
**ECSE 2920: Design Methodology**
**Technical Specification and Description Report**
**Group 7**

# Table of Contents

# 1. Objective:

## Introduction

The primary objectives of our group project were centered around ambitious goals, innovative product creation, and comprehensive skill development. These objectives were not only aimed at fulfilling academic requirements but also at ensuring personal and professional growth for each team member. Below are the detailed objectives that guided our project from inception to completion.

## Complete the Entire Project and Achieve Checkpoint A-

One of our foremost objectives was to complete the entire project, including achieving the ambitious goal of reaching Checkpoint A-. This checkpoint represented a significant milestone that required dedication, coordination, and the effective utilization of our collective skills. Achieving this milestone was crucial for demonstrating our commitment to the project and our ability to meet high standards of performance. All group members agreed that our group would at least attempt to pass checkpoint A-.

## Create a Product to be Proud Of

Beyond meeting academic requirements, we aimed to create a product that we could take pride in. This objective pushed us to go beyond the basics and strive for excellence in every aspect of the project. We wanted to ensure that the final product was not just a fulfillment of the requirements but something that reflected our creativity, and hard work.

## Learn Both Soft and Hard Skills

An essential objective of our project was the holistic development of each team member through the acquisition of both soft and hard skills. Soft skills included learning how to work effectively as a team, communicate efficiently, and manage time and resources. Hard skills encompassed technical abilities such as programming in Python, circuit design, and understanding the intricacies of hardware and software integration. This dual focus on skill development was aimed at enhancing our professional competencies and preparing us for future challenges in our careers.

**Documentation and Knowledge Transfer**

Ensuring comprehensive documentation and knowledge transfer was another critical objective. This involved creating detailed records of our design processes, decisions, and the rationale behind them. The goal was to produce a body of work that could be referenced by future teams or individuals interested in our project, thereby facilitating continuity and further innovation.

**Key Roles for Each Group Member**

Neel Desai played a crucial role in the hardware aspects of the project. He worked on the creation and development of the Printed Circuit Board (PCB). He was not only pivotal in assembling the hardware but also in debugging and troubleshooting issues that arose during the project's progression, ensuring smooth operational functionality. Tae Lee contributed significantly to the hardware team by updating the project's schematics and reorganizing the circuit wiring. His efforts in redesigning the circuit layout enhanced the clarity and efficiency of the hardware setup, making it more accessible and easier to manage for the team. Daniel Poorak was instrumental in the ideation and simulation phases of the hardware development. He was responsible for conceptualizing the hardware setup and validating these concepts through simulations. Additionally, Daniel kept the team aligned with project deliverables, ensuring that they remained on track with their timelines and expectations. On the software side, Nathan Park developed critical code that improved the project's functionality. He innovated a solution to bypass the hex-inverter by altering the logic within the code itself and implemented threading to enhance the motor's performance, resulting in smoother operation. Umair Irshad was a key player in software development, working alongside Nathan to refine the code. He also contributed to the hardware discussions, offering alternative perspectives that often solved problems the hardware team could not figure out.

## 2. Constraints:

### Constraints of the Group Project

Throughout the course of our group project, several constraints have significantly influenced our approach, planning, and execution. These constraints, inherent to the nature of the project and the environment in which it was developed, have shaped our decision-making processes and outcomes. Below, we outline the key constraints that our group encountered during the project lifecycle.

### Time Constraints

Our project was subject to strict deadlines, with each deliverable scheduled on a weekly basis. The time constraint was a critical factor that required efficient time management and prioritization of tasks to ensure timely completion of each phase of the project. Balancing the project work with academic and personal commitments posed a challenge, necessitating a structured approach to task allocation and deadline adherence. Group members were held accountable to their respective share of the group's overall work. Staying on track or ahead of time was crucial to the overall performance of the team as the sheer workload of upcoming deliverables would have made catching up nearly impossible.

### Scope Constraints

The scope of our project was defined at the outset, with specific deliverables and outcomes expected at each stage. However, the scope was subject to adjustments based on feedback, technical feasibility, and resource availability. Managing scope, particularly when integrating new features or making significant changes to the project plan, required careful consideration to avoid impacting the project timeline and resources. A deep understanding of what was required of us weeks in advance was crucial to being able to set a good platform and build our product as a group collectively in an efficient way.

### Technical Constraints

The technical complexity of the project, including the design and implementation of hardware and software components, presented significant constraints. Our group's varying levels of technical expertise and the learning curve associated with new technologies impacted our progress. Technical constraints also included limitations of the hardware and software used, which affected the project's design and functionality. Most of the time was spent researching and learning how to perform the tasks asked of us before we got to action and started working.

## Resource Constraints

Resource constraints encompassed the availability of hardware components, software tools, and human resources. Delays in acquiring necessary components or access to specific software tools impacted our project timeline. Additionally, the distribution of workload among group members, considering individual skills and availability, was a critical factor in managing resource constraints. Resource constraints influenced our choices regarding materials, technologies, and external services. Efficient resource allocation was essential to stay within budget while achieving the desired project outcomes. A prime example of this was the allocation of GPIO pins which will be discussed later in the document.

## Quality Constraints

Maintaining the quality of the project deliverables while adhering to time, scope, and budget constraints was a constant challenge. Ensuring the reliability and performance of the hardware and software components required thorough testing and validation, which had to be balanced with the project's other constraints. Getting the product to work is one thing and getting it to work well is another. Our group made it a priority to create a product that not only passed the guidelines but exceeded them and created an enjoyable user experience.

## Risk Constraints

Risk management was an integral part of our project, with various risks identified and mitigated throughout the project lifecycle. Risk constraints included technical risks, project management risks, and external risks, each requiring specific strategies to minimize their impact on the project. The biggest risk constraint faced was not damaging our hardware due to improper wiring. Short-circuiting an important component would have been a huge set back and halted our progress until the component was replaced.

## Conclusion

In conclusion, navigating these constraints required a flexible and adaptive approach, with continuous monitoring and adjustment of the project plan. Effective communication, collaboration, and problem-solving skills were crucial in overcoming these constraints and achieving the project objectives.

# 3. Technical Designs

## 3.1 Driving the program (Driver.py):

```
75  def x_press():
76      if Properties.current_state == States.menu_state:
77          Menu_Mode.menu_select()
78      elif Properties.current_state == States.etch_state:
79          Etch_Mode.x_press()
80      elif Properties.current_state == States.math_state:
81          Math_Mode.x_press()
82      elif Properties.current_state == States.calibrate_state:
83          Calibrate_Mode.cal_select()
84      elif Properties.current_state == States.g_state:
85          G_Code.x_press()
```

***Figure 3.1.0: x_press() function in Driver.py***

The user controls everything through two rotary encoders that are connected to the GPIO pins of the Raspberry Pi. When the rotary encoders are pressed, the program simply reads when the pins of the encoder switch are 0. Further calculation was needed to determine whether it is a short press, long press, or simultaneous press. When the rotary encoders are rotated, the Rotary package in the pigpio_encoder.rotary library is used to create a callback thread that calls a function depending on which is rotated and in which direction. This setup for the rotary encoders takes place in Driver.py, where all the functions that respond to user input are, such as y_rotary_down() and x_press() (Figure 3.1.0), for example. Each of these functions call the corresponding control function within the class that represents the current mode — such as in Figure 3.1.0, x_press() would call different functions depending on the current mode.

```
26  def x_rotary_up(counter):
27      if Properties.current_state == States.menu_state:
28          Menu_Mode.menu_cycle_prev()
29      elif Properties.current_state == States.etch_state:
30          Etch_Mode.y_rotary_up()
31      elif Properties.current_state == States.math_state:
32          Math_Mode.x_rotary_up()
33      elif Properties.current_state == States.calibrate_state:
34          Calibrate_Mode.cal_cycle_prev()
35      elif Properties.current_state == States.g_state:
36          G_Code.x_rotary_up()
```

***Figure 3.1.1: x_rotary_up() function in Driver.py***

For example, when in Math Mode, calling x_rotary_up() within Driver results in a call for Math_Mode.x_rotary_up() (Figure 3.1.1, line 31 and 32), and using static function calls makes the process much easier. This pattern is the same for all user input functions and every mode. This way, the Driver class controls all function calls within the other classes, while the classes themselves control their functionality. The various classes allow the system to be organized and structured in a way that allows growth.

## 3.2 Class Setup and Organization:

The organization of the system makes use of a number of classes that allow information to be accessed by every mode in the program.

```
4   class Properties:
5       # coordinate position of gantry
6       pos_x = 0
7       pos_y = 0
8       # whether the pen is down
9       pen_down = False
10      # the physical limits of the paper
11      max_y = 240
12      max_x = 180
13      # the current state (uses States class)
14      current_state = States.menu_state
15      delay = 0.00125
16      power = 0.0
```

***Figure 3.2.0: Properties class in Properties.py***

The most utilized class for this function is Properties.py, shown in Figure 3.2.0, where data that tracks the position, pen up, the physical limits of the plotter, etc. is stored. Through the properties class, other classes such as Etch-a-Sketch mode and Math mode can both access this information.

```
57    # Standard Step Functions (this is when the slope is not changing)
58      def backwardStep(motor_type,factor):
59        for i in range(4):
60          setStepper(steps[i][0], steps[i][1], steps[i][2], steps[i][3], motor_type, factor)
61          Motor.incrementPosition(motor_type, -step_size/4)
62
63
64      def forwardStep(motor_type, factor):
65        for i in range(3,-1,-1):
66          setStepper(steps[i][0], steps[i][1], steps[i][2], steps[i][3], motor_type, factor)
67          Motor.incrementPosition(motor_type, step_size/4)
```

***Figure 3.2.1: Step functions in Motor.py***

```
# lists that represent the pins of the motor, ordered for implementation in loops
y_motor = [GPIO_Pins.P1_A1, GPIO_Pins.P1_A2, GPIO_Pins.P1_B1, GPIO_Pins.P1_B2]
x_motor = [GPIO_Pins.P2_A1, GPIO_Pins.P2_A2, GPIO_Pins.P2_B1, GPIO_Pins.P2_B2]
z_motor = [GPIO_Pins.P3_A1, GPIO_Pins.P3_A2, GPIO_Pins.P3_B1, GPIO_Pins.P3_B2]
```

***Figure 3.2.2: Motor pin lists in Motor.py***

Additionally, Motor.py is the main class that operates the motors and increments the position variables that are stored in the properties class (Figure 3.2.1, lines 61 and 67) . Motor.py implements forward and backward step functions (Figure 3.2.1) that can be used for every motor and at any speed. Any modes that require motor use will directly call this class. To choose which motor to move, this class also holds the list of pin information for each motor: Motor.x_motor, Motor.y_motor, and Motor.z_motor (Figure 3.2.2). This way, any class can easily call, for example, Motor.forwardStep(Motor.x_motor, 2), which moves x motor for one step at half speed. The step functions utilize the pin information of the motors and the variable steps, which is the order in which the motor pins need to be set to HIGH and LOW, and passes this data into the setStepper() function (Figure 3.2.1, lines 60 and 66), which writes the information from the steps variable onto the pins from the motor sequentially at a delay.

```
2  class States:
3      menu_state = "menu"
4      calibrate_state = "calibrate"
5      plotter_mode = "plotter"
6      stop_state = "stop"
7      reset_state = "reset"
8
9      # plotter submenu
10     etch_state = "etch"
11     math_state = "math"
12     g_state = "gcode"
```

***Figure 3.2.3: States class in States.py***

Furthermore, States.py allows any class to access the signatures for each state. For instance, Driver.py can check if Properties.current_state == States.menu_state and Menu_Mode.py can set Properties.current_state = States.etch_state (Figure 3.2.0 and Figure 3.2.3).

```
7  class Hardware:
8      pi = pigpio.pi()
9      lcd = rgb1602.RGB1602(16,2)
10
11     # sets a HIGH output to the h_enable pin, which is connected to the enable pins of both h-bridges
12     def h_enable():
13         Hardware.pi.write(GPIO_Pins.h_enable, 1)
14
15      # sets a LOW output to the h_enable pin, which is connected to the enable pins of both h-bridges
16     def h_disable():
17         Hardware.pi.write(GPIO_Pins.h_enable, 0)
```

***Figure 3.2.4: Hardware class (without setup() function) in Hardware.py***

Another class with similar functionality is Hardware.py. This class's purpose is to simply force every class to use the same object of the pi class and LCD class. A new object for pi and LCD are created on boot (Figure 3.2.4, lines 8 and 9), and can be used through static references to the class. Now, any time that these objects need to be used in another class, they can be accessed through Hardware.pi and Hardware.LCD. This class also deals with enabling and disabling the GPIO pin that runs to the enable input for all three H-bridges. Whenever the motors need to be moved, Hardware.h_enable() is called by the current class, and Hardware.h_disable() is called after the operation is done (Figure 3.2.4). The hardware class also holds a setup function that overrides the default state for the needed GPIO pins.

```
2  class GPIO_Pins:
3      # rotary encoder pins
4      sw2 = 16
5      clk2 = 20
6      dt2 = 21
7
8      sw1 = 13
9      clk1 = 26
10     dt1 = 7
11
12     sw_y = 4
13     sw_x = 14
14
15     h_enable = 9
```

***Figure 3.2.5: GPIO_Pins class (stepper motor pins not included) in GPIO_Pins.py***

Finally, the GPIO_Pins.py class provides a universal place for all GPIO pin numbers to be stored and accessed (Figure 3.2.5). This allows other classes to not use "magic numbers" or their own instances of GPIO data, meaning that GPIO pin locations can be easily changed for every class. This class is referenced in Figure 3.2.2 and Figure 3.2.4 as well.

## 3.3 UI Interface:

The user interface has been implemented in a consistent way throughout the project, using arbitrary variables to store the key/signature of a certain state. For instance, there is a state that defines when the UI would be hovering over the “Plotter Mode” option, named States.plotter_mode (Figure 3.2.3, line 5), that stores an arbitrary string, unique to this variable. Every state is defined in this manner, and the current hovering state can be assigned to a local variable hover_state, and the current state can be assigned to variable Properties.current_state (Figure 3.2.0, line 14).

```
107     def menu_select():
108       global hover_state
109       if hover_state == States.plotter_mode:
110         hover_state = States.etch_state
111         helper_print()
112       else:
113         switch_state(hover_state)
```

***Figure 3.3.0: menu_select() function in Menu_Mode.py***

```
40  def switch_state(target_state):
41    global hover_state
42    Properties.current_state = target_state
43    # initializations for each mode
44    if Properties.current_state == States.etch_state:
45      Etch_Mode.initialize()
46    elif Properties.current_state == States.menu_state:
47      Menu_Mode.initialize()
48    elif Properties.current_state == States.math_state:
49      Math_Mode.initialize()
50    elif Properties.current_state == States.calibrate_state:
51      Calibrate_Mode.initialize()
52    elif Properties.current_state == States.g_state:
53      G_Code.initialize()
```

***Figure 3.3.1: switch_state() function in Menu_Mode.py***

For example, the current state could be the main menu state, and the hovering variable could be calibration mode; when the user selects calibration mode, calling menu_select() (Figure 3.1.0), switch_state(hover_state) is called (Figure 3.3.0), which changes the current state to calibration mode state, and the hovering variable can be assigned to auto calibration mode (Figure 3.3.1).

```
87      # cycles back through the ui, called by Driver when the x-rotary encoder is turned down
88      def menu_cycle_prev():
89        global hover_state
90        if hover_state == States.plotter_mode:
91          hover_state = States.reset_state
92        elif hover_state ==  States.calibrate_state:
93          hover_state = States.plotter_mode
94        elif hover_state == States.stop_state:
95          hover_state = States.calibrate_state
96        elif hover_state == States.reset_state:
97          hover_state = States.stop_state
98        elif hover_state == States.etch_state:
99          hover_state = States.g_state
100       elif hover_state == States.math_state:
101         hover_state = States.etch_state
102       elif hover_state == States.g_state:
103         hover_state = States.math_state
104       helper_print()
```

***Figure 3.3.2: menu_cycle_prev() function in Menu_Mode.py***

Furthermore, various state variables can be linked when the user rotates the rotary encoders — when the user rotates right (Figure 3.1,1), the hovering variable changes depending on the current hovering mode within the function menu_cycle_prev() (Figure 3.3.2).

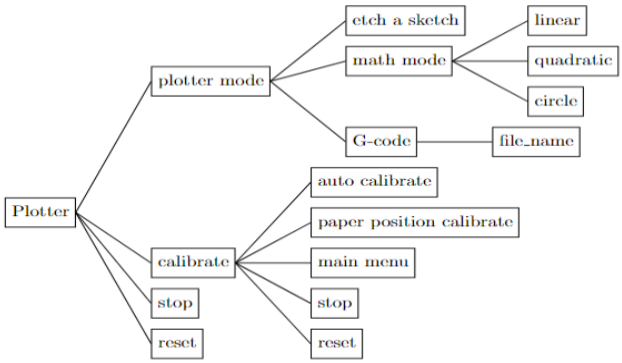


***Figure 3.3.3: User Interface Setup***

The current mode of the program determines which class deals with the current UI. For instance, on boot, the main menu class deals with the UI since the control functions in Driver.py are linked in to the control functions in the Menu Mode class when in the main menu (Figure 3.1.0, for example). The same is true for every other mode, since Driver.py's functions only call the current state's functions. Using the system as described before, such as the switch_state() function (Figure 3.3.1) and the organization of the driver controls (Figure 3.1.0 and Figure 3.1.1), every mode's submenu was linked according to the User Interface Setup in Figure 3.3.3.

The rotary functions in each class update the LCD screen on each call. See Appendix section A4 for full LCD datasheet.

## 3.4 Etch-a-Sketch Mode:

The functionality of Etch-A-Sketch Mode lies within Etch_Mode.py. When the user selects etch-a-sketch mode, the current state switches to etch mode, meaning that the functions in Driver.py now directly call the corresponding functions in Etch_Mode.py. Additionally, Etch_Mode.initialize() will be called at the time of switching states so that the LCD screen will update automatically, and any initializations, such as wait mode or speed, can be made.

```
96    def x_rotary_up():
97      global rot_per_turn, pos_x, hold_all_input
98       # if in wait mode, do nothing
99      if hold_all_input:
100       return
101
102     # run rot_per_turn times every turn
103     for i in range(rot_per_turn):
104       # if the x-limit switch is pressed, set coordinate and do nothing
105       if Hardware.pi.read(GPIO_Pins.sw_x) == 1:
106         Properties.pos_x = -Properties.max_x/2
107         return
108       # otherwise, execute step
109       Motor.backwardStep(Motor.x_motor, 1)
```

***Figure 3.4.0: x_rotary_up() function in Etch_Mode.py***

When at slow speed, any rotary encoder rotation function directly calls one step in the motor class. When at fast speed, any rotary encoder rotation function calls 8 steps in the motor class. Looking at Figure 3.4.0, this is the case since Motor.backwardStep(Motor.x_motor,1) is on a loop running rot_per_turn times, where rot_per_turn represents the speed (integer of 1 or 8).

```
86    def toggle_pen():
87      for i in range(steps_down): # run steps_down times
88        if (Properties.pen_down): # if the pen is already down, move up (forward) at 1/5 speed
89          Motor.forwardStep(Motor.z_motor, 5)
90        else:   # if the pen is up, move down (backward) at 1/5 speed
91          Motor.backwardStep(Motor.z_motor,5)
92          time.sleep(0.01)
93      Properties.pen_down = not Properties.pen_down # update boolean variable
```

***Figure 3.4.1: toggle_pen() function in Etch_Mode.py***

Because the position is already updated in the motor class, no changes to position are needed in Etch-A-Sketch mode. This class has its own function that toggles the pen up/down (Figure 3.4.1), which makes the z-motor step backward or forward steps_down times and updates the boolean that is stored in Properties.py. Just through connecting the rotary functions in Etch-A-Sketch mode to the motor class, Etch-A-Sketch mode is already almost fully functional.

The LCD screen is updated by calling Etch_Mode.print_lcd() whenever the information on it would have been changed, such as when the speed or pen is toggled.

```
196     def both_press():
197         global hold_all_input
198         if Properties.pen_down:
199             Etch_Mode.toggle_pen()
200         hold_all_input = not hold_all_input
201         Etch_Mode.print_lcd()
202         if hold_all_input:
203             Hardware.h_disable()
204         else:
205             Hardware.h_enable()
```

***Figure 3.4.2: both_press() function in Etch_Mode.py***

When Driver.py recognizes a simultaneous press and calls Etch_Mode.both_press() (Figure 3.4.2), the boolean hold_all_input is toggled (Figure 3.4.2, line 200) which blocks all interaction other than another simultaneous press (shown blocking input in Figure 3.4.0, lines 99 and 100). This function also updates the screen to show that the system is in wait mode (Figure 3.4.2, line 201), and disables/enables the h-bridges (Figure 3.4.2, lines 202-205).

When either of the limit switches are pressed, the position variable for that axis updates (as a mini-calibration) and doesn't allow the user to continue moving in that direction (Figure 3.4.0, lines 105-107). Assuming both axes are calibrated, if the position at any point attempts to exceed the bounds set in Properties.py, the program does not allow the motors to continue to move in that direction. Finally, in order for the user to exit the program, the Etch_Mode.y_press() function switches the state back to the main menu when the system is in wait mode.

## 3.5 Math Mode:

Math Mode is implemented within the class Math_Mode.py. Just as in Etch-A-Sketch mode, selecting the "Math Mode" option in the main menu switches the current state to math mode, redirecting Driver.py's main controls from Menu Mode to Math Mode.

```
116  def math_cycle_next():
117      global hovering, display_value
118      if hovering == linear_select:
119        hovering = quadratic_select
120      elif hovering == quadratic_select:
121        hovering = circle_select
122      elif hovering == circle_select:
123        hovering = linear_select
124      else:
125        display_value += 1
126      print_lcd()
```

***Figure 3.5.0: math_cycle_next() function in Math_Mode.py***

The submenu of math mode operates in the same way as the rest of the user interface, and the user can choose one of the three function types (linear, quadratic, circle), as shown in lines 118 to 123 of Figure 3.5.0. When selecting values for the coefficients, the program is constantly displaying a variable called display_value, and rotating the rotary encoders changes this variable and updates the LCD (Figure 3.5.0). Pressing the right rotary encoder saves the display value to the corresponding coefficient's variable, resets the display value, and updates the LCD to show the next coefficient. When at the last coefficient, it moves back to the first coefficient. Pressing the left rotary encoder does not save the display value, and it sets the display value to the previous coefficient and updates the display to show the previous coefficient. When at the first coefficient, it returns to the math mode submenu.

```
176      elif hovering == linear_selected:
177        hovering = printing_mode
178
179        # line is selected and printing, update lcd and plot
180        print_lcd()
181        Math_Mode.linear(m_linear, b_linear)
```

***Figure 3.5.1: Subsection of math_select() function in Math_Mode.py***

In order to print the specified function with the saved coefficients, the corresponding function is called after confirming that the plotter is calibrated. For example, once linear is selected and the user confirms by pressing the right rotary encoder, calling the math_select() function, the current hovering state switches to printing mode and the linear() function is called with the saved coefficients (Figure 3.5.1).

The linear function can be broken down into a few steps. Firstly, find an initial point on the boundary of the plotter. Next, find the final point on the boundary of the plotter. Finally, connect those two points by drawing a line between them with the appropriate slope. The first two steps are fairly simple, but it involves figuring out which “walls” the line intersect, which can be done simply by plugging in the maximum values into the given linear function and noting the resulting position.

```
128    def moveToPoint(target_x, target_y, factor):
129      # calculate factors:
130      xfactor = 1
131      yfactor = 1
132      dx = target_x - Properties.pos_x
133      dy = target_y - Properties.pos_y
134
135      if dx == 0 or dy == 0:
136        if dx == 0:
137          dx = 1
138        if dy == 0:
139          dy = 1
140        xfactor = 1
141        yfactor = 1
142      elif abs(dx) > abs(dy):
143        xfactor = 1
144        yfactor = abs(dx/dy)
145      elif abs(dx) < abs(dy):
146        xfactor = abs(dy/dx)
147        yfactor = 1
148
149      # start threads
150      x_thread = threading.Thread(target = Motor.moveToPosX, args = (target_x, xfactor * factor)) # x motor moves simultaneously
151      y_thread = threading.Thread(target = Motor.moveToPosY, args = (target_y, yfactor * factor)) # y motor moves simultaneously
152
153      x_thread.start()
154      y_thread.start()
155
156      x_thread.join()
157      y_thread.join()
```

***Figure 3.5.2: moveToPoint() function in Motor.py***

The function moveToPoint() is called within the linear function. This function is used for the actual movement of the motors when drawing the line. It is located within the Motor class for easier access to the step functions, but it could be located anywhere as needed. Given a target point, the function calculates the speeds that the motors need to move (shown by factors) and create the threads for both motors (Figure 3.5.2). We used threading in our design because it seemed to be the most seamless way for the motors to move independently at different speeds. Without threading, there would have most likely been clear steps in the final print, which was undesirable for us.

The factors (xfactor, yfactor) are a multiple of how slow the motors need to move in their own respective thread. For example, the function $y = 2x$ would require that the y-motor moves at twice the speed of the x-motor, so xfactor=2 and yfactor=1, where a factor of 1 is the fastest speed the motor can move. This factor calculation is done in lines 130 to 147 of Figure 3.5.2. Additionally, the literal factor parameter in line 128 of Figure 3.5.2 is used to slow the overall execution of the movement. For example, when factor==2, then the line would be drawn half as fast.

```
23  def setStepper(in1, in2, in3, in4, motor_type, factor):
24      global exec_time
25      current_time = time.time()
26      Hardware.pi.write(motor_type[0], in1)
27      Hardware.pi.write(motor_type[1], in2)
28      Hardware.pi.write(motor_type[2], in3)
29      Hardware.pi.write(motor_type[3], in4)
30      exec_time = time.time() - current_time
31      time.sleep(delay * factor + (factor - 1) * exec_time)
```

***Figure 3.5.3: setStepper() function in Motor.py***

In Figure 3.5.3, you can see the effect of the finalized factor parameter when stepping the motors. When this function is called on loop, the factor decreases the frequency of calls by multiplying by delay, which is the smallest increment between steps possible. The factor used in the moveToPoint() function (Figure 3.5.2) is passed into the threads where it is passed into the setStepper() function.

The exec_time variable is used to solve a very specific problem with threading. Because the threads run completely independently until completion, it is very likely for them to be unsynchronized. In most cases, this would mean that the motor that needed to move less finished slightly earlier than the other motor, even when the frequencies are adjusted accordingly. Our team narrowed the issue down to the execution time of the steps themselves, which weren't being accounted for previously. After breaking down a lot of theory, adding exec_time with a coefficient of factor-1 would account for the execution time of each step (Figure 3.5.3, line 31), where exec_time is constantly being recorded (Figure 3.5.3, line 30). Adding the execution time of each step synchronized the motors for future movements in threading as well.

The quadratic function can be broken down into a couple steps. Firstly, just as the linear function, find the initial point on the boundary of the quadratic. Next, rather than finding the final point, the quadratic function can just move until exceeding the boundary.

```
448         # start threads
449         x_thread = threading.Thread(target = Motor.moveWithSlopeX, args = ()) # moves x motor simultaneously
450         y_thread = threading.Thread(target = Motor.moveWithSlopeY, args = ()) # moves y motor simultaneously
451         slope_thread = threading.Thread(target = Motor.updateMovingSlopeQuad, args = ()) # updates slope / motor speeds simulatenously
452
453         x_thread.start()
454         y_thread.start()
455         slope_thread.start()
456
457         x_thread.join()
458         y_thread.join()
459         slope_thread.join()
```

***Figure 3.5.4: Subsection of quadratic() function in Math_Mode.py***

Our approach to the quadratic function requires three threads — one for the x motor, one for the y motor, and one to dynamically adjust their slopes (Figure 3.5.4). All three threads terminate once the position of the plotter reaches/exceeds the boundaries specified in Properties.py. The threads use static references to class variables for the factors, such as Motor.factorx for the x motor. This way, all three threads can read and write to these variables dynamically. This is important since the constantly-changing slope of the quadratic function requires new factors constantly. Furthermore, this method allows the speed of the motor to change even during a single step, which maximizes smoothness.

```
175     def updateMovingSlopeQuad():
176         while abs(Properties.pos_x) <= Properties.max_x/2 and abs(Properties.pos_y) <= Properties.max_y/2:
177             #Motor.moving_slope = 2 * Motor.a * Properties.pos_x + Motor.b
178             increment = 0.5
179             targety = Motor.a * (Properties.pos_x + increment)* (Properties.pos_x + increment) + Motor.b *  (Properties.pos_x + increment) + Motor.c
180             Motor.moving_slope = (targety - Properties.pos_y)/increment
181             Motor.calculateFactors()
182             time.sleep(2 * delay)
```

***Figure 3.5.5: updateMovingSlopeQuad() function in Motor.py***

These factors are calculated and updated in the slope_thread (Figure 3.5.5, line 181), where the current slope is calculated. The team originally used derivatives to calculate the slope, but this came with some issues. Mainly, since the slopes are not updated instantaneously, the movement of the plotter would always underestimate with positive quadratics and overestimate with negative quadratics. Therefore, we found that calculating the slope to the point that is a certain increment away is much more accurate, as shown in Figure 3.5.5. It is important to note that this "next point" is constantly being updated as the current position changes, meaning that it acts as a 'guide' for the direction of the plotter rather than a destination.

The circle function is done in a similar fashion to the quadratic function, as it also uses three threads with constantly changing motor speeds. However, while, in the quadratic function, the next point for slope calculation is a certain increment on the x-axis, the next point on the circle

would be a certain increment of degrees away from the current point. Therefore, the threads for the x motor and y motor behave exactly the same as the quadratic function, other than the stopping condition being that the circle has reached/exceeded 359 degrees.

```
202     def updateMovingSlopeCirc():
203       while (Motor.deg < 359):
204         if (Properties.pos_x == 0):
205           if (Properties.pos_y > 0):
206             Motor.deg = 90
207           else:
208             Motor.deg = 270
209         else:
210           Motor.deg = math.degrees(math.atan((Properties.pos_y / Properties.pos_x)))
211           if (Properties.pos_x < 0):
212             Motor.deg += 180
213           elif (Properties.pos_x > 0 and Properties.pos_y < 0):
214             Motor.deg += 360
215           if Motor.deg >= 360:
216             Motor.deg -= 360
217
218         increment_deg = 5
219         target_x = Motor.r * math.cos(math.radians(Motor.deg + increment_deg))
220         target_y = Motor.r * math.sin(math.radians(Motor.deg + increment_deg))
```

***Figure 3.5.6: Subsection of updateMovingSlopeCirc() function in Motor.py***

Part of the function being executed in the slope thread for the circle is shown in Figure 3.5.6. Firstly, the current degrees, saved to Motor.deg, is calculated (Figure 3.5.6, lines 203-216). Then, using the current degrees, radius, and degree increment, the next point for the x and y is calculated, saved to target_x and target_y. Finally, just as with the quadratic function, the corresponding factors for each motor are calculated and saved to the class variables so that the other two threads can read them. One exception to this process that had to be accounted for is at the points in which the slope would be 0/infinity, or close to 0/infinity (0, 90, 180, and 270 degrees). This would cause one of the factors to be extremely large, sometimes causing the motor to take a few seconds to complete a single step (factor approx. 20,000). This issue was solved by adding a maximum factor of 30.

## 3.6 G-Code:

G-Code Mode is implemented between three classes — G_Code.py, G_Functions.py, and G_Motor.py. G_Motor.py is mostly identical to the previous Motor class; however, because the axes are treated differently in this mode, a new class needed to be created for it. The relationship between the three classes can be thought of most simply as: G_Code.py calls G_Functions.py, which calls G_Motor.py.

G_Code.py deals with the user interface, file selection, and file parsing. The user interface is implemented the same as previously, with different string variables acting as representations for different states the UI is currently in.

```
127    # looks in current directory for files that end with g code, returns list of filenames
128    def create_list():
129      filenames = []
130      search_for = ".gcode"
131      file_list = os.listdir(cur_dir)
132      for filename in file_list:
133        if search_for in filename:
134          filenames.append(filename)
135      return filenames
```

***Figure 3.6.0: create_list() function in G_Code.py***

Regarding file selection, Figure 3.6.0 shows how the list of filenames is created. The os package was necessary for this section for its listdir() function. The create_list() function specifically adds the filenames that have ".gcode" from this list and adds them to the return variable filenames. The files can then be cycled through and selected using the same method as cycling through the user interface.

```
163    # starts execution of each line within the file, runs execute_command() for each command in file
164    def start_execution():
165      file1 = open(filename, 'r')
166      lines = file1.readlines()
167
168      for line in lines:
169        if len(line) != 0:
170          string_arguments = line.split()
171          arguments = get_values(string_arguments)
172          execute_command(arguments)
173      Hardware.h_disable()
174
175      file1.close()
176
177      return
```

***Figure 3.6.1: start_execution() function in G_Code.py***

When the user selects “Start” and “Calibrate ready?”, then the start_execution() function in Figure 3.6.1 is called using the file that is saved to the variable filename. The function reads the file and splits the contents into a list of lines (Figure 3.6.1, lines 165-166). For each line, the arguments (such as the command, position variables, and speed) are extracted using substring functions and data type conversion functions, and the command is executed (Figure 3.6.1, lines 168-172). Within this loop, each line in the file will be executed.

```
240       if (command == "G00"):
241         G_Functions.G00(x,y)
242       elif (command == "G01"):
243         G_Functions.G01(x,y,f)
244       elif (command == "G02"):
245         G_Functions.G02(x,y,i,j)
246       elif (command == "G03"):
247         G_Functions.G03(x,y,i,j)
248       elif (command == "G28"):
249         G_Functions.G28()
250       elif (command == "M02"):
251         G_Functions.M02()
252       elif (command == "M03"):
253         G_Functions.M03()
254       elif (command == "M04"):
255         G_Functions.M04()
```

***Figure 3.6.2: Subsection of execute_command() function in G_Code.py***

The execute_command function, as shown in Figure 3.6.2, calls the corresponding function in G_Functions.py, where all of the calculations are done and command routines are specified.

```
191     # moves to home position
192     def G28():
193       move_to_x(0)
194       move_to_y(0)
195       return
196     # stops program, pen up move to home
197     def M02():
198       G_Functions.M04()
199       G_Functions.G28()
200       return
201     # pen down if not down
202     def M03():
203       if not Properties.pen_down:
204         G_Motor.toggle_pen()
205       return
206     # pen up if down
207     def M04():
208       if Properties.pen_down:
209         G_Motor.toggle_pen()
210       return
```

***Figure 3.6.3: G28(), M02(), M03(), M04() functions in G_Functsions.py***

Figure 3.6.3 shows the routines done for some of the simple command functions, and it also demonstrates how G_Functions.py connects to G_Motor.py. For example, when the command M03 is called, the function simply checks the current state of the pen, and then calls the toggle_pen() function in G_Motor.py as needed.

The most complex commands are those for the linear functions (G00/G01) and arc functions (G02/G03), as they each use threading. Just as their implementation in math mode, the linear functions use two threads, and the arc functions use three threads.

```
156     def G00(x,y):
157       # line straight to point
158       G_Functions.G01(x,y,400)
159       return
160
161     def G01(x,y,f):
162       factors = calculate_factors(Properties.pos_x, Properties.pos_y, x,y)
163
164       # adjust speed
165       factors = [factor * (400 / f)*slow_speed for factor in factors]
166
167       G_Motor.factors = factors
168
169       # start threads
170       x_thread = threading.Thread(target = move_to_x, args = (x,)) # x motor moves simultaneously
171       y_thread = threading.Thread(target = move_to_y, args = (y,)) # y motor moves simultaneously
172
173       x_thread.start()
174       y_thread.start()
175
176       x_thread.join()
177       y_thread.join()
```

***Figure 3.6.4: G00() and G01() functions in G_Functions.py()***

In Figure 3.6.4, it is clear that G00 only calls G01 but at max speed (400 mm/minute). G01 is extremely similar to the moveToPoint() function in Figure 3.5.2. The only difference in calculation is the speed adjustment after calculating the factors (Figure 3.6.4, line 165). The f parameter determines how fast the line should be drawn relative to the max speed, meaning that when f==400, the factor would be multiplied by 1, making it the max speed. Additionally, the variable slow_speed is a constant variable that determines what the max speed would be. Without it, the max speed would be closer to 4000 mm/min. Other than speed, this function executes in the same fashion as math mode.

```
183       # draws arc cw
184       def G02(x,y,i,j):
185         G_Functions.draw_arc(x,y,i,j,5)
186         return
187       # draws arc ccw
188       def G03(x,y,i,j):
189         G_Functions.draw_arc(x,y,i,j,-5)
```

***Figure 3.6.5: G02() and G03() functions in G_Functions.py***

Figure 3.6.5 outlines how the two arc functions are set up. The last parameter (5 for G02 and -5 for G03) is the degree increment that is used to calculate the location of the next point for slope calculation. G02 looks 5 degrees ahead, and attempts to constantly move to that point, while G03 looks 5 degrees behind and does the same. Just as before with Math Mode, this "next point" is only a guide of where the plotter needs to move towards, not the destination of the plotter.

```
111   # checks whether the curent position and desired position is the same distance away from the designated center
112   def verify_radius(x,y,i,j):
113     r_threshold = 2
114
115     target_point = [x,y]
116     current_point = [Properties.pos_x, Properties.pos_y]
117     center_point = [Properties.pos_x + i, Properties.pos_y + j]
118
119     current_radius = math.sqrt((center_point[0] - current_point[0]) ** 2 + (center_point[1] - current_point[1]) ** 2 )
120     target_radius = math.sqrt((center_point[0] - target_point[0]) ** 2 + (center_point[1] - target_point[1]) ** 2 )
121
122     print("current radius: " + str(current_radius) + ". target radius: " + str(target_radius))
123     return (abs(current_radius - target_radius) < r_threshold)
```

***Figure 3.6.6: verify_radius() function in G_Functions.py***

The first step in the draw_arc() function is to verify whether the arc is valid by looking at the specified radii. This is done in the function verify_radius(), which calculates the distance between the current point and center, and the distance between the target point and center. If these two radii are not within a specified threshold apart, then it is not a valid arc. Through testing, we found that a safe threshold to use is 2. If the arc is not valid, then the draw_arc() function does not continue to execute.

```
223         G_Functions.current_degrees = calculate_degrees(center_point, [Properties.pos_x, Properties.pos_y])
224         target_degrees = calculate_degrees(center_point, target_point)
225         print("target degrees: " + str(target_degrees))
226 
227         # update circle values initially
228         # find next point
229 
230         next_point = find_next_point_circle(G_Functions.current_degrees + deg_increment, center_point, radius)
231 
232         factors = calculate_factors(Properties.pos_x, Properties.pos_y, next_point[0], next_point[1])
```

***Figure 3.6.7: Subsection of draw_arc() function in G_Functions.py***

After verifying the arc, the draw_arc() function prepares to start the threads by calculating the current degrees and initial next point, as shown in Figure 3.6.7. This calculation was not present in the circle function in Math Mode since the initial degrees and slope were always the same.

```
234         if radius > 20:
235           speed = 1
236         else:
237           speed = 20 / radius
238 
239         factors = [factor * speed * slow_speed for factor in factors]
```

***Figure 3.6.8: Subsection of draw_arc() function in G_Functions.py***

Another change to the arc function compared to the circle function in Math Mode is the speed adjustment. As with the linear functions, the slow_speed variable is present to cap the speed accordingly. However, unlike Math Mode, G-Code Mode often needs to draw small arcs and circles, which creates an issue with accuracy. Because the slope thread is being updated at a constant frequency, the factors for the motors won't be updated as many times during a smaller drawing time period. Therefore, as the size of the arc decreases, so does the accuracy. However, if smaller arcs are drawn slower, this problem would be fixed. The implementation of this solution is shown in Figure 3.6.8, where, as the radius decreases, the speed variable increases, which slows down the frequency of the motors (factor increases).

Other than these small changes, the method for drawing the arcs is identical to the circle function Math Mode.

## 3.7 Power Supply Design and Implementation:

### Overview

The development of a power supply capable of converting a 24V input into ±12V outputs was a critical component of our project. This section outlines the journey from simulation to the realization of the power supply, detailing the design choices, challenges encountered, and the solutions implemented to achieve a functional power supply that meets the project's requirements. The initial challenge was to design a power supply that could efficiently split a 24V input into symmetrical ±12V outputs. The design process began with simulations to explore various configurations and components that could achieve the desired output while maintaining stability under different load conditions.

### Op Amps and MOSFETs Configuration

A pivotal decision in our design was the use of operational amplifiers (Op Amps) and Metal-Oxide-Semiconductor Field-Effect Transistors (MOSFETs) to regulate the output voltages. See Appendix sections A7 and A8 for full PMOS and NMOS MOSFET datasheets. The LM741 Op Amps were configured to compare feedback voltage from the output with a reference voltage, adjusting the gate voltage of the MOSFETs to maintain the desired ±12V outputs. See Appendix section A6 for the full LM741 datasheet. This configuration was crucial for stabilizing the output voltage and maintaining the virtual ground at 0V, ensuring that the power supply could adapt to changes in load without significant deviations in output voltage. See Figure 3.7.0 for initial simulation in Multisim.

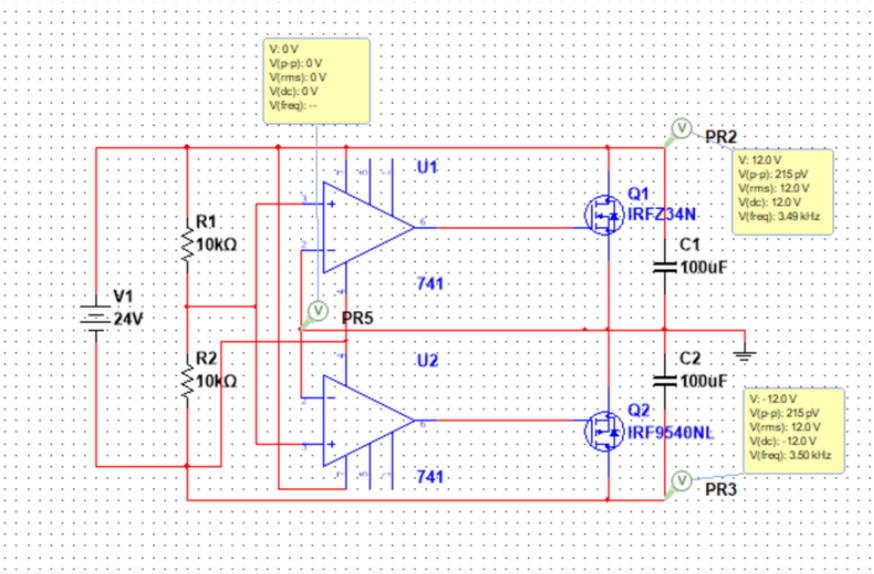


***Figure 3.7.0: Initial Power supply design with no load connected***

## Virtual Ground Concept

The concept of a virtual ground played a significant role in our design, allowing us to create a midpoint that effectively split the 24V input into positive and negative 12V outputs. The initial design utilized two op-amps to keep the virtual ground at 0V. After receiving feedback, the design was modified to only use one op-amp by using the single op-amp to drive the gates of both MOSFETs rather than two op-amps driving each gate. See Figure 3.7.1 for refined single op-amp design.

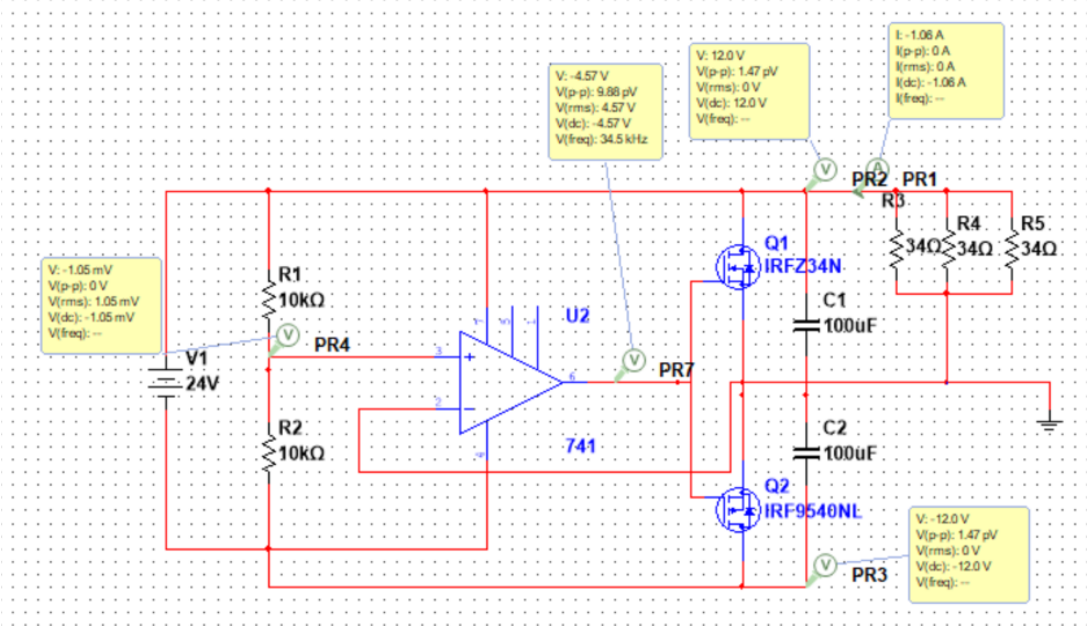


***Figure 3.7.1: Refined Power Supply Design with motor load pulling 1A of current.***

## Load Adjustments and Stability

One of the primary challenges encountered during the implementation was ensuring the power supply remained stable and balanced under various load conditions. Initial tests with no load and a 1kΩ load showed promising results, with minimal deviations from the expected ±12V outputs. However, introducing higher current draw loads, such as three stepper motors, presented challenges in maintaining voltage stability.

To address these challenges, we experimented with different component values and configurations, particularly focusing on the feedback loop involving the op-amps and MOSFETs. Adjustments were made to the resistor values in the voltage divider to fine-tune the reference voltage and ensure accurate feedback control without using large currents.

**Heat Dissipation**

Another significant issue was the excessive heat generated by the MOSFETs under high current loads. Large heatsinks were added to the MOSFETs to improve heat dissipation. Despite this, concerns remained about the long-term reliability of the power supply under continuous operation. This led to further optimizations in the circuit design to reduce power loss and improve efficiency such as a fan. See Figure 3.7.2 for the final power supply implemented on a breadboard with heatsinks on each MOSFET and a fan to further dissipate large amounts of heat.

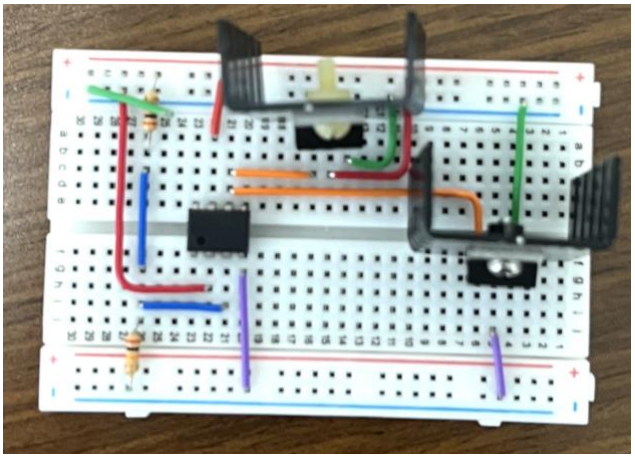

***Figure 3.7.2: Power Supply breadboard implementation with heatsinks***

**Final Design and Testing**

The final design of the power supply was implemented on a breadboard, integrating all necessary components for the ±12V outputs, including the op-amps, MOSFETs and voltage dividing resistors. Comprehensive testing was conducted to validate the performance of the power supply under different load conditions, confirming its ability to maintain stable and balanced ±12V outputs.

**Conclusion**

The development of the ±12V power supply from simulation to implementation was a complex process that required careful consideration of design choices, component selection, and circuit configuration. Through iterative testing and optimization, we overcame challenges related to voltage stability, heat dissipation, and load adaptability. The final power supply design met the project's requirements, providing a stable and reliable power source for the various components and systems within our project.

## 3.8 Analog to Digital Converter (ADC) Design and Simulation:

### Overview

The development of the Analog to Digital Converter (ADC) for our project was a critical step towards achieving precise and real-time monitoring of current levels within our system. After evaluating various ADC architectures, our team decided to pursue the Successive Approximation Register (SAR) ADC approach. This decision was made because of the SAR ADCs balance between speed, accuracy, and complexity, making it suitable for our application's requirements. This section outlines the design process, simulation results, and the rationale behind our design choices for the ADC.

### Choice of ADC Architecture

Initially, our team considered multiple ADC architectures, including Flash ADC and Delta-Sigma ADC. However, due to the Delta-Sigma ADCs complexity and the Flash ADC's limitations in resolution and scalability, we opted for the SAR ADC. The SAR ADC offered a compelling balance of high resolution, moderate speed, and manageable complexity, which aligned with our project's objectives of achieving accurate current measurements without excessively complicating the circuit design. This design approach inputted a PWM duty cycle from the Raspberry Pi 4 and compared it to the voltage being read across the shunt resistor. For this, a LM339 Comparator was used. See appendix section A5 for the full LM339 datasheet.

### SAR ADC Configuration

The SAR ADC was meticulously crafted to sense current. Firstly, we employed a set of three parallel-connected 1Ω shunt resistors to accurately detect the current flowing through the load, since it is wired in series between power supply and system. These shunt resistors, due to their parallel configuration, effectively present a combined resistance of ⅓Ω, necessitating an amplification of the resultant voltage signal to ensure adequate signal strength for comparison. This amplification is achieved by a differential amplifier with a precisely set gain of 3, which scales the signal to a level suitable for comparison. The amplified signal is then inverted by an inverting operational amplifier, to prepare for comparison. The pulse width modulation (PWM) circuit, which includes an RC filter and a buffering operational amplifier, provides a time varying comparison voltage. The buffered signal is then directed to a comparator input, where it is compared against the reference voltage from our shunt readings. The comparator’s role is to generate a binary output that switches state when the filtered PWM signal crosses the shunt reference reading. This binary output is connected to GPIO11 for processing our ADC program to accurately calculate power usage.

**Simulation Results**

The simulation of the SAR ADC was conducted using a software tool, such as Multisim, that allowed us to model the ADC's behavior under various conditions. The simulation aimed to validate the ADC's ability to accurately convert analog signals to digital values, focusing on the resolution, speed, and linearity of the conversion process. See Figure 3.8.0 for fully implemented simulation with a 1kΩ load on output of comparator to determine binary outputs in simulation.

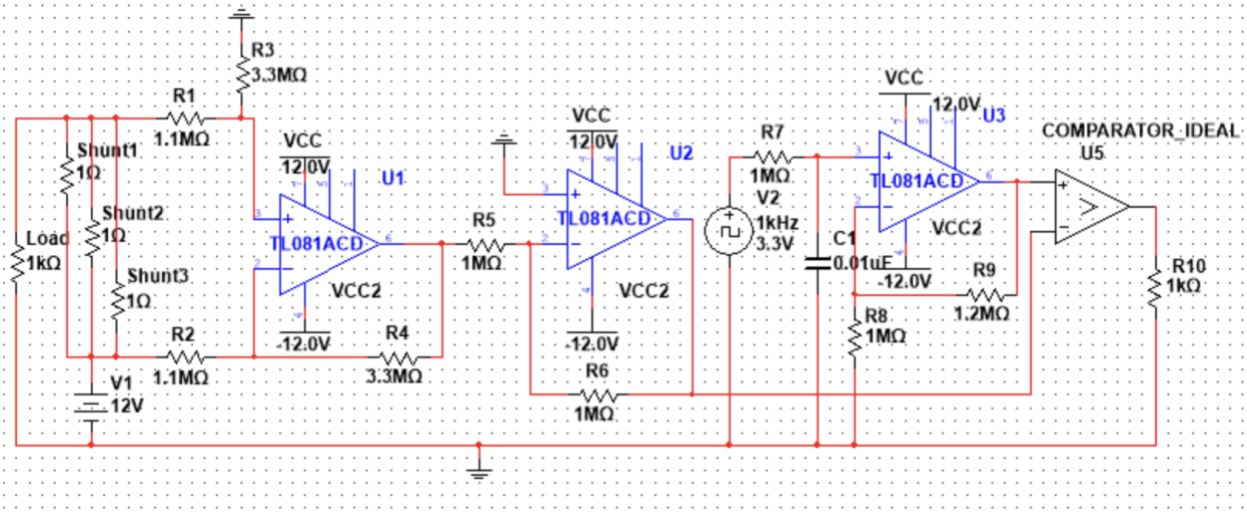


***Figure 3.8.0: Fully Implemented ADC Multisim Simulation***

**ADC Program:**

```
# creates a new wave at a certain duty cycle at the PWM pin
def set_PWM(pi, pin, frequency, duty_cycle):
    pi.set_mode(pin, pigpio.OUTPUT)
    period = 1.0 / frequency

    on_time_us = int(period * duty_cycle /100 * 1e6)
    off_time_us = int(period * (100 - duty_cycle)/100 * 1e6)
    # Create a square wave
    square_wave = [
        pigpio.pulse(1 << pin, 0, on_time_us),
        pigpio.pulse(0, 1 << pin, off_time_us)
    ]
    pi.wave_clear()  # Clear existing waveforms
    pi.wave_add_generic(square_wave)  # Add square wave
    waveform = pi.wave_create()  # Create waveform
    pi.wave_send_repeat(waveform)  # Send waveform repeatedly
    time.sleep(0)
```

```
# stops the current wave, meant for when program terminates
def stop_PWM(pi):
    pi.wave_tx_stop()  # Stop sending waveform
    pi.set_PWM_dutycycle(PWM_PIN, 0)  # Set duty cycle to 0
    pi.stop()  # Cleanup pigpio resources
```

***Figure 3.8.1: set_PWM() and stop_PWM() functions in adc.py***

On a Raspberry Pi, our adc.py program controls a Pulse Width Modulation (PWM) signal via the pigpio library. The code initializes a PWM signal on GPIO19 and monitors an input signal on GPIO11. The duty cycle of the PWM signal is modified in response to the status of the input signal.

The setPWM() method generates a square wave with a certain frequency and the duty cycle delivers it repeatedly via the pigpio library. The duty cycle is the proportion of time the signal is high relative to the overall period of the waveform. The stop_PWM() method terminates the current waveform and resets the PWM duty cycle to zero, freeing up pigpio resources.

The run() function in the ADC class constantly examines the status of the input signal of the comparator and increments the duty cycle by 1 (0-100). This incrementation continues until the comparator state reads high.

```
if CPRT_STATE == 1:

    detected_duty_cycle = duty_cycle

    if duty_cycle > 6:
        duty_cycle -= 6
    else:
        duty_cycle = 0

    if detected_duty_cycle <= 2:
        expected_voltage == 0
    else:
        expected_voltage = 0.056 * detected_duty_cycle
```

***Figure 3.8.2: Code snippet that detects duty cycles.***

When the input signal is found to be high (CPRT_STATE == 1), the PWM signal's duty cycle is changed. If the detected duty cycle is larger than 6, it is reduced by six; otherwise, it is set to zero. This subtraction is to maximize computational efficiency, as setting the duty cycle

completely back down to zero every time it reads HIGH would make it take much longer to reach higher voltages. An accurate conversion from duty cycle to voltage reading was found to be: $V = 0.056 \times duty\ cycle$. (Figure 3.8.2)

Using the detected duty cycle, the program can estimate a voltage and add this new voltage to a list of previous voltage values. After five voltages are recorded, we found that the most accurate reading is always the maximum value of this list. Therefore, it calculates the power usage in watts using the maximum voltage reading of the five and updates the LCD to display this value.

**Resolution and Accuracy**

The system's PWM control is configured to provide 100 distinct output levels by adjusting the duty cycle in 1% increments. To represent these levels digitally, a 7-bit resolution is used, which can encode up to 128 distinct values, more than covering the required range. This level of accuracy ensures that the current measurements are reliable and can be used confidently for monitoring our system.

**Conversion Speed**

In the design of our system, the update frequency is a pivotal parameter, particular in the context of dynamic monitoring. The implemented code utilizes a time-based delay to regulate the update interval. This function is set to introduce a 50-millisecond pause at the end of each loop iteration, thereby dictating the rate at which the system refreshes its readings and adjusts the PWM duty cycle. Similarly, the system is configured to update 20 times per second, which ensures that it can swiftly react to fluctuations in current levels. This rapid update capability is essential for the dynamic reading of current levels with varying loads and maintaining a high level of responsiveness.

**Breadboard Implementation**

In our ADC implementation, we opted for large resistor values in the feedback networks of both the differential and inverting amplifiers to achieve specific design goals. For the differential amplifier, resistor values of 3.3MΩ and 1.1MΩ were selected. This configuration was crucial in achieving a gain of 3, which was necessary to effectively handle the parallel combination of three 1Ω resistors, resulting in a total resistance of 1/3Ω. The choice of these high-value resistors in the differential amplifier was driven by the need to create a more precise feedback loop, which we were initially having problems with.

Similarly, for the inverting amplifier, a feedback resistor of 1MΩ was used. This choice was made to minimize current consumption in the ADC circuitry, which is a critical factor in maintaining the efficiency and performance of the system. The use of a 1MΩ resistor in the feedback loop of the inverting amplifier contributed to a more stable and accurate operation while consuming less current. Our full integration of the ADC can be seen in Figure 3.8.3.

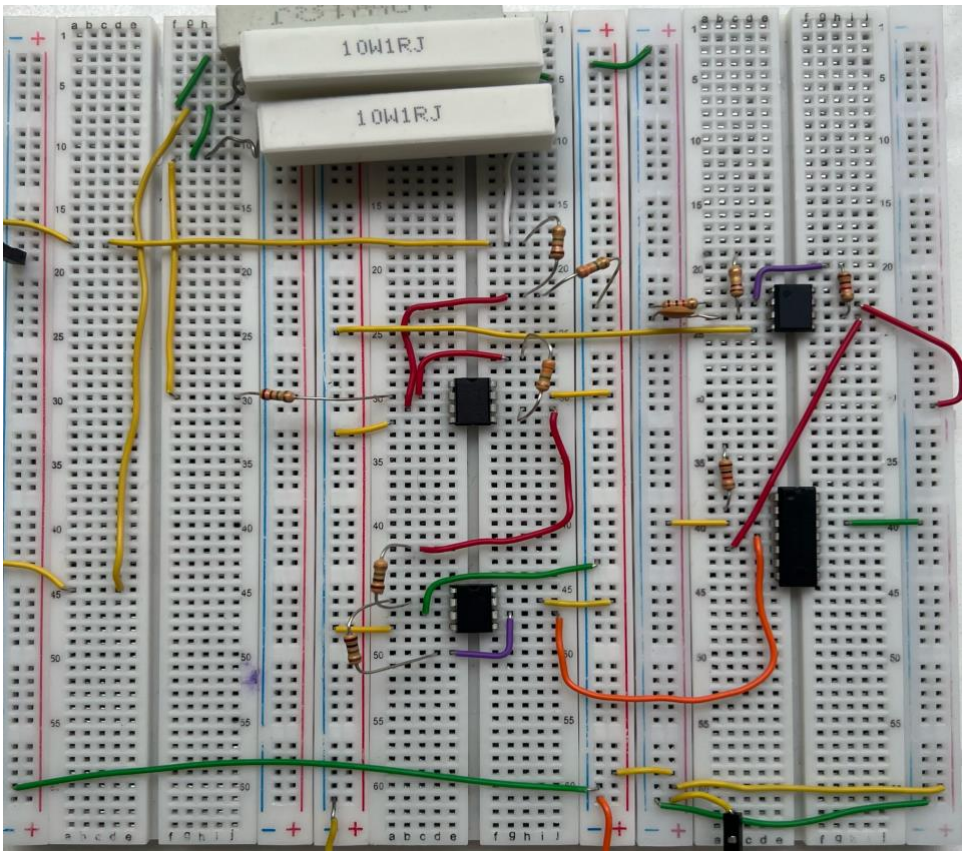


***Figure 3.8.3: Fully Implemented Physical ADC on Breadboard***

**Conclusion**

In conclusion, the development of our Analog to Digital Converter (ADC) using the Successive Approximation Register (SAR) approach proved to be a great choice, balancing high resolution, moderate speed, and manageable complexity. The project highlighted the critical importance of component selection and adaptability. The successful simulation and implementation of the SAR ADC has laid a solid foundation for accurate and real-time current measurement within our system.

## 3.9 GPIO Pins, H-bridges, and Rotary Encoders:

### Introduction

The project involves a sophisticated integration of GPIO pins, H-bridges, and rotary encoders to control stepper motors for precise movements. This setup is crucial for the project's aim to achieve accurate positioning and movement, which is fundamental in applications such as automated plotting or CNC machines.

### Setup and Configuration of GPIO Pins

The initial phase involved setting up the Raspberry Pi (RP4) to use its GPIO pins effectively. Each pin on the Raspberry Pi 4 was configured to perform specific functions such as sending signals to the H-bridges or receiving input from the rotary encoders. The configuration was carefully documented to ensure that each pin's role was clear, avoiding any potential misconnections that could lead to hardware damage or malfunction. Big differences in GPIO pin allocation with other groups include using 4 pins per stepper motor and 2 pins for the ADC. See Figure 3.9.0 for the full and updated GPIO Diagram.

| Actual Pin | GPIO Pin | Alternative Function | Default State at Power Up | State Used for Project | Pull Up/Pull Down | Project Function |
|---|---|---|---|---|---|---|
| 1 | 3.3V PWR | 3.3V POWER | High | High | Pull Up | Powers all 3.3v load demands |
| 2 | 5V PWR | 5V POWER | High | High | Pull Up | Powers all 5v load demands |
| 3 | 2 | I2C1 SDA | High | High | Pull Up | DFRobot LCD Screen |
| 4 | 5V PWR | 5V POWER | High | High | Pull Up | Powers all 5v load demands |
| 5 | 3 | I2C1 SCL | High | High | Pull Up | DFRobot LCD Screen |
| 6 | GND | GROUND | Low | Low | Pull Down | Grounding |
| 7 | 4 | | High | High | Pull Down | Limit Switch (X) |
| 8 | 14 | UART0 TX) UART 0 Transmitter | Low | High | Pull Down | Limit Switch (Y) |
| 9 | GND | GROUND | Low | Low | Pull Down | Grounding |
| 10 | 15 | (UART0 RX) UART 0 Receiver | Low | Low | Pull Down | SN754410 (Motor 3) |
| 11 | 17 | | Low | High | Pull Up | SN754410 (Motor 1) |
| 12 | 18 | | Low | Low | Pull Down | SN754410 (Motor 3) |
| 13 | 27 | | Low | High | Pull Up | SN754410 (Motor 1) |
| 14 | GND | GROUND | Low | Low | Pull Down | Grounding |
| 15 | 22 | | Low | High | Pull Up | SN754410 (Motor 1) |
| 16 | 23 | | Low | High | Pull Up | SN754410 (Motor 1) |
| 17 | 3.3V PWR | 3.3V POWER | High | High | Pull Up | Powers all 3.3v load demands |
| 18 | 24 | | Low | Low | Pull Down | SN754410 (Motor 3) |
| 19 | 10 | SPI0 MOSI | Low | Low | Pull Down | SN754410 (Motor 3) |
| 20 | GND | GROUND | Low | Low | Pull Down | Grounding |
| 21 | 9 | SPI0 MISO | Low | Low | Pull Down | H-Bridge Enable |
| 22 | 25 | | Low | High | Pull Up | SN754410 (Motor 2) |
| 23 | 11 | SPI0 SCLK | Low | Low | Pull Down | ADC Comparator |
| 24 | 8 | SPI0 CS0 | High | Low | Pull Down | Not Used |
| 25 | GND | GROUND | Low | Low | Pull Down | Grounding |
| 26 | 7 | SPI0 CS1 | High | Low | Pull Down | Rotary DT (Motor 2) |
| 27 | Reserved | I2C0 SDA | Low | Low | Pull Down | Reserved for Spare I2C Connections |
| 28 | Reserved | I2C0 SCL | Low | Low | Pull Down | Reserved for Spare I2C Connections |
| 29 | 5 | | High | High | Pull Up | SN754410 (Motor 2) |
| 30 | GND | GROUND | Low | Low | Pull Down | Grounding |
| 31 | 6 | | High | High | Pull Up | SN754410 (Motor 2) |
| 32 | 12 | | Low | High | Pull Up | SN754410 (Motor 2) |
| 33 | 13 | | Low | High | Pull Up | Rotary SW (Motor 2) |
| 34 | GND | GROUND | Low | Low | Pull Down | Grounding |
| 35 | 19 | SPI1 MISO | Low | High | Pull Up | ADC PWM |
| 36 | 16 | SPI1 CSO | Low | High | Pull Up | Rotary SW (Motor 1) |
| 37 | 26 | | Low | High | Pull Up | Rotary CLK (Motor 2) |
| 38 | 20 | SPI1 MOSI | Low | High | Pull Up | Rotary CLK (Motor 1) |
| 39 | GND | GROUND | Low | Low | Pull Down | Grounding |
| 40 | 21 | SPI1 SCLK | Low | High | Pull Up | Rotary DT (Motor 1) |

***Figure 3.9.0: Fully updated GPIO Diagram. Note: Diagram may be hard to read. For an enlarged image, check Appendix A10.***

**Integration and Configuration of H-Bridges**

H-bridges (SN754410) were used to control the direction and speed of the stepper motors. See Appendix section A1 for the full SN754410 datasheet. The RP4 GPIO pins were connected to the H-bridges to enable control over the motors. See Appendix section A2 for the full NEMA17 stepper motor datasheet. Specific GPIO pins were designated to handle the enabling and direction control of the H-bridges. For instance, GPIO9 was used to enable H-bridges, while other pins like GPIO17, GPIO22, GPIO23, and GPIO27 were used to control the motor coils. This setup allowed for precise control over the motor's direction and speed by toggling the GPIO pins high or low. Found in Figure 3.9.1 is a schematic visualization of these components interfacing with each other.

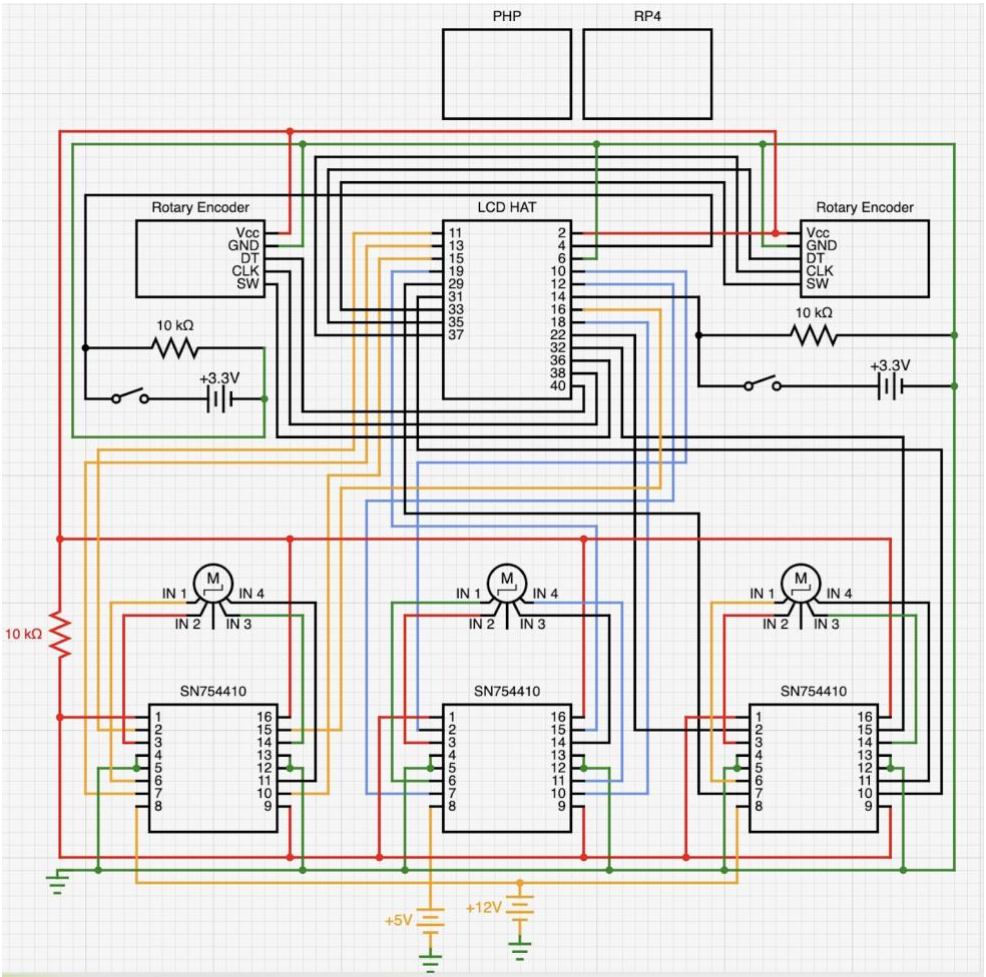


***Figure 3.9.1: H-Bridge, limit switches, and rotary encoder implementation schematic.***

**Incorporation of Rotary Encoders**

Rotary encoders (KY-040) were integrated to provide a user interface for manually controlling the stepper motors. Each encoder was connected to specific GPIO pins on the RP4, which were configured to read the encoder's signals. See Appendix section A3 for the full KY-040 datasheet. These signals, generated by the rotary movement of the encoders, were interpreted by the RP4 to adjust the stepper motors accordingly. The encoders are connected to the RP4 GPIO pins, with each encoder requiring connections for power (5V), ground, and signal outputs (DT, CLK, SW). They allowed for manual adjustment of the motor's position, which was essential for setting initial conditions or for manual override purposes. In the final design, it was decided that the rotary encoders should be placed on a separate breadboard for a more friendly user experience. See figure 3.9.2 for an image showing this implementation.

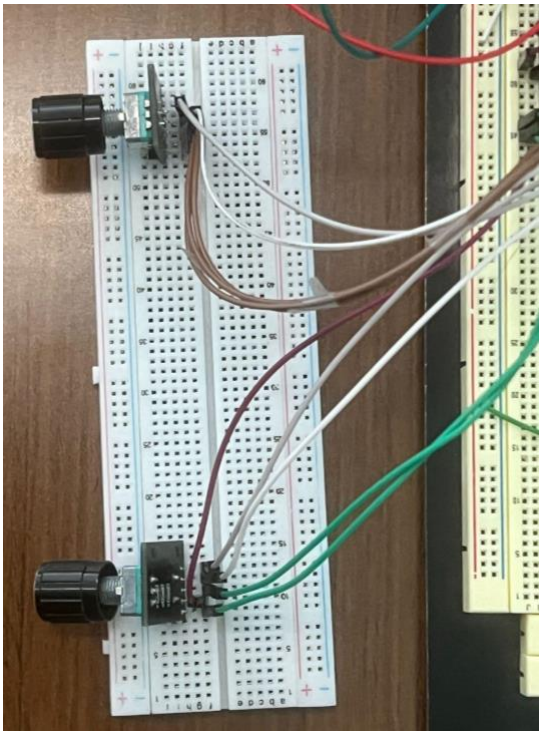

***Figure 3.9.2: Image showing separate rotary encoder placement.***

**Software Development for Control Logic**

Software was developed to tie together the hardware components. The software read the signals from the rotary encoders through the GPIO pins and controlled the H-bridges to drive the stepper motors. The program included error handling to manage potential signal noise or erroneous inputs from the rotary encoders. Additionally, the software was designed to ensure that the motors stopped at their limit positions to prevent mechanical overruns, which was managed by additional GPIO inputs from limit switches. See software part of technical designs section for more information.

**Testing and Calibration**

The final phase involved thorough testing and calibration of the system. The rotary encoders were adjusted to ensure accurate input signals. The H-bridges were tested under various load conditions to ensure they could handle the expected current and voltage without overheating or failure. The entire system was then calibrated to ensure that the movement of the stepper motors accurately corresponded to the inputs from the rotary encoders. See Figure 3.9.3 for full breadboard implementation showing limit switch connections, H-bridges, and GPIO connections.

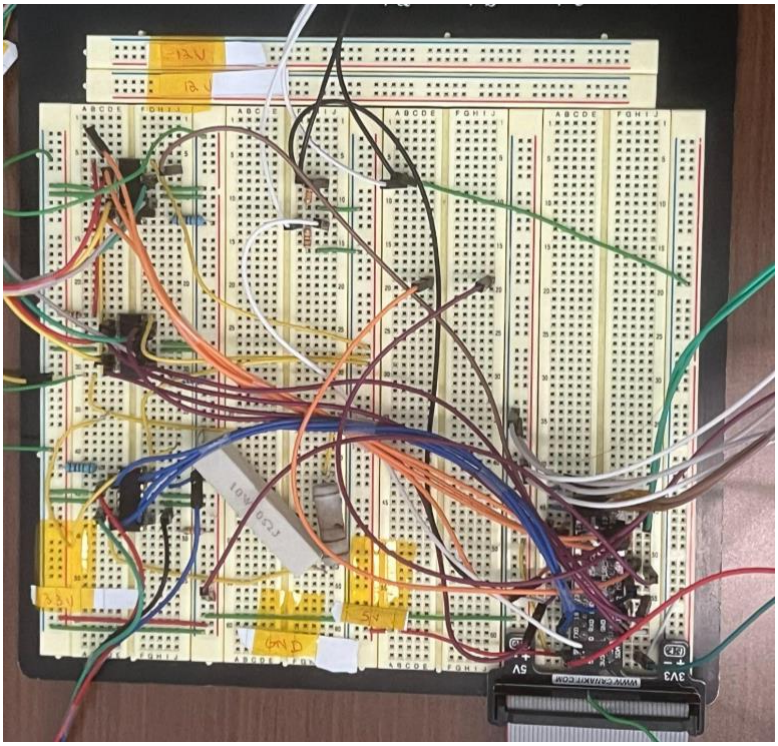

***Figure 3.9.3: Image showing main breadboard implementation.***

### Conclusion

The integration of GPIO pins, H-bridges, and rotary encoders in this project demonstrates a complex yet robust control system for stepper motors. This setup not only allows for precise motor control but also provides flexibility and manual control through the rotary encoders. The detailed documentation and careful configuration of each component ensure a reliable and efficient system capable of precise movements required in automated plotting or similar applications.

## 3.10 Printed Circuit Board (PCB) Development:

### Introduction

The development of the Printed Circuit Board (PCB) for our project was a critical step in transitioning from a breadboard prototype to a more robust and reliable hardware platform. The PCB design aimed to integrate all the necessary components, including H-bridges, rotary encoders, limit switches, and the interface for the Raspberry Pi (RP4). This section details the process of designing the PCB, from the initial schematic to the final layout, and the rationale behind the design choices made.

### Schematic Design

The initial phase of PCB development involved creating a detailed schematic that outlined the connections between all the components. The schematic served as a blueprint for the PCB layout and ensured that all components were correctly interfaced with the RP4. We chose to include three H-bridges (SN754410) to control the stepper motors, rotary encoders for user input, and limit switches for boundary detection. The LCD Pi Hat was also incorporated to provide a user interface. Since the motors were not available in Autodesk Fusion, we used connectors corresponding to the pin numbers. For more information about Autodesk Fusion, visit https://www.autodesk.com/products/fusion-360/overview?term=1-YEAR&tab=subscription. The encoders were connected using 5-pin connectors, the motors with 4-pin connectors, and the LCD Pi Hat with a 2x20 pin connector. The H-bridges were found in the design software and directly placed on the schematic, eliminating the need for additional connectors for these components. See Figure 3.10.0 to see the schematic built in Autodesk Fusion.

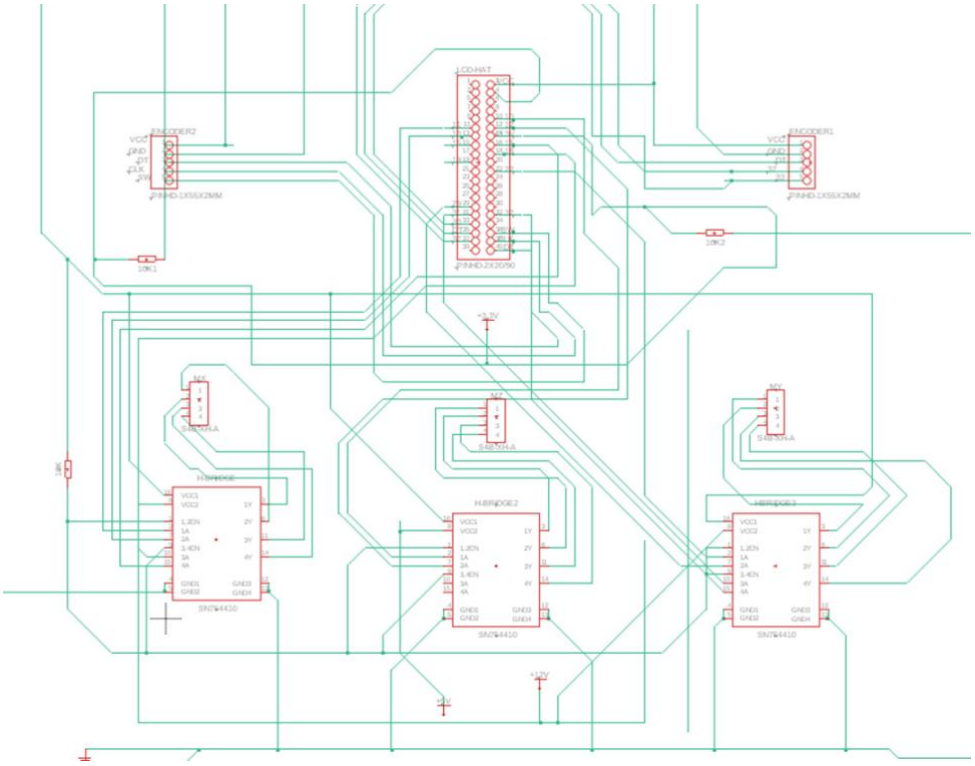

***Figure 3.10.0: Full Schematic built in Autodesk Fusion***

**PCB Layout**

After the schematic was finalized, we converted it into a PCB layout. This step involved placing the components onto the PCB and routing the connections between them. The three H-bridges were positioned at the bottom of the PCB, with the LCD Pi Hat pinout above them. The stepper motor connectors were placed above the H-bridges, and the rotary encoders and pull-up resistors were located at the top of the PCB. Careful attention was paid to the routing to ensure that all connections were correct and that there were no shorts or potential points of failure. The layout was checked multiple times to confirm the integrity of the connections. See Figure 3.10.1 for final PCB design.

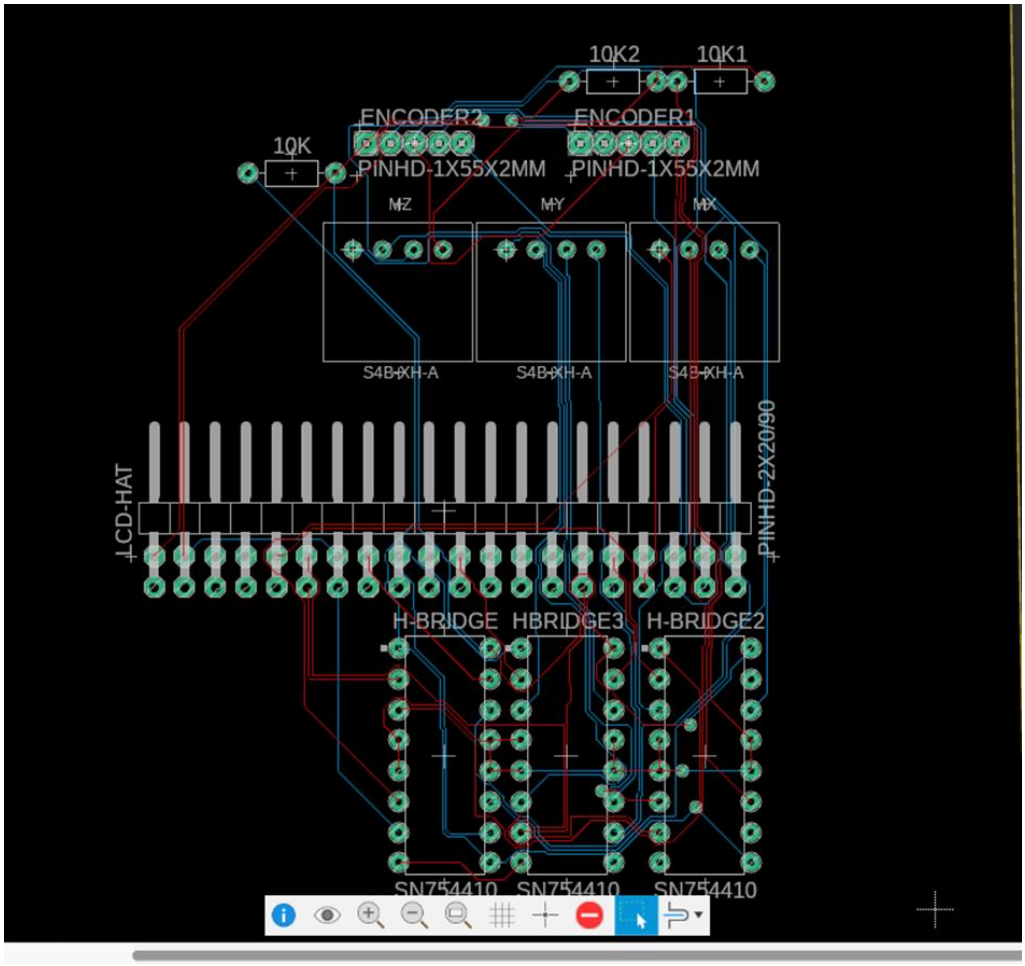


***Figure 3.10.1: Full PCB built in Autodesk Fusion***

**Conclusion**

The PCB design process was a meticulous task that required careful planning and attention to detail. The resulting PCB is a testament to the team's ability to create a hardware platform that not only meets the project's current requirements but is also capable of accommodating future enhancements. If the PCB was ordered and implemented it would significantly improve the reliability and usability of our system, marking a much more consumer-ready product.

# 4. Ethics

## Introduction

Our team was committed to ensuring the originality of our design throughout the project. To achieve this, we adopted several strategies:

### Inspiration, Not Imitation

While we took inspiration from the work of other groups, we were careful never to copy designs. We recognized that with the multitude of components involved in our project, the likelihood of two teams arriving at identical designs was slim, provided that each team worked independently.

### Independent Development

We focused on developing our project independently. This approach involved brainstorming sessions where all team members contributed unique ideas that were synthesized into the final design. By relying on the diverse expertise and creativity within our team, we ensured that our design was a product of our collective innovation.

### Regular Documentation and Review

Throughout the project, we maintained detailed documentation of our design process. This included sketches, notes, and discussions that led to the final design. Regular reviews of this documentation helped us track the evolution of our design and ensured that it remained distinct and original.

### Ethical Adherence to Professional Standards

We adhered to the principles outlined in the Engineer's Code of Ethics, which emphasizes the importance of honesty and integrity. By following these guidelines, we held ourselves accountable to the highest professional standards, which naturally fostered an environment of originality and respect for intellectual property. Refer to Appendix C for link to full Engineer Code of Ethics.

**Public Safety and Welfare**

As per the Engineer's Code of Ethics, our foremost responsibility would be to ensure that the product is safe for public use (with obvious modifications such as a PCB). This would involve rigorous testing and adherence to safety standards to prevent harm to users.

**Honest Representation**

We would need to honestly represent the capabilities and limitations of our product, avoiding any exaggeration that could mislead users.

**Conclusion**

In conclusion, our team's commitment to originality and ethical considerations is rooted in a deep understanding of the professional responsibilities of engineers. By upholding these values, we aim to contribute positively to the field and society at large.

# 5. Self-Critique

## Overview

Reflecting on our journey through this project, it's clear that our team has navigated a path filled with both achievements and learning opportunities. Our primary goal was to reach Checkpoint A-, a milestone we successfully achieved through dedication, collaboration, and technical skill. However, as we look back, there are areas where we recognize the potential for improvement, particularly in documentation and project tracking.

## Teamwork and Collaboration

Our team's performance in terms of teamwork and collaboration was commendable. Each member brought unique skills and perspectives to the table, contributing to a well-rounded and effective team dynamic. Regular meetings and open communication channels ensured that everyone was aligned with the project's goals and progress. This strong foundation of teamwork was instrumental in overcoming the challenges we faced and in achieving our goal of reaching Checkpoint A-.

## Hardware and Software Development

On the technical front, our hardware and software development efforts were marked by innovation and problem-solving. We successfully designed and implemented complex components, such as the Analog to Digital Converter (ADC) and the Power Supply, demonstrating our ability to translate theoretical concepts into practical solutions. Additionally, our group developed software that not only worked, but allowed our motors to run extremely smoothly, using complex concepts such as threading. Our approach to hardware and software integration was methodical, allowing us to build a robust system capable of meeting the project's requirements.

## Documentation and Project Tracking

While our technical achievements are a source of pride, our project's documentation and tracking aspects highlight areas for improvement. In hindsight, a more structured approach to documenting our thought processes, design decisions, and progress would have been beneficial. Regularly capturing detailed notes and taking photographs of our work could have provided a clearer picture of our journey, facilitating reflection and learning.

Additionally, maintaining a log of the hours spent on the project during different periods could have offered valuable insights into our time management and efficiency. This data could have been useful for planning and allocating resources more effectively, ensuring that we maximized our productivity without compromising the quality of our work.

## Conclusion

In conclusion, our team's performance on this project was characterized by strong teamwork, technical proficiency, and a commitment to achieving our goals. However, the experience has also highlighted the importance of thorough documentation and project tracking as essential components of successful project management. Moving forward, we will incorporate these lessons into our future endeavors, aiming not only to replicate our successes but also to address the areas where we have room to grow. This project has been a valuable learning experience, and we are confident that the insights gained will enhance our approach to future challenges.

# 6. Revision Plan

## Introduction

Reflecting on our project's progress and outcomes, there are several areas where we could implement changes to enhance the design and functionality of our system. These revisions not only aim to refine the project for academic purposes but also consider the transition of the project into a real-world product. This plan outlines potential improvements in hardware integration, user interaction, and system reliability.

## Integrated PCB Design

One significant enhancement would be the development and integration of a printed circuit board (PCB). Currently, our setup involves multiple separate components connected on a breadboard, which, while functional, does not offer the robustness or the aesthetic appeal necessary for a commercial product. By designing a custom PCB that consolidates all electronic components, we could significantly improve the system's durability and reduce the likelihood of connection issues. Attaching the PCB directly to the plotter's gantry could streamline the design, reducing the wiring complexity and improving the overall stability of the system. A well-designed PCB would provide a more professional look and feel, crucial for a commercial product, ensuring that users have confidence in the device's reliability and build quality.

## Enclosure for Rotary Encoders

Another area for improvement involves the physical handling and protection of the rotary encoders. Currently, the encoders are exposed, which could lead to operational inaccuracies or damage from environmental factors. Designing and printing custom cases for the rotary encoders would protect them from dust and physical damage while giving the device a more finished and consumer-friendly appearance. These cases would not only protect the encoders but also enhance the tactile feedback for users, potentially improving the user interface interaction.

## Wireless Rotary Encoders

Enhancing the functionality of the rotary encoders to operate wirelessly would significantly improve the flexibility and usability of the system, especially in a classroom or workshop setting where multiple users might interact with the device. Implementing Wi-Fi-enabled rotary encoders would eliminate the need for physical connections to the Raspberry Pi, reducing clutter

and enhancing ease of use. In order to do this, however, additional microcontrollers would likely have to be attached to the rotary encoders and these microcontrollers could then send information wireless via MQTT to the Raspberry Pi. Wireless encoders would allow users to control the device remotely, potentially integrating with other devices or platforms for expanded functionality.

### Motor Performance and Startup Reliability

While the motor performance has been satisfactory, the reliability of the system startup using cron has room for improvement, especially considering user experience in a real-world application. Investigating alternatives to cron for more reliable startup procedures or enhancing the existing cron setup to increase its consistency is crucial. For a commercial product, reliability upon startup is critical, as users expect the device to function correctly each time it is powered on without needing multiple attempts.

### Accessibility Features

Considering the accessibility of the device for users with disabilities could broaden the market reach and usability of the product. Integrating an audio feedback system that reads out the user interface options would make the device more accessible to visually impaired users. This feature would not only comply with accessibility standards but also enhance the user experience for all users by providing auditory cues during operation.

## Conclusion

The proposed revisions aim to refine our project from a functional prototype to a more polished, reliable, and user-friendly product. By integrating a custom PCB, providing protective enclosures for rotary encoders, enabling wireless operation, improving startup reliability, and incorporating accessibility features, we can significantly enhance the overall quality and appeal of our device. These improvements would not only meet the academic project's goals but also lay a solid foundation for potential commercial development.

## 7. Individual Reflection

**Daniel Poorak:**

Reflecting on the journey through this design course, I am still shocked by the intensity and rigor it demanded, both technically and collaboratively. This course was not just a test of individual skill but also a profound exercise in teamwork and collective problem-solving. There were moments that truly tested our resolve and capabilities, pushing us to the max of our technical knowledge. However, the support and cooperation within our group played a pivotal role in navigating these challenges. Throughout the course, our team was committed to achieving our set goals, such as reaching checkpoint A- and enhancing specific project requirements like the ADC or motor movement. The dedication to these objectives was evident in the hard work and long hours we invested. Our efforts were not just about meeting minimum standards but striving for excellence and continuous improvement. One of the most impactful aspects of our project was the strategic decisions made early in the process. Deciding to abandon the hex inverter in favor of a fully code-based solution was a turning point. This decision, among others, was the result of thorough analysis and consideration between the whole team. These early choices paid dividends as the project progressed, streamlining our processes and enhancing our final outcomes. Each person brought not only their expertise but also a passion for quality and innovation. This collective enthusiasm was a driving force behind our project's success. It created an environment where creativity and precision were critical, leading to a project outcome that we are all proud of. In conclusion, this 2920 course was a remarkable learning experience that extended beyond the confines of traditional education. It was a testament to the power of teamwork and strategic planning. The challenges we faced were significant, but they were also catalysts for growth and learning. I am confident that the skills and insights gained here will be priceless in not only my future endeavors, but my group members too.

**Neel Desai:**

Throughout this project, I have gained invaluable insights and skills that have significantly shaped my approach to teamwork and technical challenges. One of the most profound lessons was the importance of effective communication within a team. I learned that clear communication not only facilitates smoother workflow but also builds mutual respect among team members. This respect was evident as we honored each other's time, which in turn made our work sessions more productive and focused. I also discovered the importance of voicing my opinions. Initially, I was hesitant to share my thoughts, but I realized that every team member's perspective could provide unique solutions to the problems we faced. This experience taught me that my contributions are valuable, and speaking up can lead to innovative solutions that might not have been considered otherwise. On the technical front, the project was a rich

learning ground. I delved into the complexities of designing and implementing a multifaceted system, working hands on to debug pressing issues. However, there were areas where I could have improved. I wish I had engaged more deeply with the software aspects of our project. A better grasp of the coding would have allowed me to contribute more comprehensively to all facets of the project and understand the intricacies of our system more fully. In conclusion, this project was not just about building a technical system, but also about personal growth and learning how to function effectively within a team. The skills and lessons I have learned are invaluable and will undoubtedly influence my approach to future projects and professional collaborations.

**Tae Lee:**

Reflecting on our group project, I realize it was much more than just a school assignment; it was a significant growth experience. Reaching Checkpoint A- demonstrated our ability to collaborate and use our skills effectively, underscoring the importance of setting high goals and teamwork. We took great pride in the product we developed, which was a testament to our creativity and dedication. This project not only sharpened our technical skills, like Python programming and circuit design, but also our soft skills, including communication and project management. Documenting our process thoroughly ensured that future teams could build on our foundation, highlighting the lasting impact of good record-keeping. Overall, this project pushed us, broadened our abilities, and taught us valuable lessons in perseverance and collaboration.

**Nathan Park:**

Although I predominantly worked on the software of the project, I found it infinitely valuable to see the connection between hardware and software throughout the project, and I feel that the experience helps define what it means to be a computer systems engineer. Just as there exists a connection between the electrical components to the programming components of the project, there also exists the connection between electrical and computer systems engineering team members. The collaboration between the two fields proved to be perfectly matched and systematic for this project. I feel that the foundations of being an engineer, from technical knowledge to interpersonal skills, were thoroughly explored in this project, surprisingly with the latter proving to be of even greater importance. Furthermore, these communication skills were imperative to support this connection, and these skills were by far in a way the most important to have throughout this project. I learned that the presence of effective teamwork, leadership, and organization is the difference between an outstanding team and an adequate one. Above all, I learned to trust my teammates and value their perspectives. Without the support of each other, there were many times where we would have given up. There were countless points throughout the design process in which a single team member’s input changed the course of the project for

the better, and, because of these instances, I concluded that the accumulation of good ideas can only be facilitated by teams with an open and accepting mindset, and that this accumulation helps push our work to the highest standard. I believe the insights towards the value of teamwork gained from experiences like these define the core of an engineer: working together. Through the valuable lessons I've learned along the way, I'm confident that the path I've taken for this project will continue to have a profound impact upon my career.

**Umair Irshad:**

Facing time pressures, limited resources, and technical hurdles really put our teamwork and problem-solving to the test. It was a challenging ride, but it taught us a lot about adaptability, planning, and sticking together when the going gets tough.

# 8. Appendix

## 8.1 Appendix A

Hardware

Datasheet A1. SN754410 Quadruple Half-H Driver datasheet (Rev. C) (https://www.ti.com/lit/ds/symlink/sn754410.pdf)

- Used to reference the SN754410 pinout as well as referencing the Typical Application section in 10.2.

Datasheet A2. NEMA17 12v 350ma Stepper Specification Sheet (https://core-electronics.com.au/attachments/localcontent/C140-Adatasheet_77186b14070.jpg)

- Used to reference the stepper motors wiring and which wires go to which coil of the bipolar stepper. This was important because the sequence of the wires is important for proper function of the bipolar stepper motor.

Datasheet A3. Rotary Encoder for Arduino/Raspberry (https://www.handsontec.com/dataspecs/module/Rotary%20Encoder.pdf)

- Used to reference the pinout of the rotary encoder as well as looking ahead at its full functionality and how it will be integrated in the future.

Datasheet A4. LCD Pi Hat Datasheet (https://wiki.dfrobot.com/I_O_Expansion_HAT_for_Pi_zero_V1_0_SKU_DFR0604IIC_16X2_RGB_LCD_KeyPad_HAT_1_0_SKU_DFR0514_DFR0603)

- Used to reference what pins would be taken up by the LCD Pi Hat to determine which pins would remain available after integration.

Datasheet A5. LM339N Comparator (https://www.ti.com/lit/ds/symlink/lm339.pdf?ts=1713502839469&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FLM339)

- Used to reference the operating principle of the comparator as well as understand some issues such as needing the pull up resistor and the input voltage offset issues.

Datasheet A6. LM741 Op Amp (https://www.ti.com/lit/ds/symlink/lm741.pdf)

- Used to reference the operating principle and how it could potentially be implemented into our power supply design to achieve our requirements.

Datasheet A7. IRFZ34N
(https://www.infineon.com/cms/en/product/power/mosfet/n-channel/irfz34n/)

- Used to reference the NMOS MOSFET to understand things such as G,S,D, and how it would be implemented into the power supply.

Datasheet A8. IRF9540NL
(https://www.infineon.com/cms/en/product/power/mosfet/p-channel/irf9540nl/)

- Used to reference the PMOS MOSFET to understand things such as G,S,D, and how it would be implemented into our power supply.

Datasheet A9. Z-motor Stepper Specification Sheet
(https://github.com/Herring-UGAECSE-2920-S24/Deliverables/blob/main/Tools/Zaxis%20stepper.pdf)

- Used to reference the Z-motor to resolve some of our issues, namely understanding the wire pairs, and determining the voltage requirements to fix motor behavior issues.

Figure A10. Enlarged image of GPIO Power Diagram

| Actual Pin | GPIO Pin | Alternative Function | Default State at Power Up | State Used for Project | Pull Up/Pull Down | Project Function |
|---|---|---|---|---|---|---|
| 1 | 3.3V PWR | 3.3V POWER | High | High | Pull Up | Powers all 3.3v load demands |
| 2 | 5V PWR | 5V POWER | High | High | Pull Up | Powers all 5v load demands |
| 3 | 2 | I2C1 SDA | High | High | Pull Up | DFRobot LCD Screen |
| 4 | 5V PWR | 5V POWER | High | High | Pull Up | Powers all 5v load demands |
| 5 | 3 | I2C1 SCL | High | High | Pull Up | DFRobot LCD Screen |
| 6 | GND | GROUND | Low | Low | Pull Down | Grounding |
| 7 | 4 | | High | High | Pull Down | Limit Switch (X) |
| 8 | 14 | (UART0 TX) UART 0 Transmitter | Low | High | Pull Down | Limit Switch (Y) |
| 9 | GND | GROUND | Low | Low | Pull Down | Grounding |
| 10 | 15 | (UART0 RX) UART 0 Receiver | Low | Low | Pull Down | SN754410 (Motor 3) |
| 11 | 17 | | Low | High | Pull Up | SN754410 (Motor 1) |
| 12 | 18 | | Low | Low | Pull Down | SN754410 (Motor 3) |
| 13 | 27 | | Low | High | Pull Up | SN754410 (Motor 1) |
| 14 | GND | GROUND | Low | Low | Pull Down | Grounding |
| 15 | 22 | | Low | High | Pull Up | SN754410 (Motor 1) |
| 16 | 23 | | Low | High | Pull Up | SN754410 (Motor 1) |
| 17 | 3.3V PWR | 3.3V POWER | High | High | Pull Up | Powers all 3.3v load demands |
| 18 | 24 | | Low | Low | Pull Down | SN754410 (Motor 3) |
| 19 | 10 | SPI0 MOSI | Low | Low | Pull Down | SN754410 (Motor 3) |
| 20 | GND | GROUND | Low | Low | Pull Down | Grounding |
| 21 | 9 | SPI0 MISO | Low | Low | Pull Down | H-Bridge Enable |
| 22 | 25 | | Low | High | Pull Up | SN754410 (Motor 2) |
| 23 | 11 | SPI0 SCLK | Low | Low | Pull Down | ADC Comparator |
| 24 | 8 | SPI0 CS0 | High | Low | Pull Down | Not Used |
| 25 | GND | GROUND | Low | Low | Pull Down | Grounding |
| 26 | 7 | SPI0 CS1 | High | Low | Pull Down | Rotary DT (Motor 2) |
| 27 | Reserved | I2C0 SDA | Low | Low | Pull Down | Reserved for Spare I2C Connections |
| 28 | Reserved | I2C0 SCL | Low | Low | Pull Down | Reserved for Spare I2C Connections |
| 29 | 5 | | High | High | Pull Up | SN754410 (Motor 2) |
| 30 | GND | GROUND | Low | Low | Pull Down | Grounding |
| 31 | 6 | | High | High | Pull Up | SN754410 (Motor 2) |
| 32 | 12 | | Low | High | Pull Up | SN754410 (Motor 2) |
| 33 | 13 | | Low | High | Pull Up | Rotary SW (Motor 2) |
| 34 | GND | GROUND | Low | Low | Pull Down | Grounding |
| 35 | 19 | SPI1 MISO | Low | High | Pull Up | ADC PWM |
| 36 | 16 | SPI1 CS0 | Low | High | Pull Up | Rotary SW (Motor 1) |
| 37 | 26 | | Low | High | Pull Up | Rotary CLK (Motor 2) |
| 38 | 20 | SPI1 MOSI | Low | High | Pull Up | Rotary CLK (Motor 1) |
| 39 | GND | GROUND | Low | Low | Pull Down | Grounding |
| 40 | 21 | SPI1 SCLK | Low | High | Pull Up | Rotary DT (Motor 1) |

## 8.2 Appendix B

Software

Datasheet B1. RGB1602.py File
(https://github.com/Herring-UGAECSE-2920-S24/Deliverables/blob/main/Tools/rgb1602.py)

- o Used to better understand how the RGB1602 library works so that we can use some of the built in functions to better improve our implementation of LCD.

Datasheet B2. Launch on Boot Readme
(https://github.com/Herring-UGAECSE-2920-S24/Deliverables/blob/main/Deliverables/setup/launch-on-startup.md)

- o Used to reference how to get a program running upon startup of the RP4 using cron.

Datasheet B3. Stepper Motor Code
(https://www.python-exemplarisch.ch/drucken.php?inhalt_mitte=raspi/en/steppermotors.inc.php)

- o Used to reference for our RP4 stepper motor code, changed the code to work with our hardware application. Instead of using RPi library, used pigpio to stay consistent with other programs, all of which are using pigpio.

Datasheet B4. pigpio Package
(https://abyz.me.uk/rpi/pigpio/)

- o Used to read and write from GPIO Pins. Specific functions for PWM and overriding default state were also used

Datasheet B5. Rotary Class in pigpio Package
(https://pypi.org/project/pigpio-encoder/)

- o Used only for setting callbacks with left and right rotations for each rotary Encoder. Switch press functionality not used.

## 8.3 Appendix C

General

Link C1. Engineer Code of Ethics: National Society of Professional Engineers (https://www.nspe.org/sites/default/files/resources/pdfs/Ethics/CodeofEthics/NSPECodeofEthicsforEngineers.pdf)

- Used to reference ethical procedures when making decisions regarding the project