Production

The remaining design requirement is as follows:

- The extruded filament must have a diameter of 1.75 mm with a tolerance of ± 0.05 mm.

Figure *47*-Figure *48* display the successful capability of the machine to produce filament from pellets of PLA.

Figure 47: Hopper full of PLA Pellets

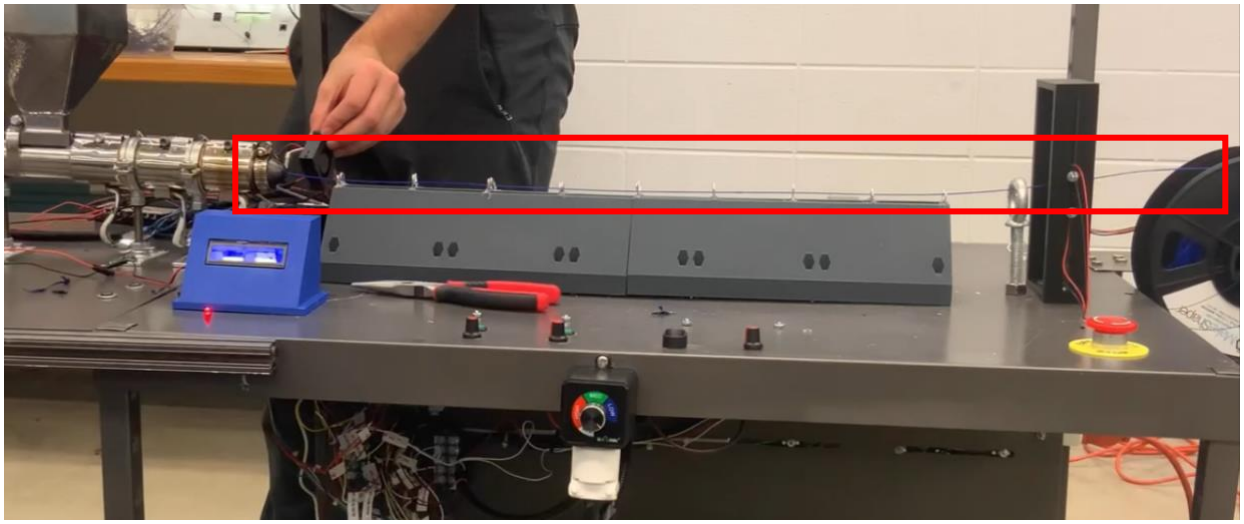

Figure 48: Filament Production

Although the machine is capable of producing filament at a diameter of 1.75 mm from pellets of PLA, as shown by Figure *49*, the ability to maintain a tolerance of ± 0.05 mm has not been achieved. Figure *50*- Figure *51* indicate two other diameter measurements of extruded filament outside of the allowed tolerance.

Figure 49: Extruded Filament at Target Diameter

Figure 50: Extruded Filament above Target Diameter

Figure 51: Extruded Filament below Target Diameter

The most significant source of error in meeting the filament diameter tolerance is stemming from an inability to consistently input material into the extruder. The cause of this error is that the joint between the hopper and the barrel is too thermally conductive, resulting in the hopper reaching a higher temperature than it should, which leads to filament prematurely melting at the outlet of the hopper. Eventually, the molten filament plugs up the entirety of the hopper outlet and ceases the ability to produce filament at all since material can no longer enter the extruder. Figure *52* indicates this source of error. The solution to this issue would be to thermally isolate the hopper from the extruder barrel via an adapter plate (with the same bolt pattern as the extruder) comprised of a material with a low thermal conductivity, such as a machinable ceramic. Unfortunately, this issue was discovered too late into the testing phase to correct it in the allotted time period.

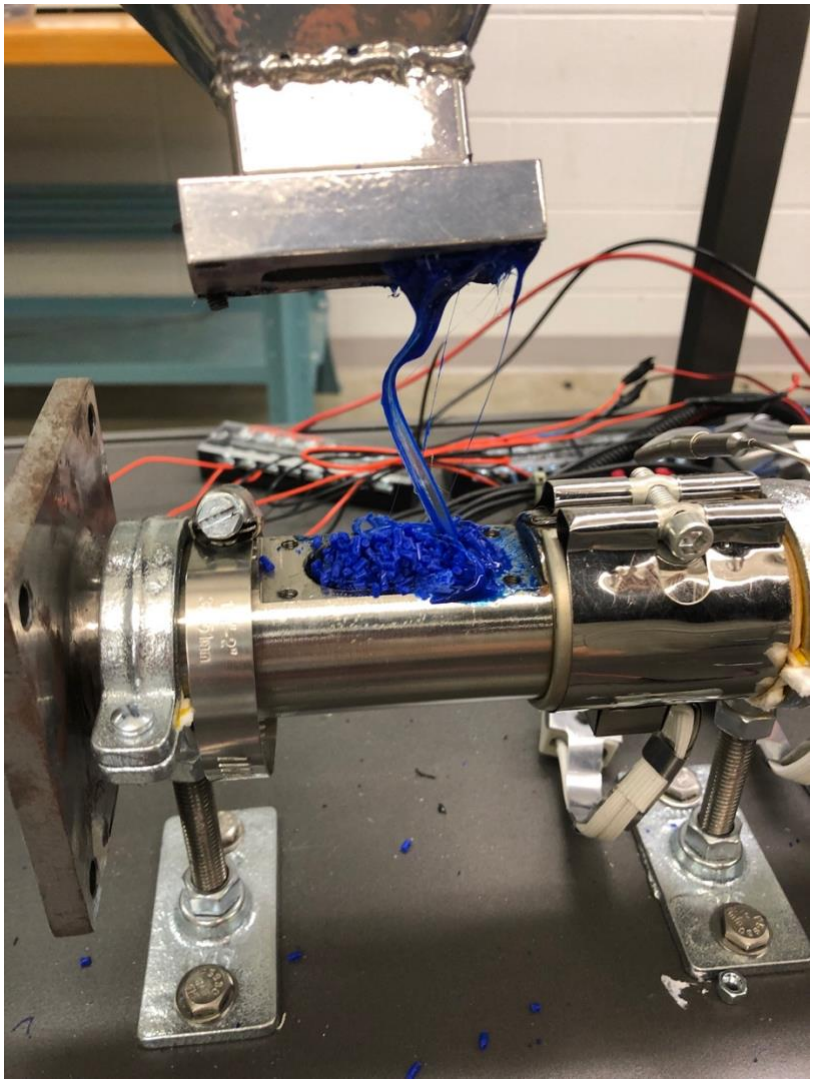

Figure 52: Material Input Source of Error

Section III: Cost Analysis

Section 3.1: Cost Analysis

The budget allotted for this project was $1000. Our initial cost estimate was $921.25. The total cost ended up being $1,561.89. Table *3* displays the costs incurred during the build process of this project.

Table 3: Cost Analysis

Mechanical Costs				
Item	Supplier	Cost/Item ($)	Quantity	Total Cost ($)
Extruding Barrel and Auger Screw	(eBay)	$369.15	1	$369.15
Cooling Fans (120mm)	Amazon	$16.11	5	$80.55
Sheet Metal for Hopper, 36"x36" 18G		$65.05	1	$65.05
Auger Screw Drive Motor (Nema 23 15:1)	Stepper Online	$53.50	1	$53.50
Band Heaters (250W, 2-pack)	Amazon	$22.99	2	$45.98
24"x24" 16G Sheet Metal	Lowes	$43.98	1	$43.98
Fasteners from McMaster (fans, threaded rod)	McMaster Carr	$37.07	1	$37.07
Gunmetal Gray Spray Paint	Lowes	$7.48	3	$22.44
Split Ring Pipe Hangers	Lowes	$20.11	1	$20.11
7pc Drill Bit Set	Harbor Freight	$12.99	1	$12.99
3/8" Ceiling Flange	Lowes	$1.89	6	$11.34
1.25" Galvanized Split Ring Hanger	Lowes	$1.93	5	$9.65
Auger Shaft Coupler (12 to 14mm)	Stepper Online	$6.96	1	$6.96
Copper Aluminum Wire (for guides on duct)	Amazon	$6.59	1	$6.59
3M Sandpaper	Lowes	$5.97	1	$5.97
3/8" Hex Nut (10 packs)	Lowes	$2.84	2	$5.68
1/4-20 Bolts	Lowes	$0.44	12	$5.28
1/4" SS Flat Washer (25 pck)	Lowes	$5.00	1	$5.00
3M Sanding Blocks (2 pack)	Lowes	$4.97	1	$4.97
1.5" Galvanized Split Ring Hanger	Lowes	$2.48	2	$4.96
Rustoleum Cystal-Clear Clear Coat	Lowes	$4.58	1	$4.58
1/4" SS Hex Nut	Lowes	$0.28	12	$3.36
Machine Screw Kit	Harbor Freight	$2.99	1	$2.99
2"x3" L Bracket	Lowes	$1.08	2	$2.16
Mounting Bracket Nema 23	Stepper Online	$1.80	1	$1.80
Total Mechanical Costs (with Tax):				$928.31
Electrical Costs				
Item	Supplier	Cost/Item ($)	Quantity	Total Cost ($)
Type K Thermocouples, ungrounded	Amazon	$39.99	3	$119.97
Arduino	Amazon	$45.45	1	$45.45
Power Supply (0-48V 480W)	Amazon	$39.89	1	$39.89

16/3 16G Wire 25'	Amazon	$25.00	1	$25.00
Stepper Motor Driver (20-50V, 4.2A max)	Amazon	$22.29	1	$22.29
Auger Motor Stepper Driver (20-50V 4.2A Max)	Stepper Online	$19.90	1	$19.90
MAX31855 Amplifier Boards (4pk)	Amazon	$19.39	1	$19.39
12V 30A DC Power Supply 360W Max	Amazon	$18.98	1	$18.98
16/2 Wire 60'	Amazon	$17.88	1	$17.88
22G Solid Core 6 Color Wire (30'/color)	Amazon	$16.49	1	$16.49
4x qty 8 Positions Terminal Strips	Amazon	$15.49	1	$15.49
12v DC power converter	Amazon	$13.89	1	$13.89
Adafruit Breakbeam Sensors (3mm)	Amazon	$6.86	2	$13.72
Terminal Strips	Amazon	$12.99	1	$12.99
520 Piece Terminal Set	Harbor Freight	$12.49	1	$12.49
16/2 Lamp Cord 25'	Lowes	$12.43	1	$12.43
22G Solid Core 6 Color Wire (33'/color)	Amazon	$11.99	1	$11.99
Fan Speed Controller/Dimmer	Amazon	$11.99	1	$11.99
5k Linear Potentiometer	D.I.Y. it!	$3.99	3	$11.97
Emergency Stop Button	D.I.Y. it!	$10.99	1	$10.99
14 G Solid Core Wire (Blue)	Lowes	$0.27	40	$10.80
16/3 300V Power Cord 12' @ $0.81/ft	Lowes	$0.81	12	$9.72
9V Power Supply Adapter	Amazon	$9.50	1	$9.50
USB Male to Female Cord 6.6' (2 pk)	Amazon	$8.49	1	$8.49
16 G Black Primary Wire	Lowes	$6.03	1	$6.03
14 G Red Primary Wire	Lowes	$6.03	1	$6.03
16 G Red Primary Wire	Lowes	$6.03	1	$6.03
14 G Black Primary Wire	Lowes	$6.03	1	$6.03
3/4" x 60" Electrical Tape (10pk)	Harbor Freight	$5.99	1	$5.99
125V 15A Male Plug Connecter	Harbor Freight	$2.99	2	$5.98
125V 15A Female Plug Connecter	Harbor Freight	$2.99	2	$5.98
Potentiometer Control Knob	D.I.Y. it!	$1.99	3	$5.97
4 Relay Board	Amazon	$4.89	1	$4.89
Latching Push Button	D.I.Y. it!	$2.29	2	$4.58
10-12 G Butt Connecter 10pk	Harbor Freight	$3.99	1	$3.99
Rocker Switch	D.I.Y. it!	$3.99	1	$3.99
36" Test Leads	Harbor Freight	$2.99	1	$2.99
14-16G Butt Connectors (10pk)	Harbor Freight	$2.99	1	$2.99
1/4" x 14' Corregated Black Tubing	Harbor Freight	$2.99	1	$2.99
1/4" x 8' Black Tubing (shrink wrap)	Harbor Freight	$1.99	1	$1.99
11/64" x 8' Red Tubing (shrink wrap)	Harbor Freight	$1.99	1	$1.99
18-22 G Butt Connectors (10pk)	Harbor Freight	$1.99	1	$1.99
Total Electrical Costs (with Tax):				$633.58
Miscellaneous Shipping Costs:				$35.47
Grand Total:				**$1561.89**

From Table 3, it is evident that a significant portion of the cost for this project was incurred in miscellaneous electrical components, wiring, and assorted wiring termination methods. In total, approximately 40% of the cost for the project was “electrical.” When we determined our cost estimate, we didn’t have many details for the wiring/control system figured out, so it was difficult to anticipate what the costs would be. In addition, the necessity for ungrounded thermocouples was something we didn’t anticipate and added almost $120 to the total cost of the project.

Conclusion

The build process of this project began with acquiring and repurposing an extra cart from the PFW CME Department that could be used to fasten all of the required mechanical and electrical components to. After the cart was extended and painted, all required mechanical and electrical components were incrementally added to the cart. The mechanical mounting of components presented very few issues, but the electrical components proved to be more time consuming due to the extensive amount of wiring that was required. In addition, developing all of the required Arduino code to integrate with the electrical components was a significant undertaking. Although the team was initially inexperienced with the level of wiring presented by this project and the syntax used with the Arduinos, we were able to draw from our academic and professional knowledge to surmount any obstacles we encountered.

Overall, the testing process proved the machine's capability of reaching a nozzle temperature of 300°C and that this temperature is easily adjustable. In addition, the testing process proved the machines capability to produce filament at a diameter of 1.75 mm. Future improvements could be made to the machine's material input subsystem, which the team believes would allow for the tolerance of ±0.05 mm to be achieved.

References

1. MAXIM. *MAX31855 Cold-Junction Compensated Thermocouple-to-Digital Converter*. PDF. February 2012.

2. McCauley, Mike. "AccelStepper." AccelStepper: AccelStepper Library for Arduino. April 20, 2020. Accessed December 04, 2021. https://www.airspayce.com/mikem/arduino/AccelStepper/.

Appendices

MOTOR_FUNCTIONS_A2

```
//ME-488 Senior Design II Extruding Machine Stepper Motor Code

//Fall 2021

//Developed by Mason Averill and Kade Bontrager




//----------VARIABLES THAT ARE ABLE TO BE CHANGED BY USER WITHOUT CAUSING ERROR----------


//Initialize the maximum motor speeds for the potentiometers maximum setting

//These values are arbitrary, so they are scaled experimentally for the motor speeds. An increase in value will result in a higher motor speed

//If you increase either value below, be sure to check the maximum motor speeds set in the setup portion of the code, each value below MUST

//be less than the maximum motor speed set for each motor

  double aug_Max_Motor_Speed = 4000;

  double spl_Max_Motor_Speed = 4000;


//--------------------------------------------------------------------------------


//Import necessary libraries (if you receive an error when compiling/verifying this code, make sure you have this library added!)

#include <AccelStepper.h>



//Set Stepper Motor Pins and initialize stepper motor objects

const int augMotPul = 2;
```

```
const int augMotDir = 3;
const int splMotPul = 4;
const int splMotDir = 5;

AccelStepper stepper1(AccelStepper::DRIVER, augMotPul, augMotDir);
AccelStepper stepper2(AccelStepper::DRIVER, splMotPul, splMotDir);



//Set pin locations for the on/off switches
  int spl_Method_Toggle_Pin = 12;
  #define aug_On_Off_Pin 53

//Set analog pins for the potentiometer readings
  const byte aug_Pot_Analog_Pin = 0;  //Set analog pin to A0
  const byte spl_Pot_Analog_Pin = 1;  //Set analog pin to A1

//Initialize statuses for the potentiometers and on/off switches
  int aug_On_Off_Status = 1;
  double aug_Pot_Analog_Reading = 0;
  double spl_Pot_Analog_Reading = 0;
  int spl_Method_Status;

//Initialize motor speed variables for each motor
  double aug_Motor_Speed;
  double spl_Motor_Speed;



//-----Weird stuff to make AnalogRead non blocking------
```

```
//this is for the Auger potentiometer
bool working;

//this is for the Spool potentiometer
bool working1;

int code_Status = 0;

//-----------------------------------------------------

//--------------------------------Begin Setup--------------------------------
void setup() {

  Serial.begin(9600);

  //More weird stuff to make analogRead non-blocking
  ADCSRA = bit (ADEN);
  ADCSRA |= bit (ADPS0) | bit (ADPS1) | bit (ADPS2);



  //Set pinModes for the push button and rocker switch
  pinMode(aug_On_Off_Pin,INPUT_PULLUP);
  pinMode(spl_Method_Toggle_Pin,INPUT_PULLUP);



//Set maximum speed for each motor REQUIRED BY ACCELSTEPPER FUNCTION
  stepper1.setMaxSpeed(4500); //controls the maximum allowable speed for the auger motor
```

```
  stepper2.setMaxSpeed(4500); //controls the maximum allowable speed for the spool motor

}
//-------------------------------End Setup--------------------------------



//-------------------------------Begin Main Loop----------------------------

void loop() {

//--------alternating status to read each potentiometer separately------- begin with auger motor
if(code_Status == 0){

  //----------------------------Begin Auger Potentiometer Reading in a Non-Blocking Manner--------
-------------------------
  ADMUX = bit (REFS0) | (aug_Pot_Analog_Pin & 0x07);
  if(!working)
  {
    bitSet (ADCSRA, ADSC);
    working = true;
  }
  if(bit_is_clear(ADCSRA,ADSC))
  {
  double value = ADC;
  working = false;
  aug_Pot_Analog_Reading = value;
  code_Status = 1;
  }
```

```
}
//----------------------------End Auger Potentiometer Reading in a Non-Blocking Manner----------
--------------------



//Read status to determine if it is ready to read potentiometer 2------ spooling motor
if(code_Status == 1){
  //----------------------------Begin Spool Potentiometer Reading in a Non-Blocking Manner---------
-----------------------
  ADMUX = bit (REFS0) | (spl_Pot_Analog_Pin & 0x07);
  if(!working1)
  {
    bitSet (ADCSRA, ADSC);
    working1 = true;
  }
  if(bit_is_clear(ADCSRA,ADSC))
  {
  double value1 = ADC;
  working1 = false;
  spl_Pot_Analog_Reading = value1;
  code_Status = 0;
  }
}
//----------------------------End Spool Potentiometer Reading in a Non-Blocking Manner-----------
--------------------



//----------------------------Begin spool Motor Turning Logic based on on/off--------------------
-------
```

```
//Read status of on/off button for spooling motor, then turns the motor if it is on and potentiometer has a value greater than zero
  spl_Method_Status = digitalRead(spl_Method_Toggle_Pin);

  if(spl_Method_Status == 1){
    spl_Motor_Speed=spl_Pot_Analog_Reading*spl_Max_Motor_Speed/1023;

  if(spl_Motor_Speed>0)
    {
    //sets the speed and runs the motor in the arbitrary units for the AccelStepper functions
    stepper2.setSpeed(spl_Motor_Speed);
    stepper2.runSpeed();
    }

  }
//----------------------------End spool Motor Turning Logic based on on/off----------------------------------

//----------------------------Begin Auger Motor Turning Logic based on on/off-------------------------------

//Read status of on/off button for auger motor, then turns the motor if it is on and potentiometer has a value greater than zero
  aug_On_Off_Status = digitalRead(aug_On_Off_Pin);

  if(aug_On_Off_Status == 0)
  {
  aug_Motor_Speed=aug_Pot_Analog_Reading*aug_Max_Motor_Speed/1023;
```

```
  if(aug_Motor_Speed>0)
    {
    //sets the speed andn runs the motor in the arbitrary units for the AccelStepper functions, must be negative in next line for proper spinning direction
    stepper1.setSpeed(-aug_Motor_Speed);
    stepper1.runSpeed();
    }
  }
//----------------------------End Auger Motor Turning Logic based on on/off-------------------------------


//REPEATS LOOP PORTION


}
```

HEATING_FUNCTIONS_ARD1_mk2

Must be used in the same folder and file path as the Band_Heaters_mk2 and Temperature_Sensing

```
//ME-488 Senior Design II Extruding Machine Heating Functions Code

//Fall 2021

//Developed by Mason Averill and Kade Bontrager



//THIS CODE IS MARK 2, WHERE THE NOZZLE BAND HEATER IS CONTROLLED BY THE THERMOCOUPLE AT THE BARREL (MIDDLE THERMOCOUPLE). THIS WAS MODIFIED TO MARK 2 WHEN ISSUES PERTAINING TO COOLING AT THE EXIT AROSE.

//THE COOLING AT THE EXIT DISTORTED THE READINGS OF THE NOZZLE THERMOCOUPLE, MAKING THE NOZZLE BAND HEATER BE ON AT ALL TIMES, HEATING THE FILAMENT TOO MUCH.


//-------------------------THIS CODE MUST BE USED WHEN THERE IS EXTRA COOLING DIRECTLY AT THE NOZZLE, FROM FANS, BLOWERS, ETC.-----------------------------------------


//USER CONTROLLED VARIABLES, THESE ARE THE ONLY CHANGEABLE VALUES THAT WILL NOT CAUSE LOGICAL ERRORS, THESE ARE USED FOR CALIBRATION AND SETTING POTENTIOMETERS


    //Define maximum temperature that can be set by the potentiometer (in degrees Celcius)

    const double max_set_temp = 310


    //Calibration constants determined by experimentation, describes the scalar value between the set point temperature and the actual temperature at the location

    //These values are to be used as guidelines to regulate the band heaters being on or off in order to ensure that the nozzle temperature matches the set point temperature
```

```
        //Nozzle_Calibration_Constant should ideally be 1 and unchanged, since the set point temperature should describe the temperature here
        //Ex: for a requested nozzle temperature of 100, a value of 1.2 for the hopper calibration constant would result in the temperature at the hopper being 120.
        //Changing these values allows for control over the temperature distribution established in the extruder
        const double Nozzle_Calibration_Constant = 1.0;
        const double Barrel_Thermocouple_Calibration_Constant = 1.0;
        const double Hopper_Thermocouple_Calibration_Constant = 1.2;

        //Tolerance constants for the regulation of temperatures
        const double upper_tolerance = 0.0001;
        const double lower_tolerance = 0.01;


//------------------------------DECLARE GLOBAL VARIABLES--------------------------
//This section is before any of the functioning code and will declare all of the pins, variables, etc.

//-----------------------------------Heat Related---------------------------------
    //------Band Heaters-------------

    //----------------Band Heater Code Global Variables---------------------
        double Nozzle_Reg_Temp;
        double Barrel_Reg_Temp;
        double Hopper_Reg_Temp;

        //Declare Pins for band heater relays
        #define Band_Heater_1   34
        #define Band_Heater_2   35
```

```
#define Band_Heater_3   36


//Introduce the memory variable for temperature control
int TcaseN;
int TcaseB;
int TcaseH;


//-----Set Point Temperature----


//initialize the set point temperature variable
double Set_Point_Temp = 0;


//-----Temperature Sensing------


//Temp Sensing Logic for 3X Max31855

//Input 3.3V, use a 100 micro-farad capacitor at the connection point for the thermocouple to the MAX31855 (connecting both legs of thermocouple)
//If readings are sporadic the capacitor is the first thing that should be investigated

//Import necessary libraries for temp sensing (if you receive an error when compiling this code ensure you have these libraries installed!)
#include <SPI.h>
#include "Adafruit_MAX31855.h"
```

```
//define pins for temp sensing

//Max31855 Number 1, Nozzle
#define MAXDO_Nozzle   27
#define MAXCS_Nozzle   26
#define MAXCLK_Nozzle  25

//Max31855 Number 2, Barrel
#define MAXDO_Barrel   30
#define MAXCS_Barrel   29
#define MAXCLK_Barrel  28

//Max31855 Number 3, Hopper
#define MAXDO_Hopper   33
#define MAXCS_Hopper   32
#define MAXCLK_Hopper  31

//Initialize Max31855's

//Nozzle
Adafruit_MAX31855 nozzle_thermocouple(MAXCLK_Nozzle, MAXCS_Nozzle,
MAXDO_Nozzle);

//Barrel
Adafruit_MAX31855 barrel_thermocouple(MAXCLK_Barrel, MAXCS_Barrel,
MAXDO_Barrel);

//Hopper
```

```
Adafruit_MAX31855 hopper_thermocouple(MAXCLK_Hopper, MAXCS_Hopper,
MAXDO_Hopper);


//Variables to be referenced

//Nozzle
//number of samples to consider when computing average temperature for Nozzle
const int sample_size_nozzle=3;
//row vector in which temperature data will be stored for Nozzle
double store_temperature_nozzle_C[sample_size_nozzle]={0};
//average temperature in C for Nozzle
double average_temperature_nozzle_C=0;
//Nozzle temperature variables in C and F
double nozzle_temperature_C=0;
double nozzle_temperature_F=0;

//Barrel
//number of samples to consider when computing average temperature for Barrel
const int sample_size_barrel=3;
//row vector in which temperature data will be stored for Barrel
double store_temperature_barrel_C[sample_size_barrel]={0};
//average temperature in C for Barrel
double average_temperature_barrel_C=0;
//Barrel temperature variables in C and F
double barrel_temperature_C=0;
double barrel_temperature_F=0;

//Hopper
```

```
    //number of samples to consider when computing average temperature for Hopper
    const int sample_size_hopper=3;
    //row vector in which temperature data will be stored for Hopper
    double store_temperature_hopper_C[sample_size_hopper]={0};
    //average temperature in C for Hopper
    double average_temperature_hopper_C=0;
    //Hopper temperature variables in C and F
    double hopper_temperature_C=0;
    double hopper_temperature_F=0;
```

```
//-----------Set up variables for push button on band heaters-----------
const int band_heater_on_off_pin = 2;
int band_heater_on_off_status = 0;

//-----------Initialize pins and variables for potentiometer readings
const byte set_point_temp_pot_analog_pin = 0;
double set_point_temp_pot_analog_reading = 0;


//-----------Define LED Light pins
const int hot_LED_light_pin = 3;

//Set up variables for making analogRead non-blocking
bool working;
```

```
//-----------------Begin LCD Setup----------------
#include <LiquidCrystal.h>
const int lCd4 = 8, lCd6 = 9, lCd11 = 6, lCd12 = 5, lCd13 = 7, lCd14 = 4;
LiquidCrystal lcd(lCd4, lCd6, lCd11, lCd12, lCd13, lCd14);
//-----------------End LCD Setup------------------

//-----------------Begin Formatting Variables-----
//this will store the analog reading as a string (after rounding)
String formatted_Set_Point_Temp;
String formatted_Nozzle_Temp;
String formatted_Barrel_Temp;
String formatted_Hopper_Temp;

//initialize string variables that will contain formatted information to be printed to each line of the LCD
String line_1;
String line_2;
//-----------------End Formatting Variables-----

//---------------------------------Begin Time Keeping Variables (continuously updating the LCD causes a "blur" affect)---------
double time_1=0;
double elapsed_time;
//---------------------------------End Time Keeping Variables---------


void setup() {
```

```
Serial.begin(9600);
pinMode(LED_BUILTIN, OUTPUT);

//Declare pinMode for on/off pin
pinMode(band_heater_on_off_pin,INPUT_PULLUP);

//Declare pinMode for LED light
pinMode(hot_LED_light_pin, OUTPUT);

//Declare pinMode for band heater relays
pinMode(Band_Heater_1,OUTPUT);
pinMode(Band_Heater_2,OUTPUT);
pinMode(Band_Heater_3,OUTPUT);

//Automatically write high (off) for safety
digitalWrite(Band_Heater_1, HIGH);
digitalWrite(Band_Heater_2, HIGH);
digitalWrite(Band_Heater_3, HIGH);

//Weird stuff for making analogRead non-blocking
ADCSRA = bit (ADEN);
ADCSRA |= bit (ADPS0) | bit (ADPS1) | bit (ADPS2);
ADMUX = bit (REFS0) | (set_point_temp_pot_analog_pin & 0x07);

//initalizes lcd, in this case we have a 16 character, 2 row model
lcd.begin(16, 2);
```

```
}

void loop() {



  Temperature_Sensing();

  //read push button status for turning on and off band heaters
  band_heater_on_off_status = digitalRead(band_heater_on_off_pin);

//-----------------------------Begin Potentiometer Reading Logic----------------------
  if(!working)
  {
    bitSet (ADCSRA, ADSC);
    working = true;
  }

  if(bit_is_clear(ADCSRA,ADSC))
  {
  double value = ADC;
  working = false;
  //Serial.println(aug_Pot_Analog_Reading);
  set_point_temp_pot_analog_reading = value;
  //Serial.print("Set point potentiometer reading: ");
  //Serial.println(set_point_temp_pot_analog_reading);
  }
//-----------------------------End Potentiometer Reading Logic-------------------------
```

```
//Convert analog reading to set point temperature
Set_Point_Temp = set_point_temp_pot_analog_reading * max_set_temp/1023;


//------------------------------Begin Writing Set point temp and all temperature readings to LCD------------------------
      //Need to format reading for display, rounds reading to desired decimal place and converts to string
      formatted_Set_Point_Temp=String(Set_Point_Temp,1);
      formatted_Nozzle_Temp=String(average_temperature_nozzle_C,1);
      formatted_Barrel_Temp=String(average_temperature_barrel_C,1);
      formatted_Hopper_Temp=String(average_temperature_hopper_C,0);


      //format the entirety of the first line to be printed
      line_1="Set Temp:"+formatted_Set_Point_Temp;


      //format the entirety of the second line to be printed (UNCOMMENT AND COMMENT WHAT IS DESIRED AT THE TIME, only one allowed at a time)
      //line_2="Noz Temp:"+formatted_Nozzle_Temp;
      //line_2="Bar Temp:"+formatted_Barrel_Temp;
      //line_2="Hop Temp:"+formatted_Hopper_Temp;
      line_2=formatted_Hopper_Temp+" "+formatted_Barrel_Temp+" "+formatted_Nozzle_Temp;



      //check time elapsed between last print to LCD
      elapsed_time=millis()-time_1;


       if(elapsed_time>1000)
         {
```

```
//print to lcd
//set cursor at the beginning of the first line
lcd.setCursor(0,0);

//print the first line to the lcd (noting that the first line may or may not be 16 characters)
lcd.print(line_1);

//set the cursor to the end of the line just printed
lcd.setCursor(line_1.length(),0);
//print null characters from the end of the first line just printed through the rest of line 1 on the LCD (noting that if you exceed the 16 char limit nothing bad happens)
//this is necessary because if a value begins at 0.0, then changes order of magnitude, the end digits just hang around
//when the value is incremented back to being less than 1000 (same idea with 100), any time the order of
//magnitude of the value is changing, the end digits will stick around. so even though the analog reading is rounded to perhaps
//150.0, if there was ever a 1000+ value it will be displayed as 150.00.
lcd.print("                ");

//set cursor to beginning of second line
lcd.setCursor(0,1);

//print the second line to the lcd (noting that the second line may or may not be 16 characters)
lcd.print(line_2);

//set the cursor to the end of the line just printed
lcd.setCursor(line_2.length(),1);
lcd.print("                ");
```

```
        time_1=millis();
        }

//-----------------------------End Writing Set point temp and all temperature readings to LCD-----------------------


//-----------Call band heater function if button is pushed   (with these buttons, 0 is pushed, 1 is off), write band heater relays to off if not pushed-------------------------
  if(band_heater_on_off_status == 0)
  {
  Band_Heaters_mk2();
  }

  if(band_heater_on_off_status == 1)
  {
  digitalWrite(Band_Heater_1, HIGH);
  digitalWrite(Band_Heater_2, HIGH);
  digitalWrite(Band_Heater_3, HIGH);
  }

}
```

Band_Heaters_mk2

```
//-----------------------------------ADD TO GLOBAL VARIABLES IN MAIN-----------------------------

void Band_Heaters_mk2() {



//Set regulation temperatures based on calibration constants
  Nozzle_Reg_Temp = Set_Point_Temp * Nozzle_Calibration_Constant;
  Barrel_Reg_Temp = Set_Point_Temp * Barrel_Thermocouple_Calibration_Constant;
  Hopper_Reg_Temp = Set_Point_Temp * Hopper_Thermocouple_Calibration_Constant;



//Regulating Nozzle and Barrel band heaters based on Barrel thermocouple value due to extra cooling at exit

  if(average_temperature_barrel_C >= Barrel_Reg_Temp*(1+upper_tolerance)){
    TcaseB = 1;
  }

  if(average_temperature_barrel_C <= Barrel_Reg_Temp*(1-lower_tolerance)){
    TcaseB = 0;
  }

  if(TcaseB == 0){
  digitalWrite(Band_Heater_2, LOW);
  digitalWrite(Band_Heater_1, LOW);
  }
```

```
  if(TcaseB == 1){
  digitalWrite(Band_Heater_2, HIGH);
  digitalWrite(Band_Heater_1, HIGH);
  }

//Regulating Hopper band heater based on Hopper thermocouple value

  if(average_temperature_hopper_C >= Hopper_Reg_Temp*(1+upper_tolerance)){
    TcaseH = 1;
  }

  if(average_temperature_hopper_C <= Hopper_Reg_Temp*(1-lower_tolerance)){
    TcaseH = 0;
  }

  if(TcaseH == 0){
  digitalWrite(Band_Heater_3, LOW);
  }

  if(TcaseH == 1){
  digitalWrite(Band_Heater_3, HIGH);
  }

}
```

Temperature_Sensing

```
void Temperature_Sensing(){

//---------------------------------TEMP SENSING LOGIC BEGIN-------------------------------------

//update/read temperature for nozzle temperature
nozzle_temperature_C=nozzle_thermocouple.readCelsius();
//nozzle_temperature_F=nozzle_thermocouple.readFahrenheit();

//update/read temperature for barrel temperature
barrel_temperature_C=barrel_thermocouple.readCelsius();
//barrel_temperature_F=barrel_thermocouple.readFahrenheit();


//update/read temperature for hopper temperature
hopper_temperature_C=hopper_thermocouple.readCelsius();
//hopper_temperature_F=hopper_thermocouple.readFahrenheit();


//----------------------------BEGIN ERROR HANDLING--------------------------------
//handle 0 C and nan (nothing hooked up) error (set current temp to a high value, ensuring band heater isn't energized)

if (nozzle_temperature_C==0 || true==isnan(nozzle_temperature_C)){
  nozzle_temperature_F=10000;
  nozzle_temperature_C=10000;
```

```
  //average_temperature_nozzle_C=10000;

}

if (barrel_temperature_C==0 || true==isnan(barrel_temperature_C)){
  barrel_temperature_F=10000;
  barrel_temperature_C=10000;
  //average_temperature_barrel_C=10000;
}

if (hopper_temperature_C==0 || true==isnan(hopper_temperature_C)){
 hopper_temperature_F=10000;
 hopper_temperature_C=10000;
 //average_temperature_hopper_C=10000;
}

//----------------------------END ERROR HANDLING--------------------------------


//-----------------------------------BEGIN NOZZLE LOGIC-----------------------------------

//only want to store the value just read for nozzle temperature if it isn't faulty (nan)
if(false==isnan(nozzle_temperature_C) && nozzle_temperature_C<10000){

//"shifting" the array left for stored nozzle temperatures, so oldest value gets rejected
  for (int i=1; i<=sample_size_nozzle-1; i++) {
  store_temperature_nozzle_C[i-1]=store_temperature_nozzle_C[i];
  }
```

```
//newest nozzle temperature reading takes the last index in the array
store_temperature_nozzle_C[sample_size_nozzle-1]=nozzle_temperature_C;

//compute the average value of the nozzle temperature
double sum=0;
//add each element stored in the nozzle temperature array
for (int i=0; i<=sample_size_nozzle-1; i++){
sum=sum+store_temperature_nozzle_C[i];
}
average_temperature_nozzle_C=sum/sample_size_nozzle;
}

//-------------------------------------END NOZZLE LOGIC-------------------------------------

//-------------------------------------BEGIN BARREL LOGIC-------------------------------------

//only want to store the value just read for barrel temperature if it isn't faulty (nan)
if(false==isnan(barrel_temperature_C) && barrel_temperature_C<10000){

//"shifting" the array left for stored barrel temperatures, so oldest value gets rejected
  for (int i=1; i<=sample_size_barrel-1; i++) {
  store_temperature_barrel_C[i-1]=store_temperature_barrel_C[i];
  }

//newest barrel temperature reading takes the last index in the array
store_temperature_barrel_C[sample_size_barrel-1]=barrel_temperature_C;
```

```
//compute the average value of the barrel temperature
double sum=0;
//add each element stored in the barrel temperature array
for (int i=0; i<=sample_size_barrel-1; i++){
sum=sum+store_temperature_barrel_C[i];
}
average_temperature_barrel_C=sum/sample_size_barrel;
}

//-------------------------------------END BARREL LOGIC-----------------------------------

//-------------------------------------BEGIN HOPPER LOGIC-----------------------------------

//only want to store the value just read for hopper temperature if it isn't faulty (nan)
if(false==isnan(hopper_temperature_C) && hopper_temperature_C<10000){

//"shifting" the array left for stored hopper temperatures, so oldest value gets rejected
  for (int i=1; i<=sample_size_hopper-1; i++) {
  store_temperature_hopper_C[i-1]=store_temperature_hopper_C[i];
  }

//newest hopper temperature reading takes the last index in the array
store_temperature_hopper_C[sample_size_hopper-1]=hopper_temperature_C;

//compute the average value of the hopper temperature
double sum=0;
//add each element stored in the hopper temperature array
for (int i=0; i<=sample_size_hopper-1; i++){
```

```
sum=sum+store_temperature_hopper_C[i];
}
average_temperature_hopper_C=sum/sample_size_hopper;
}

//-----------------------------------END HOPPER LOGIC-----------------------------------

//print output to Serial Monitor if you desire (nice for troubleshooting and confirming reasonable values)

/*
//Nozzle
//Serial.println("The current Nozzle temperature in Fahrenheit is:");
//Serial.println(nozzle_temperature_F);

Serial.println("The current Nozzle temperature in Celsius is:");
Serial.println(nozzle_temperature_C);

Serial.println("The average Nozzle temperature in Celsius is:");
Serial.println(average_temperature_nozzle_C);

//Barrel
//Serial.println("The current Barrel temperature in Fahrenheit is:");
//Serial.println(barrel_temperature_F);

Serial.println("The current Barrel temperature in Celsius is:");
Serial.println(barrel_temperature_C);

Serial.println("The average Barrel temperature in Celsius is:");
```

```
Serial.println(average_temperature_barrel_C);

//Hopper
//Serial.println("The current Hopper temperature in Fahrenheit is:");
//Serial.println(hopper_temperature_F);

Serial.println("The current Hopper temperature in Celsius is:");
Serial.println(hopper_temperature_C);

Serial.println("The average Hopper temperature in Celsius is:");
Serial.println(average_temperature_hopper_C);
*/



//----------------------------------TEMP SENSING LOGIC END---------------------------------------


//CODE FOR BLINKING BUILT_IN_LED WHEN a thermocouple reading error exists
if(hopper_temperature_C==10000 || barrel_temperature_C==10000 ||
nozzle_temperature_C==10000){
digitalWrite(LED_BUILTIN, HIGH);
}

else{
  digitalWrite(LED_BUILTIN, LOW);
}

//CODE FOR MAKING LED LIGHT ON WHEN NOZZLE TEMP ABOVE 50 C
```

```
if(average_temperature_nozzle_C>=40){
  digitalWrite(hot_LED_light_pin, HIGH);
}

else{
  digitalWrite(hot_LED_light_pin,LOW);
}

//choose delay between readings (applicable to all)
//delay(500);




}
```