Smart Motor Syringe System

Connor Wilson, *Member, IEEE,*
EE Dept. Senior Project, *California Polytechnic San Luis Obispo*, Winter 2022
Advisor: Dr. Benjamin G. Hawkins

***Abstract*—In this document a proof-of-concept design is developed and implementation to showcase the latest development in an ongoing effort to produce a series of automated syringe pumps.**

I. Introduction

THERE is a desire to control flow rates in syringes in a steady and automated manner in the medical field; whether that be IV fluids or micro-fluids. According to PR NewsWire, the automated syringe pump market is currently valued at $4 billion USD and is set to double its growth by 2030 [1]. These numbers suggest administering fluids in a controlled manner is a profitable and valuable piece of technology.

However, this task is difficult to do manually, requiring precise digital control to achieve success. Because this device is related to medical and bio-engineering research, greater emphasis is placed on maintaining consistent flow rates and precision control. To achieve this key feature, typical syringe pumps utilize the precision of micro-controllers, large stepper motors, and reduction gear ratios in the drive train.

With the introduction of precision control, keeping this project low cost is a secondary goal to make medical grade technology readily available. While existing products in the market are numerous, they are traditionally priced above $1000 USD and are not easily programmable nor user friendly. Using readily available, non-proprietary hardware and software, I aim to provide reliable automated syringe systems for a fraction of the cost. This system of syringe pumps will also be controlled from a single controller, allowing the user to operate without manually addressing individual devices. This hub controller will add hierarchy that provides a more native and intuitive interface for the user, saving time and other resources used during setup.

It should be noted that my contribution to this project does not constitute a finished product but rather an ongoing pursuit for the minimum viable product as defined by Doctor Ben Hawkins.

II. Background

A. Purpose of a Syringe Pump

A syringe pump (or driver) is defined as a "small infusion pump, used to gradually administer small amounts of fluid (with or without medication) to a patient or for use in chemical and biomedical research." [2] The need for slow incorporation of fluids is used in many applications. Intravenous (IV) therapies, for example, require syringe pumps in order to deliver medication over the course of several minutes or hours. Automating these doses removes most human error in high-risk procedures while saving the medical staff man-hours.

Fig. 1: Example of a Syringe Pump's IV Application [3]

Additional uses for syringe pumps lie in micro-fluid research applications such as: enzyme kinetics, chemical synthesis, and potentially dangerous chemical reactions. For this project, the immediate use of this product will be for Dr. Ben Hawkins's research in enzyme solution gradients.

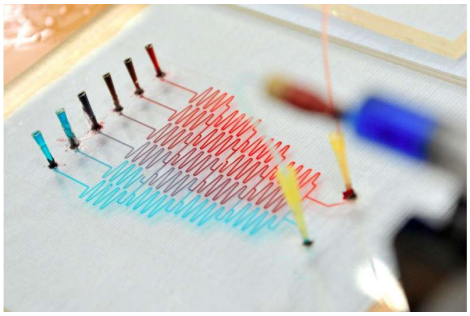

Fig. 2: Example of Microfluid Application [4]

B. Description of the Smart Motor Syringe Pump

The SMSS (Smart Motor Syringe Pump) performs as the defined syringe pump, including but not limited to: output accurate/precise flow rates, display input/output hardware parameters, and control several syringe pumps, all while remaining under $500 USD.

The SMSS also aims to solve two problems, the first being inconsistent flow rates. Looking at existing technology, most low end pumps work on the basis of summing impulses over time such as the NE-300 in Figure 3. An ideal pump would have smooth flow and quick transient responses. However, the main concern for the SMSS in this stage is to maintain a smooth flow rate and address sluggish response times in another prototype. Quintessentially, the SMSS is

built to output flow rates that are precise and accurate. An understanding of the system's plant and controller interface is required to operate within the desired tolerances and well worth the time investment. This will be discussed later in the paper.

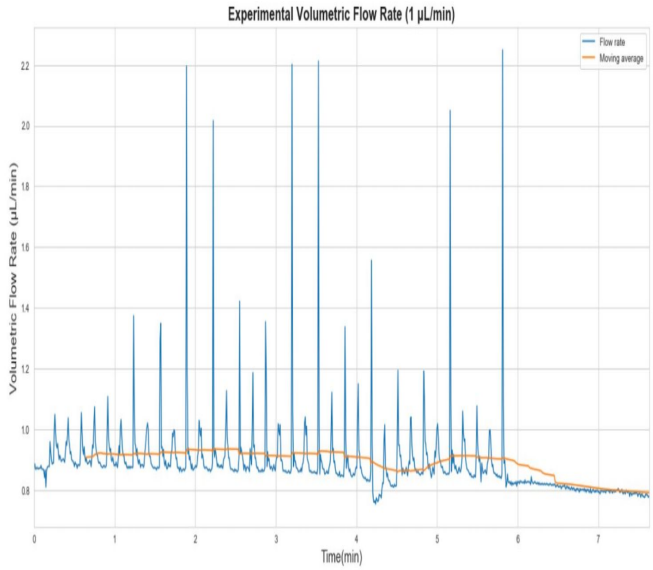

Fig. 3: Output Flow from a NE-300 Syringe Pump [5]

The second problem arises with the addition of other syringe pumps: multiple pumps will often require a synchronized start/stop time from which to collect data from. The main/sub hierarchy implemented aims to synchronize the start/stop of several units and eliminate the errors presented when triggering each pump manually.

Looking broadly at this project, the SMSS aligns with a research market: laboratory applications that deliver small amounts of fluid while collecting data on the system. It should be noted that while syringe pumps can differ in applications, their internal structure is the same: they all consist of a stepper motor, pusher block, and a syringe holder (Figure 4).

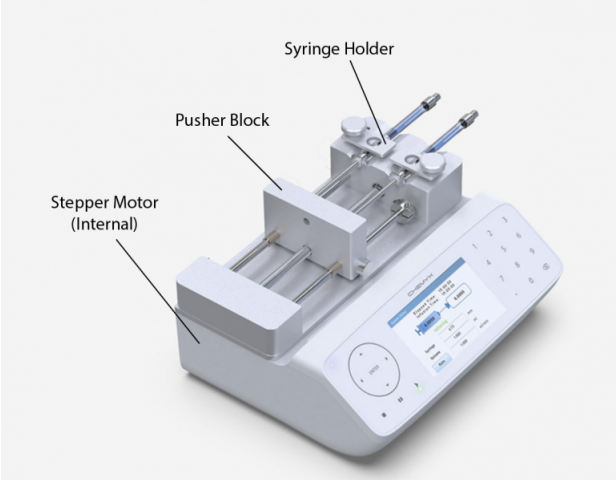

Fig. 4: Components of a Syringe Pump [6]

The SMSS makes use of the Raspberry Pi 3's computing power by implementing a display for the user; a feature forgotten on some devices on the market. The user interface prompts for user requests but also informs the user of resulting error codes, run time, and hardware issues with their requests.

III. Product Planning and Research

Evaluation of the SMSS and it's viability as a product requires an analysis of existing technology on the market. Only when I looked at the existing products could I determine the engineering requirements and priority of each.

A. Marketing

This subsection is focused on the marketing aspect of a syringe pump in an attempt to translate customer needs into a viable and cost effective product.

1) Marketing Research: Using Google's tool for search trends, there was a low number of searches in the US and internationally for syringe pumps. This signals that the amount of new customers looking to purchase an SMSS is small, or that existing customers in the market do not use Google to search for new products. These customers might look to a medical supplier or academic group when looking for a new syringe pump. The Dublin PRNewswire [1] has stated that the IV pump market is valued at $4 billion USD (in 2020) and is expected to double by 2030. This speculation is pre-COVID but it does hold some merit. Looking at Figure 5, there appears to be a larger customer base outside the US; range for Figure 5 is $\in$ (12-211) hits in January 2021.

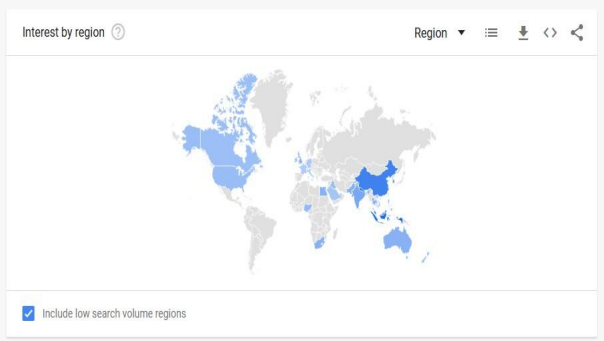

Fig. 5: Global Google Search of "Syringe Pump" 2021 [7]

Further information was gathered in Figure 6 when searching for syringe pumps: there are many applications and characteristics for existing products on the market. My sole customer's [Dr. Hawkins] request for a research-oriented device represents smaller portion when compared to infusions, but it should be noted that the SMSS can achieve the minimum viable product for multiple applications. Qualifying for various FDA medical device classes will be left up to the next iteration of the SMSS.

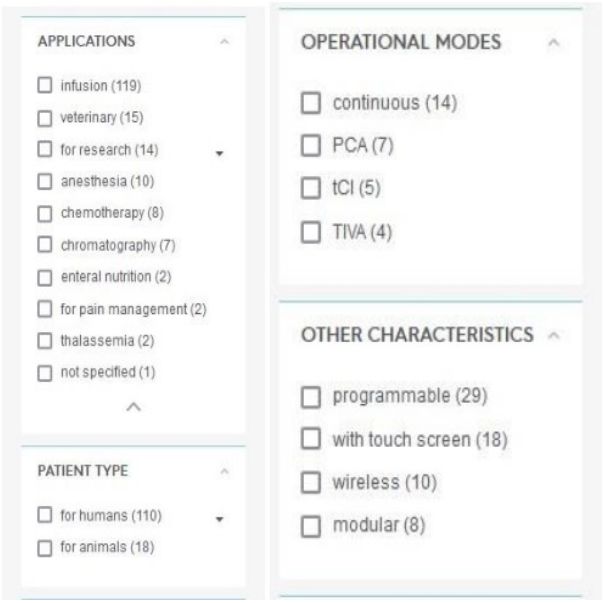

Fig. 6: Options for Pumps in MedicalExpo.com [8]

2) Customer Archetype: The Smart Motor Syringe System has many established competitors in the market but none have a portion over 40%. All of these competitors have costly devices nearing $5000 USD but come with features such as failure alarms, custom fluid capacity syringes, unique flow patterns, and battery backup power. However, after viewing a handful of device data sheets on MedicalExpo.com [8], very few competitors provide testing data or flow rate tolerances. With our main customer being research oriented, the sensitivity of the pump's output is important. I believe bolstering and advertising the engineering requirements on top of the SMSS's steady output as the minimum viable product will carve out a spot in the syringe pump market.

The table below provides three examples of the customer base.

TABLE I: Customer Archetype

Customer	Description	Reason	Product Use
Research Student	This group is likely to adopt new technologies with an emphasis on easy-use DIY and low cost.	This group aims to utilize the minimum viable product aspect while remaining open to learning experiences around DIY and emerging tech. The ability to buy multiple units and sync them is important for testing data.	This group will utilize this product frequently but not in a commercial setting. I can expect his product to remain relevant for a long period until replacement over the course of a decade.
Remote Medical Worker	These customers are looking for a robust solution that can be torn open and repaired. Cost is likely to be a factor.	Practicing medicine in remote locations means that supplies are limited. The need for medical attention will still require medical equipment.	This group will likely run the equipment into the ground and replace parts themselves. Cost is a strong factor as well as robustness and open-source likeness
Third-World Medical Center	These customers care most about the cost of the product as well as the functionality. MVP and functionality are critical	Currency may not be equal to USD and require a large portion of the budget to buy basic medical equipment.	[Same as above]

The following table outlines competitors in the market. It should be noted that most companies are overseas and have a large pool of products related to the medical field. This list is not shy of catheters, IV fluids, syringes, thermal components, 3-D imaging machines, and other augmented devices. One example is Norgren, a large Switzerland company. They have an international presence here in the US that shouldn't be excluded when researching domestic markets in the US. I should mention that it is not easy to track companies as a whole entity, but rather its branches and partners. Wellead is an example of a company here in the US that is difficult to contact or buy from locally. I believe a portion of these companies have third parties to sell and establish their branding within the medical market.

The table below provides four examples of competing companies.

TABLE II: Market Competitors

Market Leader	Description
COMeN [8]	Specializes in veterinary medical devices (Located in China)
wellead [8]	Specialized in a wide variety of syringe pumps and medical equipment. One of the top 10 medical companies in this field. (Located in USA)
MedRena [8]	A small company with a good product line. (Located in China)
NORGREN [8]	A pipetting company that makes channel pumps and syringe pumps. (Located in Switzerland)

3) Marketing Requirements: The SMSS's marketing requirements are derived from the business model and marketing data model in Appendix D Figure 47/48 with Professor Ben Hawkins. In depth, Dr. Hawkins's technological needs are a large influence on the project. That being said, the SMSS stands to be a marketable product in the medical sector.

TABLE III: Marketing Requirements

#	Marketing Requirement	Justification
1	High precision/accuracy flow rates	Constitutes the use of automation, to avoid human inconsistencies and save labor.
2	Programmable flow rates	Customers need various flow rates for different applications.
3	Modular	Customer's require multiple pumps to be controlled in a single project while being able to change setup and hierarchy.
4	Affordable cost (Priced $500)	This cheaper price point appeals to budget-concerned customers
5	Easy User Interface	Users should enjoy using the device.
6	Reliable	Critical work depends on this device

The highest priority of the project is to produce high precision/accuracy flow rates from a standard syringe. The customer, Dr. Hawkins, requested that the pump perform to the same tolerance as existing products while avoiding impulse problems. This requirement does inadvertently imply the use of water in the syringe pumps: the viscosity of the fluid will change the flow rate and the accuracy of the device.

The next highest priority is programmable flow rates: allowing the user to achieve a desired flow rate from the available hardware in a jog or timed manner. This requirement does also inadvertently imply that the only mechanical signal acting on the syringes are unit steps that only produce constant flow. Additional mechanical signals such as a ramp, triangle, or exponential function are not within the scope of this version of the SMSS.

A modular marketing requirement states the user has ability to use a single pump or scale the project up to three pumps. In another frame, the SMSS function without a complete set of devices.

An affordable price point is market disrupting while outperforming the competition with other higher priority qualities. An affordable cost also dictates a lower cost per iteration of the project; this student project will continue to build in complexity without inflating in cost.

An easy interface is a key point as the user shouldn't be required to code or write scripts to use the device. A low learning curve for the device keeps the SMSS competitive and relevant with current Cal Poly research projects outside the electrical engineering department.

Reliability is the last marketing requirement and it is last because the SMSS is still in a prototype state. Reliability for the SMSS is hardware robustness and resistance to various types of noise in the field. This could be interpreted in many ways: study housing, noise rejection, operating in partial failure modes, etc.

B. Engineering Requirements

The engineering requirements seen in the table below are derived from the marketing requirements in Appendix D Table III. In order to meet the market requirements of the customer, these engineering requirements were tested and validated before release. This list is not all encompassing,nor in order of priority, but does outline the minimum viable product to remain competitive in the market. It should also be noted that the requirements derived here are set for this specific version of the SMSS and not guaranteed to hold between generations.

Verification of these requirements covers a wide array of testing setups and is discussed in the Testing section of the paper.

TABLE IV: Engineering Requirements

Market Req.	Engineering Req.	Justification
4	Total production shouldn't exceed $500	This covers cost (with net profit) while remaining competitive in the low-end market
1,6	The pump shall hold a mini -mum flow rate of 0.1 $\frac{mL}{hr}$ within $\pm 0.01\frac{mL}{sec}$ of target rate.	Similar products have this tolerance and the application of the pump needs to remain relevant for medical purposes
1,6	The pump shall hold a maximum flow rate of 100 $\frac{mL}{hr}$ within $\pm 0.01\frac{mL}{sec}$ of target rate.	Similar products have this tolerance and the application of the pump needs to remain relevant for medical purposes
4	The pump shall not contain proprietary software or hardware.	I want to avoid costly parts that could be difficult to alter.
3	The pump must be able to house syringes of size 2mL to 20mL	Dynamic range of syringes is necessary for compatibility of products in the field.
2,3	The hub controller must be able to handle 3 pumps with same flow rates.	Scaling to 3 pumps is defined by the customer
5	The controller shall have a GUI.	The product needs to be useful for a wide audience
3,4	The drive train and housing of the pump can be 3D printed.	The product should remain available to a wide range of personnel and institutions.
1,2,5	The controller must be able to output digital log information on output flow rates: 255 Samples $\in$ (1min, 30min, 1hr 6hr, 24hrs, 2days, 1wk)	The product needs to be able to provide researchers with data they can analyze.

C. Scheduling

The current GANNT chart on the SMSS is seen in Appendix D Figure 49. This GANTT was utilized to the best of its predictions but was not a living document for the duration of the quarter.

D. Bill of Materials

The SMSS project ideally requires funding for parts and labor. The chart in Appendix D Figure 50 outlines the current status of the market pricing as of March 2022 and the stockroom of this continuous project. Furthermore, my contribution to this project is highlighted in orange and added to the overall cost of the project. The total funding required for the system is below $500 USD with some items missing documentation from previous work.

Additional man hours have been added for adequate time to debug and perform software validations. It was increased from 20 hours a week to 45 given the condensed time frame and large portions of software.

IV. System Design

The Smart Motor Syringe System is a mixture of both hardware and software. The following section discusses both aspects of the device in detail.

A. System Hierarchy

The SMSS has many layers to its complexity. For the sake of clarity, a functional decomposition is illustrated below.

1) Functional Decomposition (Level 0) : In the highest level of the SMSS, the black box diagram of the system is comprised of three inputs and three outputs. The inputs of the device are: 120VAC from a power outlet, a single or series of syringes, and user requests from the GUI. The outputs are: linear force placed on the plunger of the syringe, a GUI display for the user, and output flow rate data.

The user will load a syringe into the device, supply power and interact with the GUI by enter the requested data from the SMSS on a keyboard. The requested data for each pump includes the volumetric flow rate, radius of the syringe, capacity of the syringe, and the duration of the device. Upon starting, the device will run the motors and supply enough linear force to produce the requested flow rate. Depending on the user selection, a stop can be time sensitive or triggered manually.

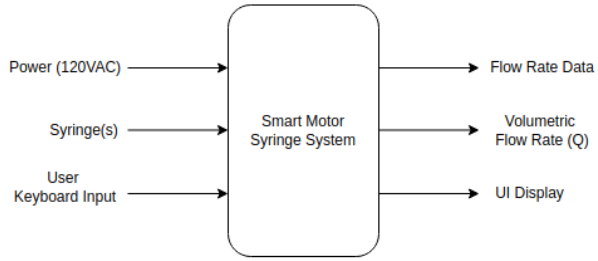

Fig. 7: Level 0 of the SMSS

TABLE V: Design Level 0

Market	SMSS
Inputs	• Power, 120 VAC • Syringe $\in$ [1mL, 30mL] • User input from keyboard
Outputs	• Volumetric flow rate (Q) from syringe • Flow rate data • User interface display
Functionality	After supplying power and water-filled syringe(s), the user will interact with the GUI and enter hardware metrics (on a keyboard) for each pump. After starting the machine, it will produce linear force on each syringe along with measured flow rate data from each device. The GUI will visually displayed for the user on a monitor.

2) Functional Decomposition (Level 1) : The next level of hierarchy, Figure 8, further breaks down the system into individual sections. The motor power supply (VMOT), digital power (VDIG), and syringes are split respectively for simplicity. The hub controller is a GUI driven controller designed to send commands to each of the pump controllers. It is separated from the other pump controllers to highlight its higher hierarchical place as well as it's interaction with user inputs/output devices. It also acts as a main device

with the pump controllers assuming the sub role for I2C communication protocol; the hub controller also supplies VDIG to all of the individual pump controllers. Lastly, the pump controllers are the electrical and mechanical hardware used to translate electrical quantities into linear force. This is a large set of equipment that demands an additional level of decomposition to break down cleanly; this is seen in Figure 9.

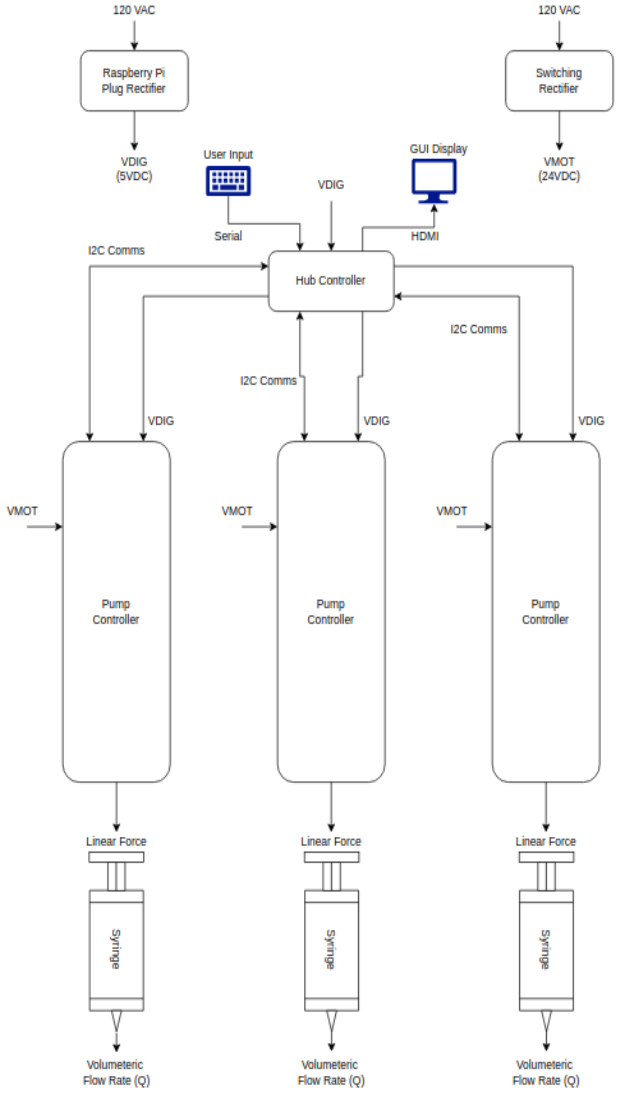

Fig. 8: Level 1 of the SMSS

TABLE VI: Switching Power Supply in Level 1

Module	Switching Rectifier
Input	120 VAC
Output	24VDC (VMOT)
Function	Convert the 120 VAC wall source and provide the stepper motors with 24VDC and +1A continuous current.

TABLE VII: Digital Power Supply in Level 1

Module	Plug Rectifier
Input	120 VAC
Output	5 VDC (VDIG)
Function	Convert the 120 VAC wall source and provide the Raspberry Pi with 5VDC and +1A continuous current.

TABLE VIII: GUI Display in Level 1

Module	Display Monitor
Input	HDMI from hub controller
Output	Visual information for user
Function	The monitor displays the GUI for the user

TABLE IX: User Input in Level 1

Module	Keyboard
Input	User key press
Output	Serial signal for controller
Function	Translates user key presses into digital signals

TABLE X: Hub Controller in Level 1

Module	Hub Controller
Input	•5 VDC •User data from Keyboard •I2C response from motor controller •Measured flow rate data from motor controller
Output	•5 VDC •GUI display on monitor •I2C commands for motor controller •Compiled flow rate data from each pump
Function	Directs the state of each motor controller unit, translates user requests into usable data, and stores measured flow rate data from each unit.

TABLE XI: Syringe in Level 1

Module	Syringe
Input	•Linear force •Water
Output	Volumetric Flow (Q)
Function	Translates linear force into flow rate

TABLE XII: Pump Controller in Level 1

Module	Pump Controller
Input	•5 VDC (VDIG) •24VDC (VMOT) •I2C commands from Hub
Output	•Linear force on syringe •I2C response to Hub •Measured flow rate to Hub
Function	The motor controller and plant system are responsible for translating the hub commands into a linear force as well as supply the hub with response information and flow rate data.

3) Functional Decomposition (Level 2): Level 2 of the functional decomposition focuses on an individual pump controller and the electrical/mechanical components inside. The top block in Figure 9 is the motor controller. For this version of the SMSS, the motor controller is an Arduino Nano that is connected to the hub controller with I2C and digital power VDIG. The motor controller controls the local logic flow and adjusts the setting/performance of the stepper motor (indirectly through the stepper driver). The stepper driver translates the low power signals from the motor controller into a high-power signal usable by the stepper motor. Lastly, the ball screw translates the rotation of the stepper motor into linear velocity.

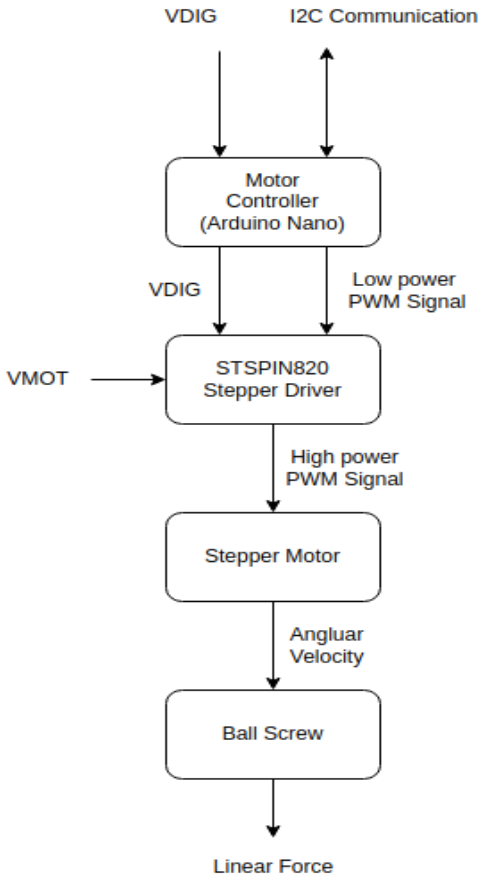

Fig. 9: Level 2 of the Pump Controller

TABLE XIII: Motor Controller in Pump Controller Level 2

Module	Motor Controller
Input	•5 VDC (VDIG) •Bi-directional I2C communication
Output	•5 VDC (VDIG) •Low power PWM signal
Function	The motor controller converts hub controller requests into a PWM signal for the stepper driver. The motor controller collects flow rate data and can act as a standalone device should the user only need one pump.

TABLE XIV: Stepper Driver in Pump Controller Level 2

Module	Stepper Driver
Input	•5 VDC (VDIG) •24 VDC (VMOT) •5v logic level PWM signal
Output	24v logic level PWM signal
Function	The stepper driver takes in a relatively low voltage PWM signal, modulates that signal with VMOT, and outputs a higher power PWM signal with options for step division (micro stepping).

TABLE XV: Stepper Motor in Pump Controller Level 2

Module	Stepper Motor
Input	High power PWM
Output	Angular velocity ω
Function	Converts the PWM pulses into rotational movement.

TABLE XVI: Ball Screw in Pump Controller Level 2

Module	Ball Screw
Input	Rotational Velocity
Output	Linear force (and velocity)
Function	Convert the rotation of the motor into linear force/velocity.

B. Hub Controller

This subsection discusses the hub controller, covering the GUI interface as well as the logic flow. The user interface will be a step-by-step example walk through that covers all of the interactions the user will come across. The logic flow will be presented with flow diagrams and outlines.

1) User Interface: The user interface uses a terminal-based library called Curses or NCurses. This library is already installed on most OS and is usually seen in BIOS menus or other applications requiring a lightweight GUI. It should be noted that curses operates in the terminal, often forgoing the use of a mouse and relying solely on the keyboard for user input. Curses does offer mouse integration but due to time constraints, the keyboard will be the only input device for the hub controller while in the terminal. It is still recommended to use a mouse to interact with the raspberry pi.

Upon powering the hub controller (or raspberry pi for this version of SMSS), a yellow LED will light up. This light is attached to the STANDBY state of the motor controller (Arduino Nano). Table XVII pairs the corresponding color combination of LED to the motor controllers current state.

TABLE XVII: Device State per LED Combination

LED	State
Yellow	Standby
Blue	Data collection ("Datapull") for timed run mode.
Blue + White	Data collection ("Datapull") for jog mode.
Green	Motor running in timed run mode.
Green + White	Motor running in jog mode.

Figure 10 shows the starting condition of three syringe pumps and their yellow STANDBY LEDs. The LEDs located on the three Nano boards do not represent the controller states; only the three rows of LEDs off of the boards are status LEDs. It should be noted that device one is the right-most device, progressing numerically to the left (the left-most device is number 3).

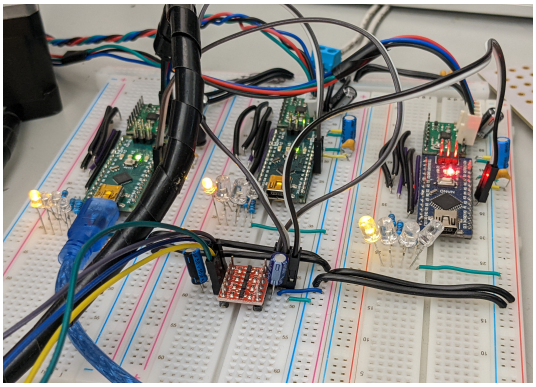

Fig. 10: LED Indicators on Start

Figure 11 shows the command lines to navigate the user to the GUI. The first line, *cd Desktop/SMSS22*, changes the terminal directory to the location of the hub controller's python file. the next line, *python3 Hub.py* runs the python script. It is recommended to resize the terminal window to a larger one before pressing enter.

** The user can press CTRL+z to exit the GUI. **

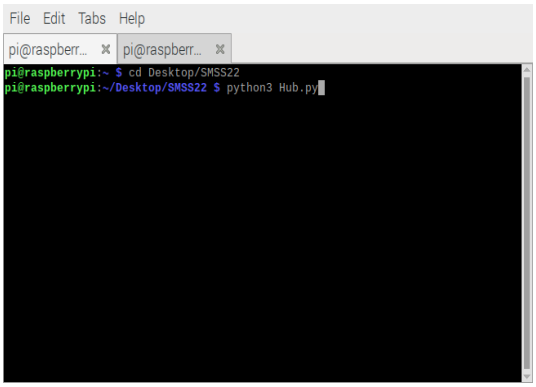

Fig. 11: Running Hub.py in the Terminal

Figure 12 represents the controls for selecting the device run states as well as the direction of the motors in a later menu. The WASD keys are used to move the highlighted cursor left/right with W acting as the enter key.

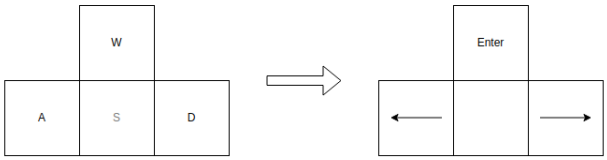

Fig. 12: Controls for Preset Options

Figure 13 shows the start screen of the GUI after selecting how the each device should be run. Notice each device window has its own mode and the white block cursor waiting on bottom near the CONFIRM button. Pressing W here will advance the GUI to data collection. Figure 14 shows the color pattern of the LEDs after selecting the run modes. Notice device 1 has both blue and white LEDs while device two only has blue on. Device three still has yellow on and is put in standby or "OFF" until another cycle.

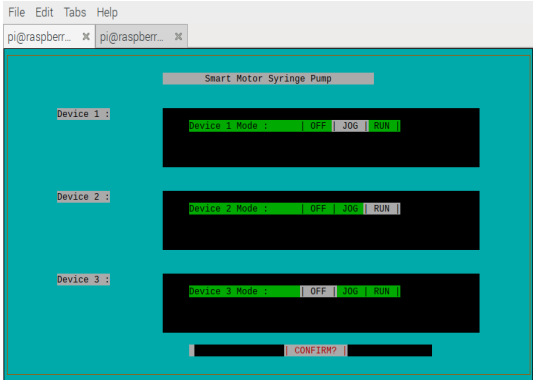

Fig. 13: Selecting Each Run State on the Different Devices

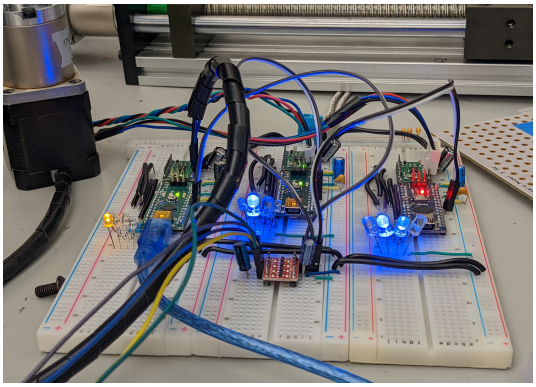

Fig. 14: LED Indicators for Device/State

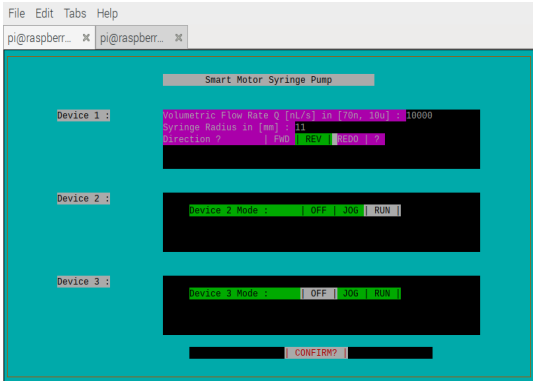

Fig. 15: Entering User Data into the Hub Controller

Figure 15 gives an idea on what it would be like entering information into the hub. The GUI will prompt the user for nominal values and the user will enter then using the number keys and enter. Figure 16 illustrates the keys used when entering parameters. It should be noted that the direction of the motor is still selected using WASD. the option REDO is used to revert back to STANDBY mode for that particular device in order to reassign a run mode. Figure 17 shows example data entered into the GUI with the CONFIRM below awaiting a W for confirmation.

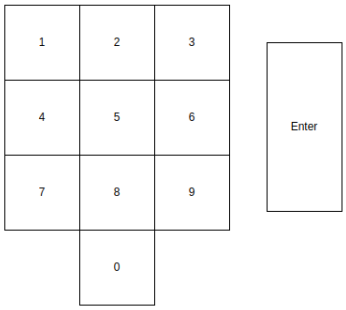

Fig. 16: Controls for Numerical Options

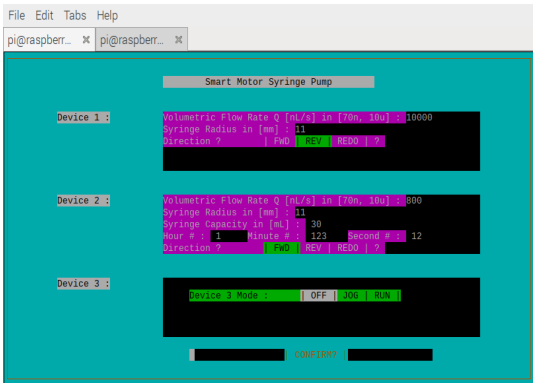

Fig. 17: Concluding Data Pull from User

After confirming the data pull state, the devices will enter the RUN state (Figure 18). The GUI will display the parameters that are being output and the time elapsed if running in run mode. Figure 19 shows the LEDs for each device. Device 1 is in jog and shows green and white lights, device 2 is in run mode and only uses a green LED, and device 3 is still showing a yellow LED because it is still in standby mode.

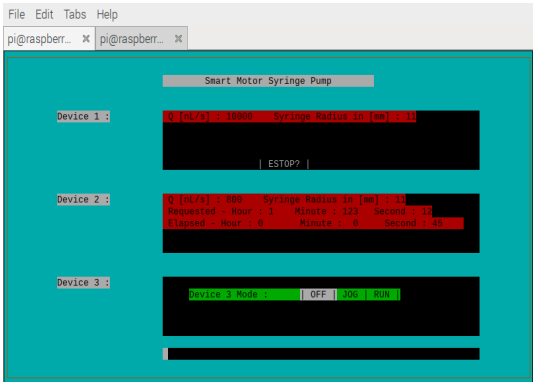

Fig. 18: GUI Run State

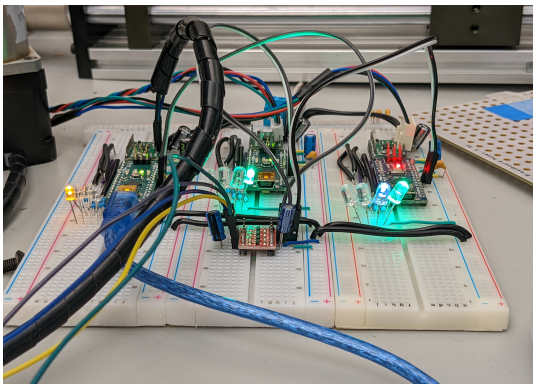

Fig. 19: LED Indicators during Run/Jog State

Figure 18/20 also give information on the estop cursor. Figure 18 shows the estop cursor on device one as "| ESTOP |" while Figure 20 has its estop cursor at the bottom (| ESTOP ALL? |). The first figure will estop the device 1 only while the second figure will stop all of the devices if pressed.

Figure 21 shows the estop controls; they are using WASD but in a different order than the mode selection.

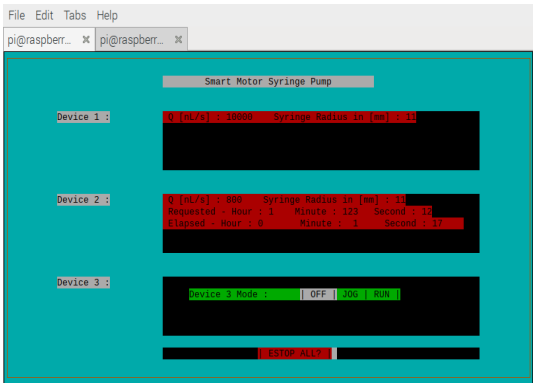

Fig. 20: Run Screen : E-Stop All Devices

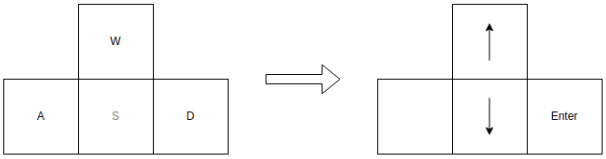

Fig. 21: Controls for E-Stop

Figure 22 shows an example of an estop event. The estop cursor is replaced with a | STOP'D | mark, the motor haulted, and the motor controller returns to the STANDBY state as seen in Figure 23

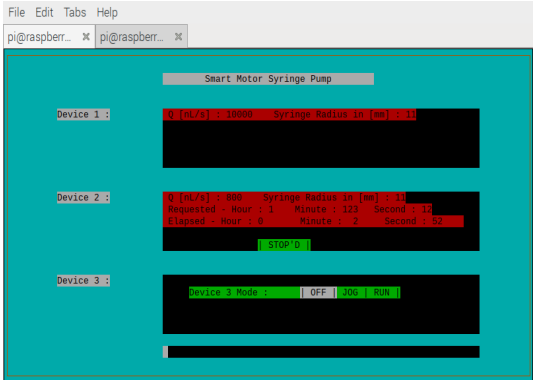

Fig. 22: Run Screen : E-Stop Device #2

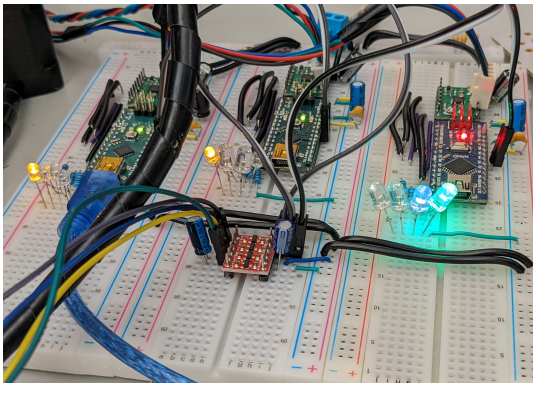

Fig. 23: LED Indicator with E-Stop Event

The hub controller communicates with each motor controller after the user has entered in the requested parameters. No motor controller hardware is the same and as such, each user request is checked my the motor controller to see if the hardware can produce the correct output. If the requested volumetric flow rate is too large for the hardware to output, an error message will appear (Figure 24) and the user will be prompted to enter new parameters. Other error codes exist: Figure 25 is too small a Q value and Figure 26 refers to the run time exceeding the fluid capacity of the syringe.

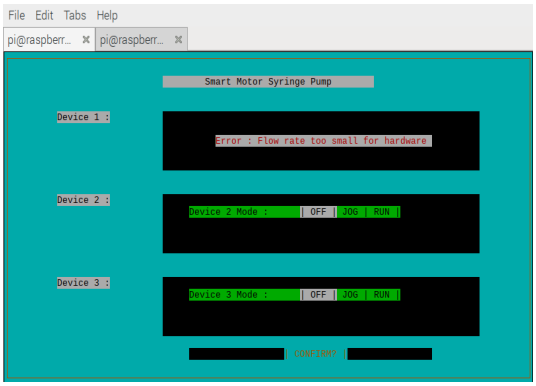

Fig. 24: Error Code : Too Large for Hardware to Output

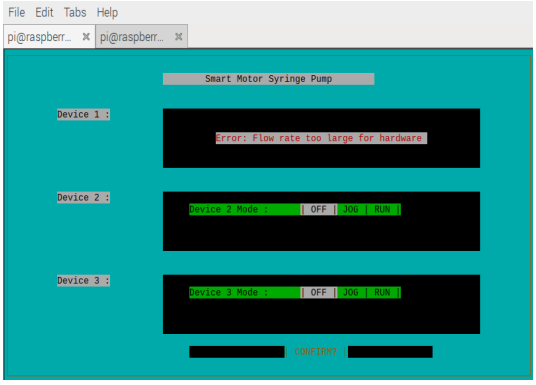

Fig. 25: Error Code : Too Small for Hardware to Output

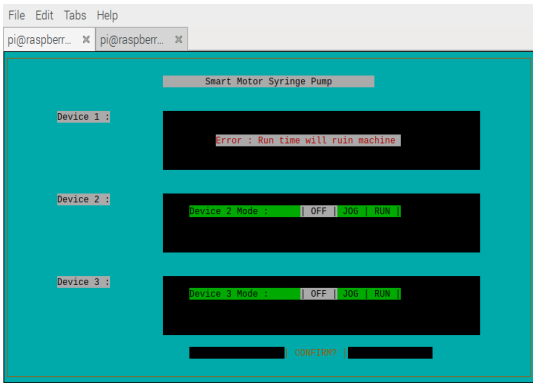

Fig. 26: Error Code : Too Long for Hardware to Run

After all devices have halted, the GUI screen will return to polling for the device run mode and all of the indicator LEDs will revert back to yellow.

2) Logic Flow: This subsection discusses the logic flow of the hub controller. The python code of this controller can be seen in Appendix C.

Looking at Figure 27, the hub controller has four states that it operates in: START, DATAPULL, RUN, and RESET. The START state is written to do a few items: note which devices are active on the I2C bus, reset the motor controllers to their STANDBY state, and ask the user which run mode is proper for each device. After all active devices have selected a run mode, the hub transitions to DATAPULL. DATAPULL will prompt the user for data parameters (volumetric flow rate Q, syringe radius, syringe capacity, duration, and motor direction) and send that data to the individual motor controllers. Should the user request be outside the operating range of the hardware, the motor controller will send back an error code that will be displayed on the GUI. The program will loop until all devices are error free and the system is ready to progress into the next state. The RUN state will turn on the stepper motors and poll for a stop condition. Should the user select an estop in the GUI or the motor controller reach its max time duration, the motor controller will shut off the motor and the hub will reset for that device. After all devices have reached a stop condition, the hub will proceed to the RESET state to clear all user parameters in the hub/motor controllers. The hub will return to the START state and STANDBY in the motor controllers.

It should be noted that the GUI referenced in the previous subsection is the same here and can be referenced as such.

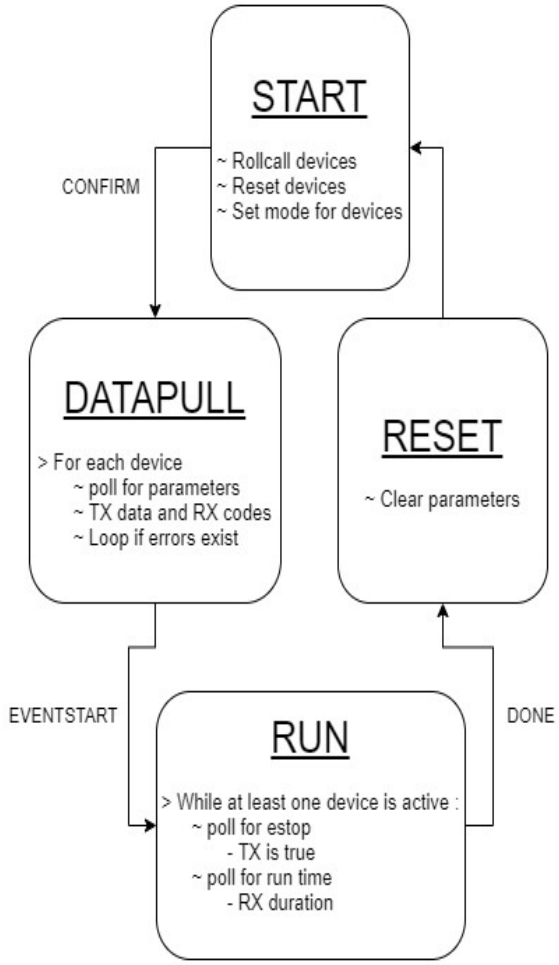

Fig. 27: Top Level Flow Chart of Hub Controller

C. Motor Controller

This subsection discusses an individual motor controller in the SMSS in three different sub-sub-sections. The first will discuss the characterization of the system, the second will map the control flow, and lastly there is a section about the I2C communication between the hub and motor controllers.

1) Plant Characterization: Starting with the syringe, the mechanical conversion that takes place in the syringe is as follows in Figure 29. The volume of fluid displaced by the syringe plunger is proportional to the speed at which the plunger is moving. This plunger speed (V) is applied across the plunger's surface area, producing a volumetric flow $\frac{m^3}{sec}$. This unit can be converted to $\frac{L}{sec}$ with unit conversion.

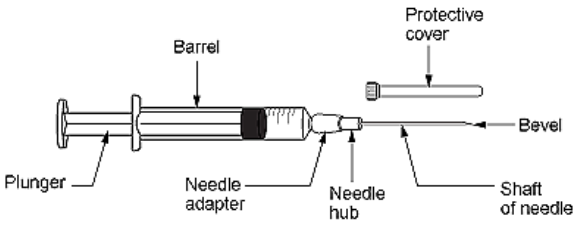

Fig. 28: Components of a Syringe

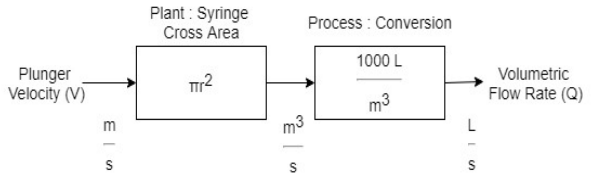

Fig. 29: Plant Conversion of Syringe

After understanding how the linear velocity V is translated into a volumetric flow rate Q, lets discuss how the gearbox's RPM is converted into a linear velocity. Figure 30 illustrates how the RPM from the gearbox is translated to a linear velocity on the syringe plunger. The gearbox shaft is coupled to the screw rod at a 1 to 1 ratio; this segment does not require further analysis as no slip was witnessed. The interaction of the screw rod and the pillow block does however : the pillow block is attached to the frame of the ball screw kit and must be engaged with the screw rod to move. The helical spiral in Figure 31 represents the screw rod that engages with the threaded bearing in the pillow block. The linear speed of the pillow block is dependant on the the rotational speed as well as a few factors. The position of a single particle on the helix is represented as r(t), with respect to time.

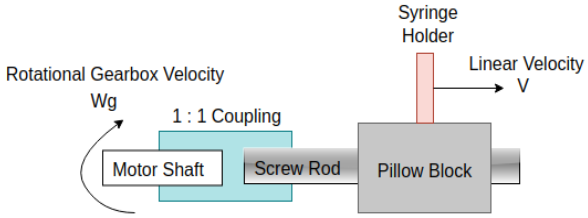

Fig. 30: Translation of Angular Velocity to Linear Velocity

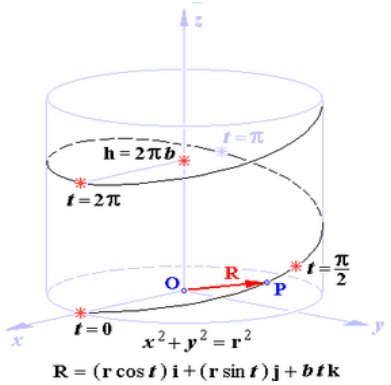

Fig. 31: Cylindrical Helix Model for Ball Screw

$$Let\ r(t) = \rho\cos(\overrightarrow{\omega_g}t)\,\hat{\mathbb{A}}_x + \rho\sin(\overrightarrow{\omega_g}t)\,\hat{\mathbb{A}}_y + \gamma\overrightarrow{\omega_g}t\,\hat{\mathbb{A}}_z \quad [m] \quad (1)$$

Where ρ is the screw rod radius and γ is the rod's pitch. The rod's pitch is the z-axis distance between points of the same xy angle.

To find the velocity of the particle with respect to time, the derivative of r(t) is taken.

$$\overrightarrow{v}(t) = \frac{d}{dt}r(t) \quad \left[\frac{m}{s}\right] \quad (2)$$

$$\overrightarrow{v}(t) = -\rho\overrightarrow{\omega_g}\sin(\overrightarrow{\omega_g}t)\,\hat{\mathbb{A}}_x + \rho\overrightarrow{\omega_g}\cos(\overrightarrow{\omega_g}t)\,\hat{\mathbb{A}}_y + \gamma\overrightarrow{\omega_g}\,\hat{\mathbb{A}}_z \quad \left[\frac{m}{s}\right] \quad (3)$$

Focusing attention on the Z-axis movement :

$$\begin{aligned} \overrightarrow{v_z}(t) &= \gamma\overrightarrow{\omega_g} \quad \left[\frac{m}{s}\right] \\ \overrightarrow{\omega_g} &= \frac{\overrightarrow{v_z}(t)}{\gamma} \quad \left[\frac{rad}{s}\right] \\ \overrightarrow{RPM}_{gearbox}(t) &= \frac{2\pi\overrightarrow{v_z}(t)}{\gamma} \quad \left[\frac{Rev}{s}\right] \end{aligned} \quad (4)$$

The final equation relates the rotational speed of the gearbox to the linear velocity of the pillow block. Figure 32 is the plant of the system from this point. From here, relating individual steps of the stepper motor to the output RPS is the final step in characterizing the system. The STSPIN820 motor driver can output up to 256 microsteps, allowing the driver to further divide the input PWM freq at the cost of motor torque [9]. The stepper motor itself markets 200 steps per revolution; both motor step and the driver's microstep will alter the motor's speed. Figure 33 depicts the addition of both step properties as well as the final representation of the hardware system.

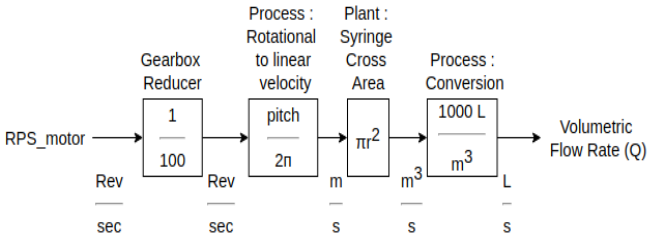

Fig. 32: Translation of Motor Speed to Flow Rate

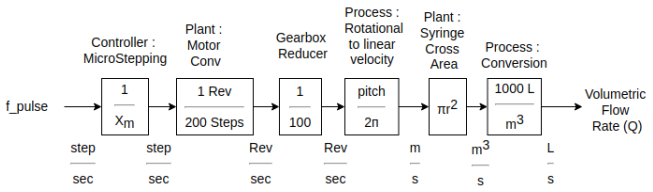

Fig. 33: Complete Hardware Characterization of Pump

Working backward to find the appropriate frequency, f_{user} can be defined as :

$$\begin{aligned} f_{user} &= X_m * \frac{400\pi NQ}{1000\gamma\pi r^2} \\ &= X_m * \frac{2NQ}{5\gamma r^2} \end{aligned} \quad (5)$$

where N is the gear ratio, γ is the pitch of the ball screw, X_m is the microstep divisor, f_{user} is the frequency derived from the user requested Q, and r is the radius of the syringe plunger.

From experimentation, it was found that the minimum frequency the stepper motor should operate at or above is 120 Hz ($f_{min} = 120Hz$). With this standard, finding the proper microstep size is a simple equation :

$$\begin{aligned} f_{user} &\geq f_{min} \\ X_m * \frac{2NQ}{5\gamma r^2} &\geq f_{min} \\ X_m &\geq \frac{f_{min}}{\frac{2NQ}{5\gamma r^2}} \\ \therefore X_m &\geq \frac{5\gamma r^2 f_{min}}{2NQ} \end{aligned} \tag{6}$$

The range of Q is limited to the frequency range of the stepper motor and was found experimentally as well. Both maximum and minimum Q values would stall the stepper motor. It was also noticed that frequencies above and below the range could move the motor but not reliably. The reliable range is :

$$Q \in [10n, 16\mu] \left[\frac{L}{sec}\right]$$

2) Control Flow: The individual pump/motor controllers have the same code structure but can have different hardware parameters (syringe radius, ball screw radius, etc). Moreover, each controller can be placed into a I2C mode - the hub controller will dictate the flow of user info, or in serial mode - the Arduino IDE serial terminal will prompt the user for information. It should be noted that the serial method is device specific and wont control more than one pump controller. The serial mode is also referred as manual mode because it is easy to tune parameters for an individual pump without the use of an additional controller. To use either mode, comment out the unused setting as in Figure 34.

Both modes of the motor controller use different communication protocols but are similar in state structure. Figure 35 represents the highest level of flow for the motor controller : 6 states that will continue to loop until power is cycled. The controller will start and stay in STANDBY until the controller has received a START/JOG request. After leaving, the controller will go in to either DATAPULL_S (for timed duration) or DATAPULL_J (for jog) to retrieve the hardware parameters from the user. This data can be from I2C or serial depending on which portion of code had been enabled. After checking for errors, the device will proceed to the appropriate RUN state. Once the device has received an ESTOP or time-dependent FINISH command, the device will reset all variables in ESTOP and proceed to wait in STANDBY. This top level is almost identical to the hub controller but has many lower level differences.

Lower level flow charting can be found in Appendix D.

```
// Test Variables (COMMENT OUT TO DESELECT) -- DONT FORGET
#define TESTING                 // Comment to cancel testing
//#define SERIAL_COMMS            // Comment to cancel serial
#define I2C_COMMS               // Comment to cancel I2C cc
//#define STSPIN220               // Comment out to NOT use t
#define STSPIN820               // Comment out to NOT use the
```

Fig. 34: Tool selection for the Motor Controller

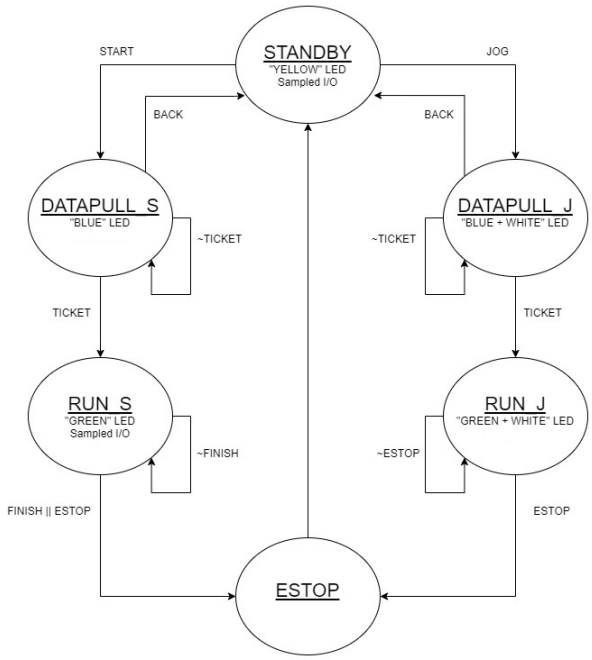

Fig. 35: Top Level Flow Chart of Motor Controller

3) I2C Communication Between Controllers: The hub controller assumes the position of the main device with the pump controllers acting as sub devices. The sub devices don't communicate with each other; only the hub controller sends a receive (MOSI) or request (MISO) command. The hub controller is a raspberry pi, which a logic high of 3.3V, and the pump controller, arduino nano, has a logic high of 5v. With a difference in logic level voltage, a bi-directional logic converter was used in order for the devices to communicate to each other without hardware damage. For simplicity, a Sparkfun MOSFET converter was used as seen in the Appendix D schematics.

The data sent and requested by the main controller was often over the single packet 255 bit limit. To overcome this, data packets are processed to handle numbers less than 16,777,215. This is done by sending factors that are multiplied by 256 as well as remainders that are summed with the total number. The trade off for this solution is the extra time it takes to transmit three extra packets of information.

If the number is large enough to fill the second factor,

$$Number = 256 * (256Fact_2 + Remain_2) + Remain_1$$

else :

$$Number = 256Fact_1 + Remain_1$$

where each variable above represents a single packet.

V. Testing

This section outlines the process for verifying the engineering requirements (Table IV) of the SMSS. The financial requirement, keeping the SMSS under $500 USD is verifiable by viewing the Bill of Materials in Appendix D Figure 50. This requirement is met for this generation of the SMSS because the cost was under $500 USD.

The second and third requirement will be validated in the next section below; the fourth requirement can be verified by noting the libraries and hardware used. The code (Appendix C) that was used is open source online and the hardware is not locked by physical/digital means. Both Arduino and Raspberry Pi devices are designed to be altered down to the copper or the kernel.

The fifth requirement was verified visually by inserting different syringes into the housing. The recorded range of syringes was 1mL to 30mL, going beyond the requirement.

Requirement six, concerning 3 pump controllers, is also verified in the subsection below. That requirement requests an analysis of three separate devices and their output flow rate.

Requirement seven is the GUI requirement and it is met by viewing the hub controller section earlier on in the paper.

Requirement eight, requesting that the drive train and pump housing be 3D printed, has fallen short of its goal. The drive train is a metallic ball screw that must be metal for friction and wear purposes. 3D printing this piece would require additional work for a lesser component. The housing of the entire system has also yet to be 3D printed because the footprint of the SMSS will continue to decrease. It was not fit to enclose the entire system at this stage of prototyping. Printing a piece that hold the ball screw and syringe together has proved to be a value component in this stage of the SMSS.

The last requirement was also short of its goal. This generation of the SMSS was unable to integrate viable feedback in the system. Different feedback methods (potentiometers, resolvers, pressure sensors) were considered but were left out due to budget and time constraints. Without feedback, output data couldn't be produced and sent to the hub controller.

A. Verification Method for Hardware

The output flow rate of the SMSS was measured with a LabSmith uProcess kit. This kit measures the pressure inside a CapTile cavity and was the only method used. To use this device properly, the user must setup the device as shown in Figure 36 and note physical measurements of the differential tubing. This tube's inner diameter and length are important in translating the pressure measurement into a measured volumetric flow rate. While the SMSS is applying pressure on the plunger of the syringe, water is flowing through the filter and into the three-way CapTile. The uPOS senor then measures the pressure at its highest potential at the intersection in reference to the lowest potential point at the other end of the differential tube. The pressure across the differential tube is the measured result $\triangle P$ $[kPa]$. The change in pressure can be translated into a volumetric flow rate with the following equation. It should be noted that the GUI has Q in terms of nL/sec, hence the conversion to nL/s here.

$$\begin{aligned} \triangle P &= P_{meas} - P_{atm} \quad [kPa] \\ &\approx P_{meas} - 1 \quad [kPa] \end{aligned} \tag{7}$$

$$\begin{aligned} Q_{deriv} &= \frac{\pi\,(\triangle P)\,D^4}{128\mu\,\mathbb{L}} \left[\frac{m^3}{sec}\right] \\ &= \frac{\pi\,(\triangle P)\,D^4}{128\mu\,\mathbb{L}} * \frac{10^3 L}{m^3} * \frac{10^9 nL}{L} * \frac{10^3 Pa}{kPa} \left[\frac{nL}{sec}\right] \\ \therefore &= \frac{\pi\,(\triangle P)\,D^4}{128\mu\,\mathbb{L}} * 10^{15} \left[\frac{nL}{sec}\right] \end{aligned} \tag{8}$$

where

- $\triangle P$ is the change is pressure measured by the uPOS (kPa)
- D is the inner diameter of the differential tube (m)
- μ is the dynamic fluid viscosity of water at 21 C (Pa * sec)
- $\mathbb{L}$ is the length of the differential tube for which the pressure change takes place (m).
- Q_{deriv} is the volumetric flow rate derived from the change in pressure measurement (nL/sec). This variable will be compared with the Q value entered by the user in the GUI.

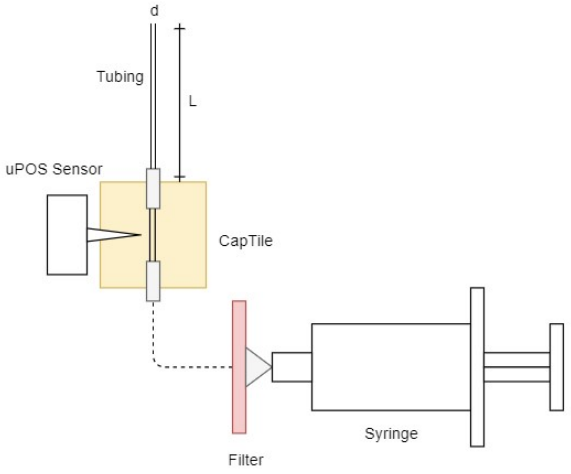

Fig. 36: Verification Model

B. Results

Using the setup in the sub-section above, a variety of nominal values were sampled. This method was used to experimentally find the operating range of the SMSS.

1) Ceiling of Q Range: The ceiling of an individual syringe pump was initially verified visually by witnessing the spin of the motor while coupled with a syringe. Later in the project, this was properly verified using the uProcess kit.

All other values above 16μL/sec were not repeatable and ended up stalling the motor. It should be noted that this real max value of 16μL/sec is far below the set engineering requirement; below both the maximum and minimum requirement. Here are the results :

TABLE XVIII: Parameters of Ceiling Test

Variable	Value	Unit
Syringe #	S1	-
Motor #	M1	-
Arduino #	A1	-
Motor Controller #	STSPIN820	-
Sensor	uPOS250-T116	-
Pipe Length $\mathbb{L}$	0.0820	m
Pipe Inner Dia D	0.15	mm
Dynamic Fluid Viscosity μ	0.000975 @ 20°C	Pa*sec
Syringe Radius(GUI)	11	mm
Motor Direction	Forward	-

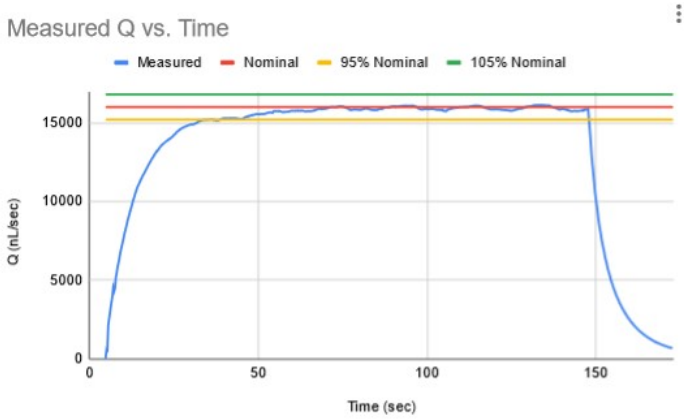

Fig. 37: Coarse Measurement for Nominal Q : 16000 nL/sec

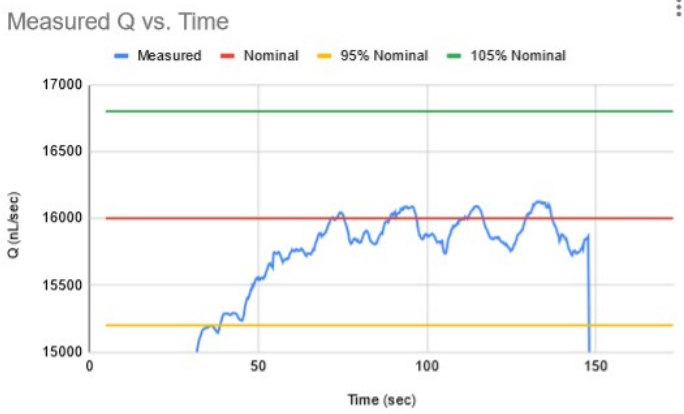

Fig. 38: Fine Measurement for Nominal Q : 16000 nL/sec

Figure 37 is the entire capture of processed data. There is an exponential rise to 16000 nL/sec and it holds around that number until the pump is turned off, near 150 seconds. The coarse figure appears to hit and stay on the nominal value however, Figure 38 shows differently. The same value lines are present but the measured result is oscillating around the nominal value. It should be noted that the water flowing out of the differential tube is jet-like and not droplets.

2) Floor of Q Range: The smallest output was verified using the uProcess kit as well. The minimum engineering requirement (100$\mu L/sec$) is higher than the experimental minimum (10nL/sec).

TABLE XIX: Parameters of Floor Test

Variable	Value	Unit
Syringe #	S1	-
Motor #	M1	-
Arduino #	A1	-
Motor Controller #	STSPIN820	-
Senor	uPOS250-T116	-
Pipe Length $\mathbb{L}$	0.0820	m
Pipe Inner Dia D	0.15	mm
Dynamic Fluid Viscosity μ	0.000975 @ 20°C	Pa*sec
Syringe Radius(GUI)	11	mm
Motor Direction	Forward	-

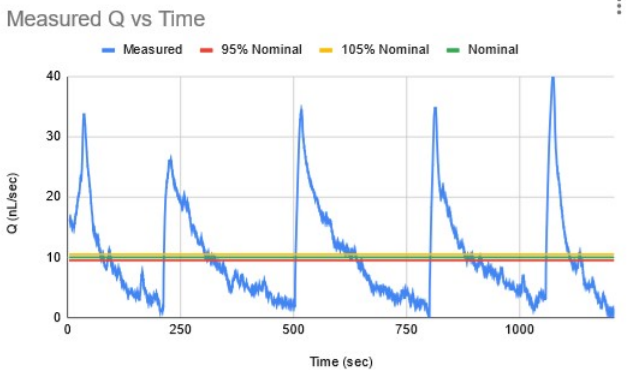

Fig. 39: Measurement for Nominal Q : 10 nL/sec

Fig. 40: Measurement for Nominal Q : 70 nL/sec

Figure 39 illustrates the volumetric flow output when the nominal Q value is 10 nL/sec. The large pressure spikes seen aligned with droplets falling from the differential tube. When the droplet was building in size on the tip of the tube,

the pressure would decrease. When the droplet fell, there was a large spike in pressure. It is difficult to say whether the pump could hit this nominal value accurately.

Figure 40 has similar spikes to the previous figure but are less pronounced. The pressure spikes are related to the physical droplets as well; the syringe pump can hit this Q value reliably and largely stay within ±10% untuned.

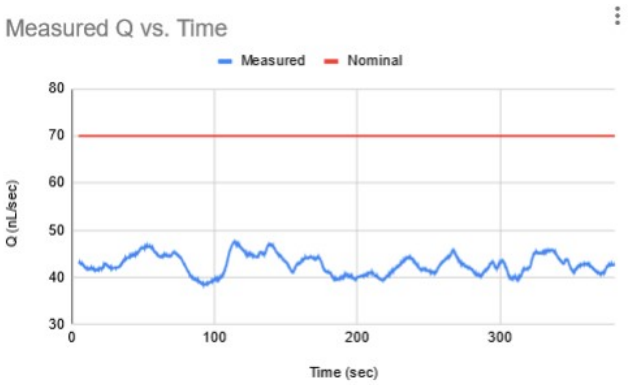

Fig. 41: Special Measurement for Nominal Q : 70 nL/sec

From this point, the data suggested that the exiting droplets were having an impact on the low Q data. Figure 41 shows the same nominal flow request but with the differential tube under water in the collection dish. The measured data is far from the nominal value, using the assumption from equation (7), but the large spikes are smoother than the previous test.

3) Additional Data within Range: Additional tests were made within the Q range described above. These tests are useful in understanding unforeseen problems in the system.

TABLE XX: Parameters of Additional Data

Variable	Value	Unit
Syringe #	S1	-
Motor #	M1	-
Arduino #	A1	-
Motor Controller #	STSPIN820	-
Senor	uPOS250-T116	-
Pipe Length $\mathbb{L}$	0.0820	m
Pipe Inner Dia D	0.15	mm
Dynamic Fluid Viscosity μ	0.000975 @ 20°C	Pa*sec
Syringe Radius(GUI)	11	mm
Motor Direction	Forward	-

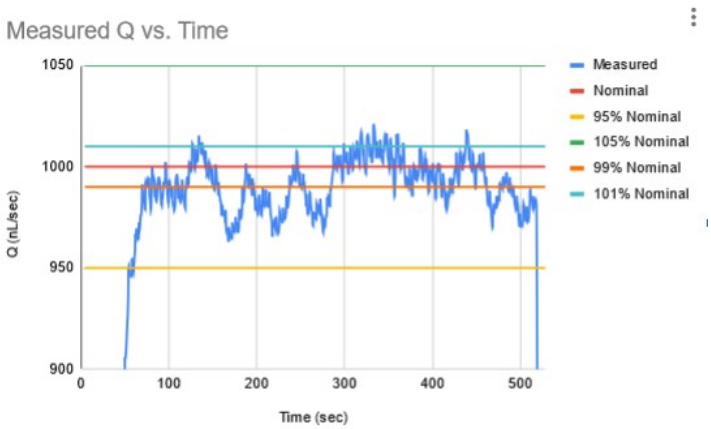

Fig. 42: Measurement for Nominal Q : 1000 nL/sec

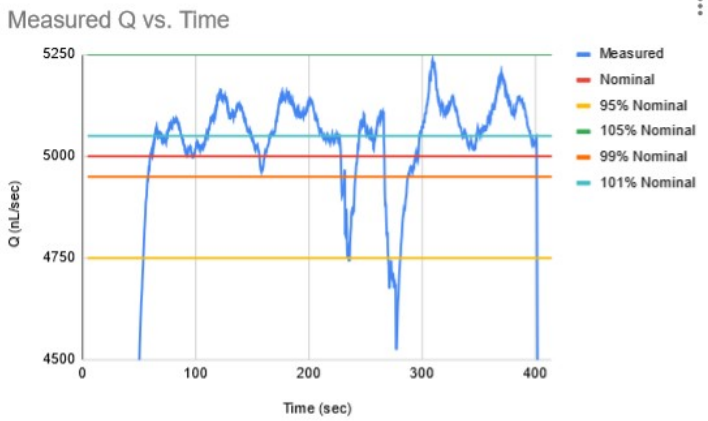

Fig. 43: Special Measurement for Nominal Q : 5000 nL/sec

Figure 42 is additional data within the Q range. There are inconsistencies between 200 and 300 seconds while running the system naturally. Figure 43 is another example of inconsistencies : the large dips between 200 - 300 seconds are the outcome motor failure. For one reason or another, the motor was stalling during testing, emitting a large hum and vibration. The cause for this is still unknown and output returns to the typical pattern after this event.

C. Analysis

After viewing the data, there is confidence that the pump system can achieve nominal rates within 70-16000 nL/sec, but a few problems have to be addressed. The measurement method used needs further investigation into the droplet phenomenon: does surface tension or falling droplets have an effect on low Q output? One correction could be to record data with the exit tubing under water; further testing is needed.

Other issues came from the 3D printed parts attached to the ball screw assembly: the amount of flex in these parts gave inaccurate readings for the initial round of tests. There were re-prints of existing parts with higher infill percentages as well as newly drafted parts that were over engineered to avoid flexing under pressure. The original set of prints did

end up breaking over time and were recorded deafening the pressure data as their internal structure began to fail. It would be recommended to replace the existing 3D printed parts with materials that has little flex.

Another issue came from the motor or motor controller: there were periods where the motor struggled to output the correct frequency. This can be seen in the data in Figure 43 as a large trough in the blue measured data. More testing is required to identify the root of this problem.

The idea of fine tuning the motor controllers was conceived early in the planning phase of the project but ended up not happening. It was decided that digitally tuning the controller before addressing the other mechanical / data validation problems would not be the best course of action. The error measured in the data was also above nominal in the lower Q values while below in higher values, making the tuning process more than a simple offset of frequency.

It should be noted that the range of Q dictated by the engineering requirements was not met : the individual pumps that comprise the SMSS are capable of nano-liter or micro-liter flow rates but not milliliter. A future re-assessment of the engineering requirements is needed in order to confirm that the smaller range is beneficial. Fortunately customer Dr Hawkins has expressed that the finer Q range is beneficial.

VI. Conclusion

This generation of the Smart Motor Syringe System demonstrated several new additions to this ongoing project while getting closer to a smooth, nominal volumetric flow rate. The new additions include but are not limited to : the introduction of a hub controller, a simple GUI, and multiple configuration setting for three complete motor controllers.

The hub controller and GUI were successful in directing the user's requests into viable data across all three devices. The user can control up to three devices, run each device manually or timed, get real time error codes, and emergency stop each device individually or all together. While the code has not been exhausted of all edge case bugs, there are still some underlying issues to address. For future endeavors, the python code written can be improved with more modular-based function scripting and the removal of most global variables in the code. Both of these items would improve the readability of the code and potentially increase the performance time. Communication between devices was also witnessed to be slower than initially thought, taking up to 100ms per transmission. Adjusting the I2C speed on both hub and motor controller unfortunately introduced more errors.

Looking at the individual motor controllers, each device performed close to the nominal requested Q. Most of the measured volumetric flow rates came within a 5% tolerance and demonstrated a finer accuracy than previously prescribed in the initial engineering requirements. That being said, there were considerable items that needed addressing. There were unforeseen motor stalls mid-test as well as large pressure spikes when measuring small Q values. This observation suggests that there should be a different testing setup that prevents droplets from forming on the differential tubing. Lastly, the use of 3D printed parts became a problem and contributed to a number of accuracy errors early on. It is strongly recommendation to replace all of the printed parts with metal or a material that doesn't flex. Only after designing parts that were either over-engineered and/or over-filled was there little dampening/delay in the system.

Beyond the existing technology presented in this report, there are some short and long term endeavors. Better power connectors for the motor supply are an immediate upgrade that would reduce some uncertainty in troubleshooting the stepper motors. A feedback loop was a short term goal and engineering requirement that did not take off due to time and financial constraints. A feedback loop for each syringe pump would allow better control of the motor controllers while providing logging data to the hub controller. Additional output signals were a short term goal that was cut as well. Instead of the single step function output, Dr. Hawkins requested a ramping output with the potential for other wave forms. This item was eventually cut due to time constraints. A Long term goal for this project includes a PCB design to reduce the project's footprint.

Appendix A
Impact Analysis of Senior Project

Project Title : Smart Motor Syringe System
Student Names : Connor Wilson
Advisor's Name : Ben Hawkins

A. Summary of Functional Requirements

The Smart Motor Syringe System is a syringe pump that aims to autonomously output multiple flow rates and provide the user with research-grade log data. It will be using three modular syringe pumps inside the system to administer three separated fluids. This system will be cost effective by utilizing open source hardware and software while providing medical grade flow rate tolerances.

B. Primary Constraints

The primary constraints of the SMSS include an absence of feedback, proprietary equipment, and budget. The absence of a feedback loop places a large risk on the system should part of the system change. The lack of custom equipment is also a constraint because the user can introduce a syringe or controller that is not well adapted to the existing SMSS system. Calibration would be required. Lastly, budget is a constraint because there cannot be high quality parts or application specific components without going over budget.

C. Economic

- Human Capital : The SMSS will create jobs in engineering, manufacturing, sales, and marketing. The SMSS could have an international presence should the ideal company see growth opportunities overseas.

- Financial Capital : The SMSS is designed to perform a highly complex task for under market value. The open-source technology in this device will provide a large margin of overhead and profit while saving the customer a portion as well. The financial capital is also aimed at potential investors as the product grows into a company.
- Natural Capital : While the SMSS is made from non-renewable parts that cannot be viably broken down into raw resources, the SMSS components can be re-purposed very easily. Most of the components inside the SMSS are stand alone devices, like the Raspberry Pi or the Arduino Nano, that can be used for other projects. A 3D printed housing case can also be broken down in a PLA/ABS foundry much easier than a complex polyethylene injection mold.
- Cost and Timing : The cost of the SMSS is broken down into assembly and validation. Since most of the manufacturing is done by the component sellers (Raspberry and Arduino), most of the time and cost spent in manufacturing is assembly. Validation on the other hand will require large sets of unit testing and calibration. This portion can easily be $100 per product. Overall, as more testing and assembly turn to automation, the less spent on cost and time.

TABLE XXI: Expected Cost/Revenue if Manufactured on a Large Scale

If Manufactured	Units	Reasoning
Units per year	200	This is the initial market before appealing to the average consumer
Manufacturing Cost	$400	Number includes material, parts, labor, and re-occurring validation
Purchase Price	$500	Minimum Viable Product
Profit per year	$10,000	Estimated
Operating cost for user	$200 USD for power $ 40 for annual care	Device requires power and requires regular service for lubricated parts.

D. Manufacturing

The SMSS will require a separate system for hardware assembly and validation along with software. Beyond the physical, validation testing and calibration will need to be performed on each unit as part of strict quality control. If the unit is sold as a kit, the open source hardware and software will be provided and the quality assurance will be left up to the user.

E. Environmental Impact

The SMSS will have a notable impact on the environment including significant power consumption, material consumption, and end-of-life disposal. The power consumed by this unit will add further demand for electricity and will not have a power saving feature due to its critical performance features being the minimum viable product. 3D printed shells will also require a lot of power from the grid.

Beyond power consumption, the SMSS will require large refinement of materials in order to make the components. This includes silicon for chips as well as precious metals that are in high demand. Chip shortages brought on by COVID-19 are also brought into play as manufacturers seek out unsavory alternatives for refined materials.

This refinement also makes this product hard to recycle and reclaim. Stripping this project back into its separated forms will require a large amount of processing and possibly toxic chemicals that can cause damage to organic life. This includes the ABS printing filament.

F. Sustainability

- **Issues maintaining the complete device or system** The SMSS relies on the functionality of multiple microcontrollers, stepper motors, ball screws, and power converters. Fluids in the syringe could cause potential harm to the electrical components . Physical damage or build-up of particles along the drive train could cause potential strain and damage to the system, shortening the lifespan of the product.
  The SMSS is also an open source project, allowing the user to replace commercial hardware components and reprogram them using an online database (Github) of code.
- **How the project impacts the sustainable use of resources** The SMSS does use electronic hardware that is hard to recycle but is easy to replace or reuse. It uses open source hardware/software and can be purchased, maintained, or re-purposed should the user no longer use the SMSS as intended.
- **Upgrades that would improve the design of the project** Calibration kits and negative feedback are two endeavors that would greatly improve the performance and reliability of the SMSS. The calibration kits would allow the user to ensure the precision/accuracy rates are correct as the system ages. Tuning software parameters with the calibration kit could potentially save the user money. Feedback compensation is another idea that would allow the system to perform better and hold tighter tolerances.
- **Challenges associated with upgrading the design** When upgrading the SMSS, it is important to take into account the customer base. If the customer is fluent in open source code/hardware, then applying updates through Git is likely the course of travel. Should the SMSS reach a launch data through the progress of others, it could potentially be up-gradable without pulling code requests.
  The hardware will be easy to upgrade in the early prototype phases until a release date. These upgrades are not limited to those in the conclusion.

G. Ethical

The SMSS is an open source syringe pump that has a few concerns. First, having a medical device with potential security risks can cause a few issues and benefits. The security risks involved with open source are up for debate as some open source systems, such as Linux, are stronger because of their large user community involvement. That being said, the Raspberry and Arduino systems have security risks that could cause unforeseen consequences.

Additional concerns arise if the user decides against using medical grade parts such as precision stepper motors or syringes. This option leaves the SMSS functionality up to the user should they void.

Using this technology to intentionally cause harm goes against my personal ethics. Misusing or misdiagnosing concentrations or flow rates into an organism is also not safeguarded by this device and responsibility must be placed on the user.

H. Health and Safety

The SMSS is a medical device that could have grave consequences if system or user errors occur. Beyond validation, the user needs to fully understand the risk involved with injecting an organism with a particular fluid and concentration. Improper use of this equipment could also result in dire consequences. This could be a valid reason why this technology is not readily advertised to the public as it comes with a steep set of instructions for proper operation. Overall, this system could result in injury, serious harm, or even death if not used properly.

I. Social and Political

The political/social impact of the SMSS involves the right-to-repair. As companies turn away from patients, they deny consumers the right to repair their own products. By keeping proprietary technology secret, consumers are forced to seek third parties that specialize in reverse engineering. While the SMSS is open source, it does have a high operating standard. Should the SMSS become a viable and successful product, the open source angle of marketing could cause consumers to demand more legal freedoms with their products. That being said, consumers should have access to proper calibration to meet the SMSS's bare engineering requirements.

There are conflicts with having an option to repair the SMSS. As a prebuilt system, the SMSS will run as intended just as other devices on the market. However, repairing and altering this product should also be viable.

APPENDIX B
SCHEMATICS

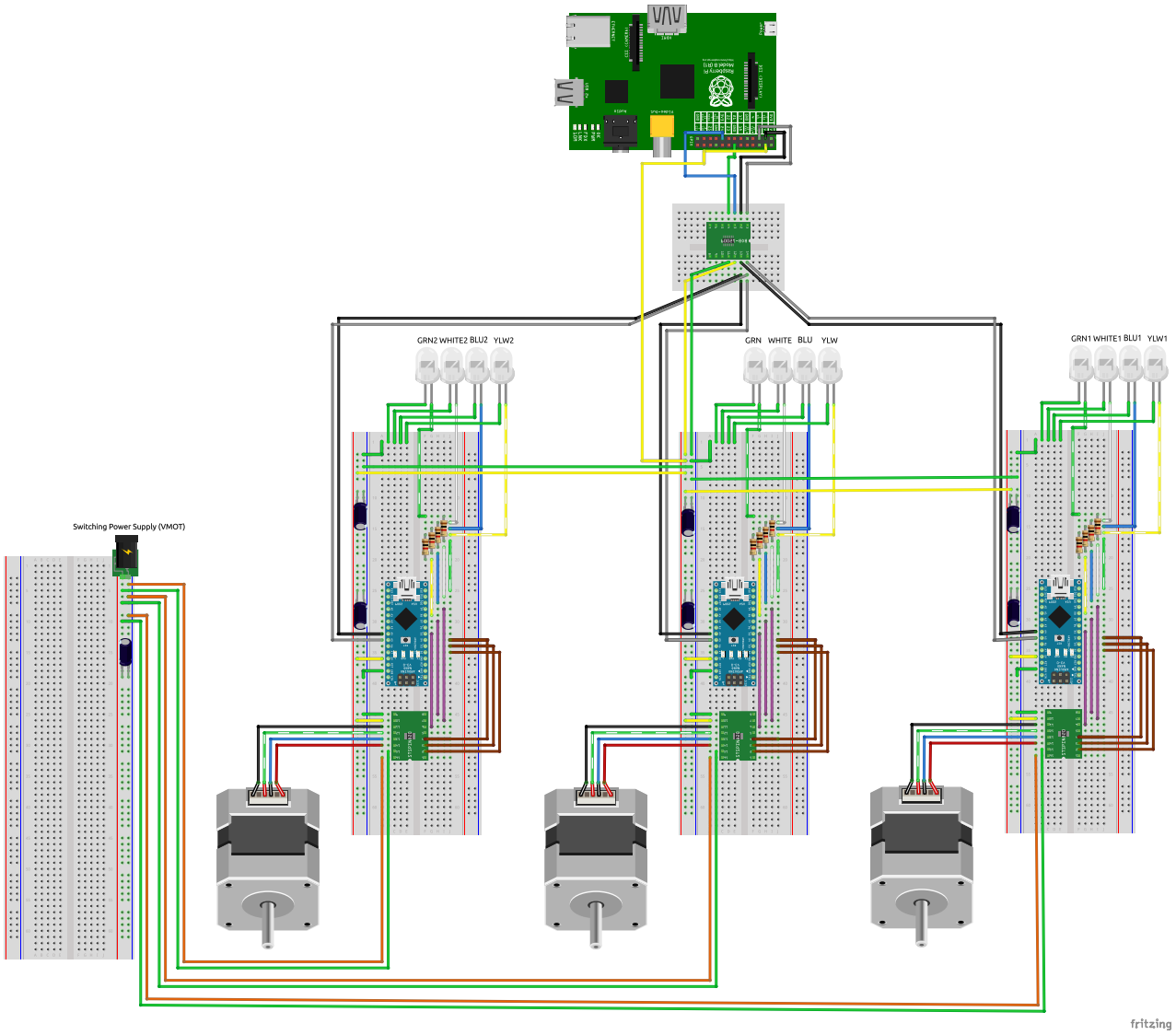

Fig. 44: Hardware Layout of the SMSS

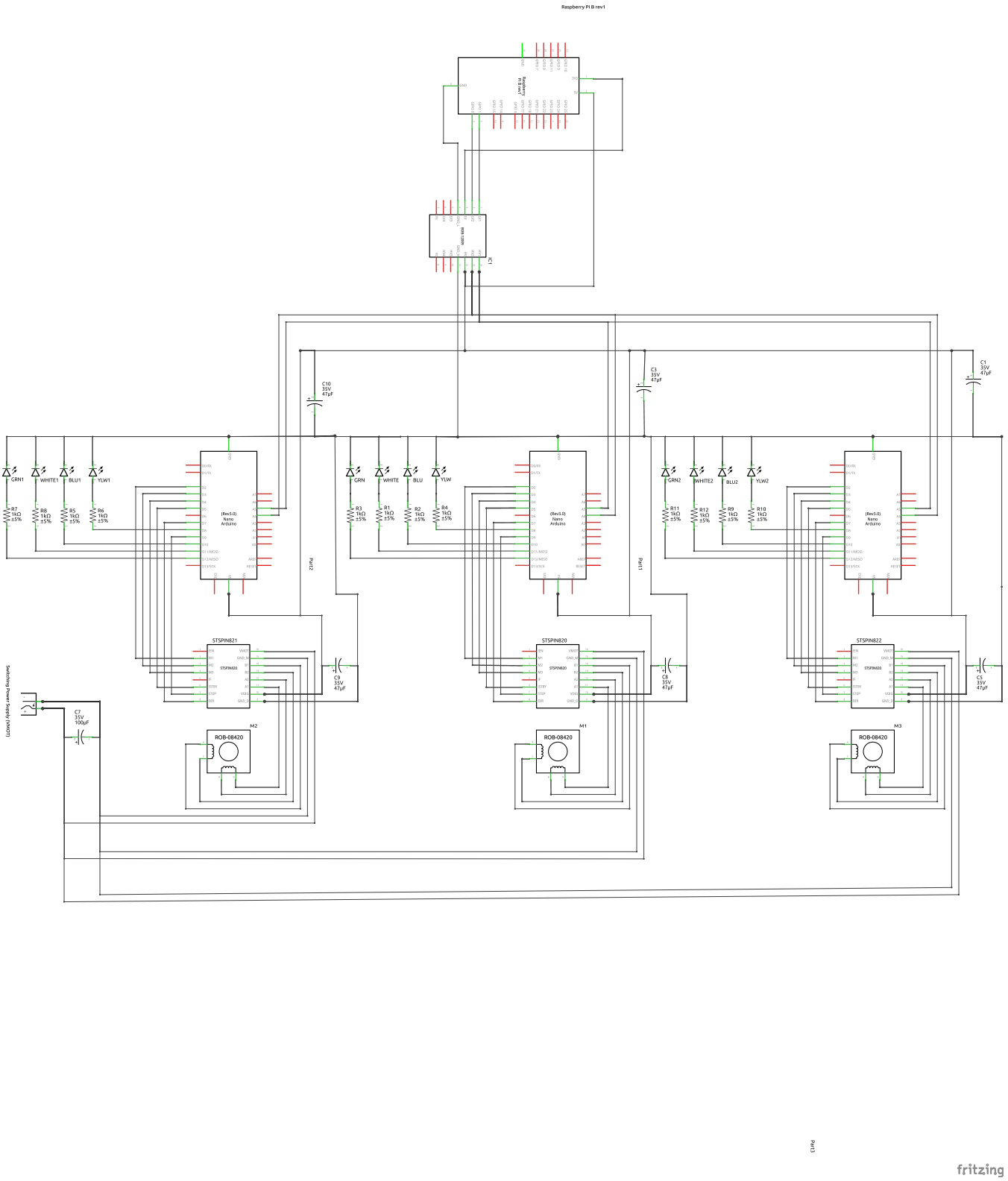

Fig. 45: Schematic of the SMSS

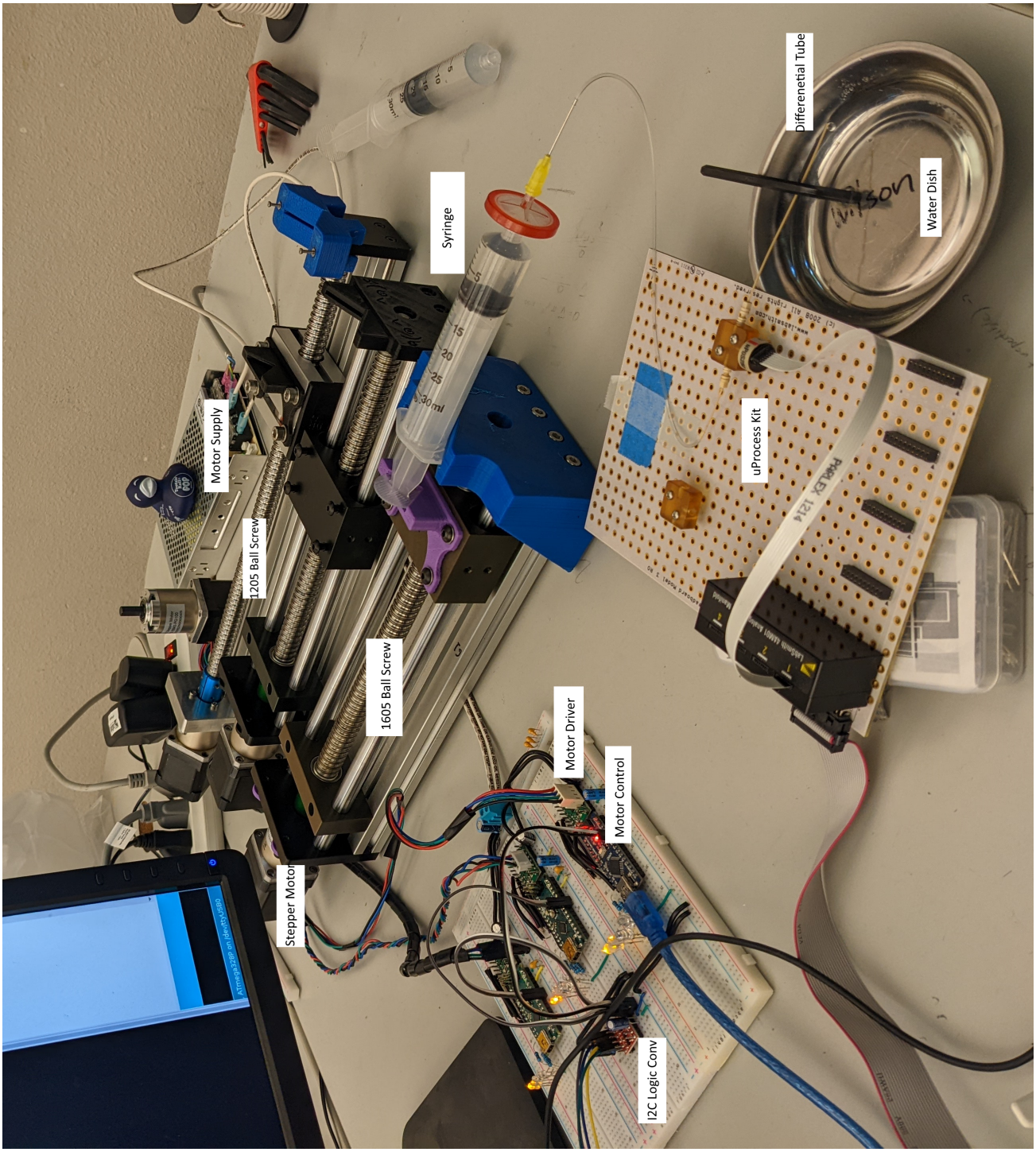

Fig. 46: Component Locations in the SMSS

APPENDIX C
CODE

A. Motor Controller

```

// Connor Wilson
// Winter 2022
// SMSS Motor Controller V1.0

#include <TimerOne.h>           // Custom PWM freq ( ONLY WORKS WITH MEGA328 - NO MICRO BOARDS) - PWM only for pins(9, 10) on nano
#include <Wire.h>               // I2C



// Test Variables (COMMENT OUT TO DESELECT) -- DONT FORGET TO CHANGE ARD ADDRESS (i2c_address) BELOW
#define TESTING                 // Comment to cancel testing
//#define SERIAL_COMMS          // Comment to cancel serial comms [manual testing]
#define I2C_COMMS               // Comment to cancel I2C comms
//#define STSPIN220             // Comment out to NOT use the STSPIN220 Pololu low-volt motor controller
#define STSPIN820             // Comment out to NOT use the STSPIN820 Pololu high-volt motor controller


// State machine
enum State_enum {STANDBY, DATAPULL_S, DATAPULL_J, RUN_S, RUN_J, STOP};  // States of the state machine
unsigned int state = STANDBY;
                        // default state
#define START 'S'
#define JOG 'J'
#define BACK 'B'
#define ESTOP 'E'
bool TICKET = false;        // true if user supplied data is valid with hardware
bool FINISH = false;        // true if user requested duration is matched by hardware



#ifdef STSPIN220 // Low voltage motor controller library
// Pin Connections
#define M1 2                    // microstep config bit 1
#define M2 3                    // microstep config bit 2
#define STBY 5                  // Controller's standby enable (both EN and STBY must be low for low power mode)
#define STEP 9                  // PWM signal for motor
#define DIR 7                   // Direction of motor
// NOTE : STEP, DIR, M2, M1 are all used to set the microstep in the init latching
//        but only STEP and DIR are availible after init


// LED State Indicators
#define YELLOW 8
#define BLUE 10
#define WHITE 11
#define GREEN 12


// Microstep settings for md39a[ M1, M2, STEP, DIR ]
#define S_FULL  0x0
#define S_HALF  0xA
#define S_QRTR  0x5
#define S_8     0xE
#define S_16    0xF
#define S_32    0x4
#define S_64    0xD
#define S_128   0x8
#define S_256   0x6
char step_config = S_FULL;                  // Set default to full step
#endif


#ifdef STSPIN820 // High voltage motor controller library
// Pin Connections
#define M1 2                    // microstep config bit 1
#define M2 3                    // microstep config bit 2
#define M3 4                    // microstep config bit 3
#define STBY 5                  // Controller's standby enable (both EN and STBY must be low for low power mode)
#define STEP 9                  // PWM signal for motor
#define DIR 7                   // Direction of motor
// NOTE : M1-3 are used for microstep latching (ie STEP/DIR are not used for latching like the STSPIN220)


// LED State Indicators
#define YELLOW 8
#define BLUE 10
#define WHITE 11
#define GREEN 12



// Microstep settings for md39a[ M1, M2, STEP, DIR ]
#define S_FULL  0x0
#define S_HALF  0x4
#define S_QRTR  0x2
#define S_8     0x6
#define S_16    0x1
#define S_32    0x5
//#define S_64  no 64 bit
#define S_128   0x3
#define S_256   0x7
char step_config = S_FULL;                  // Set default to full step
#endif


// Motor States
#define FORWARD LOW             // motor direction for pin logic
#define REVERSE HIGH            // motor direction for pin logic
bool DIRECTION = FORWARD;       // motor direction
bool RUNNING  = false;          // true if motor is running




// Pulse/Freq Variables
#define DC  0.8 * 1024                      // Duty Cycle of PWM
                [UNSURE WHAT RATIO IS BEST]
#define MTR_FREQ_MAX 875                    // Max freq motor can handle [Hz]
```

```
#define MTR_RES_MIN 120                       // Min freq for smooth performance [Hz]
#define MTR_FREQ_MIN 0                        // Min freq motor can handle [Hz]
       [UNKNOWN]


// Physical Variables
#define INTERNAL_STEP 200                     // Internal step size of the motor
#define GEAR_RATIO  99.05                     // gear ratio of the planetary gear
#define THREAD_PITCH  0.005                   // thread pitch of the actuator's screw [m]
float syr_capacity = 30;                      // capacity of syringe in mL
float syr_radius_user = 11;                   // RADIUS of the 30ml BD syringe [m]
float q_user = 0;                             // volumetric flow rate as requested by user [L/s]
float freq = 200;                             // [sec/step]
unsigned long sec_user = 0;                   // User requested sec duration
unsigned long min_user = 0;                   // User requested min duration
unsigned long hour_user = 0;                  // User "          hour "
unsigned long duration = 0;                   // Run time {msec}
unsigned long time_stamp_start = 0;           // time keeping
unsigned long time_stamp_end = 0;             // "


// Serial Variables
#ifdef SERIAL_COMMS
char receivedChar;                            // pool for char data
boolean newData_char = false;                 // true if new char data is RX
boolean newData_str = false;                  // true if new str data is RX
const byte numChars = 32;                     // str length when TX
char receivedChars[numChars];                 // pool for str data
bool displayed = false;                       // true if graphic is displayed
#endif

// I2C Variables
#ifdef I2C_COMMS

// I2C Addresses
#define PI_ADD 0x01                           // Pi address
#define ARD_ADD_1 0x14                        // Arduino #1
#define ARD_ADD_2 0x28                        // Arduino #2
#define ARD_ADD_3 0x42                        // Arduino #3
int i2c_address = ARD_ADD_3;                  // EDIT THIS TO CHANGE ADDRESS

// I2C Variables
int poll_user = 0;                          // run state of the ard [OFF(0), JOG(1), START(2) ]
int recData = 0;                            // storage for dictation codes
char strRX[2000];                             // RX storage
char dur_mark = 0;                          // marker to aid recv/req track incoming hour, min, sec
bool datapull_flag = false;                 // completed datapull from I2C
int motor_dir = FORWARD;                    // [ (1) - REV, (0) - FWD, (2) - *BACK to STANDBY] *some cases(change l8r)

// Pi's Dictating (main) codes
#define POLLDATA 1                          // Ard will recv[OFF, JOG, START]
#define Q_DATA 2                            // Ard will recv[q_user]
#define R_DATA 3                            // Ard will recv[syr_radius_user]
#define CAP_DATA 4                          // Ard will recv[syr_capacity]
#define DUR_DATA 5                          // Ard will recv[hour_user, min_user, sec_user]
#define DIR_DATA 6                          // Ard will recv[DIRECTION]
#define EVENT_DATA 7                        // Ard will proceed to RUN
#define ESTOP_DATA 8                        // Ard will proceed to STOP
#define REDO_DATA 9                         // Ard will reset to STANDBY from DATAPULL
#define BLOCK_DATA 10

// Arduino's Requested (Sub) codes
#define LARGE_ERR 0x3                       // User req too large (Q) for hardware constraints
#define SMALL_ERR 0x4                       // User req too small (Q) for hardware constraints
#define DUR_ERR 0x5                         // User req too long (time) for hardware constraints
#define LD_ERR 0x6                          // User too large and long
#define SD_ERR 0x7                          // user too small and long
#define DICT_ERR 0x8                        // wrong dictating codes
#define NO_ERR 0x10                         // No error to report
int error_code = NO_ERR;

#define WAIT_CODE 0x15                      // Ard needs time to pull data and validate
#define FINISH_CODE 0x16                    // Ard's duration has been reached and finished RUN_S
#define TIME_CODE 0x17                      // Ard is not done with RUN_S and will return time(h,m,s) if prompted
bool redo_flag = false;
bool time_flag = false;                     // bool to help with req time code [ true - sent TIME_CODE already]
char block_recv = 0;
unsigned long block[6] = {0, 0, 0, 0, 0, 0}; // [BLOCK_DATA, dict_code, fac1, rem1, fac2, rem2] for data RX (I2c only transmitts 256 bit)


// Arduino checkpoints
bool dp_checkpoint = false;                 // true - ard has finished processing TICKET
bool eventstart = false;                    // true - Pi has req to RUN the DATAPULL'd ard
bool estop_user = false;                    // true - Pi req arduino to stop

#endif


void setup() {

#ifdef SERIAL_COMMS
  Serial.begin(9600);                           // Serial comms init
#endif

```

```

#ifdef I2C_COMMS

#ifdef TESTING
  Serial.begin(9600);
#endif

  Wire.begin(i2c_address);                          // I2C bus logon with sub address
  Wire.onReceive(recvEvent);                        // run recvEvent on a main write to read <-
  Wire.onRequest(reqEvent);                         // run reqEvent on a main read to write ->
#endif

  pinMode(STEP, OUTPUT);                            // Setup controller pins
  pinMode(DIR, OUTPUT);
  pinMode(STBY, OUTPUT);
  pinMode(M1, OUTPUT);
  pinMode(M2, OUTPUT);

#ifdef STSPIN820
  pinMode(M3, OUTPUT);
#endif

  pinMode(BLUE, OUTPUT);                            // Indicator LED pin setup
  pinMode(YELLOW, OUTPUT);
  pinMode(GREEN, OUTPUT);
  pinMode(WHITE, OUTPUT);
  digitalWrite(DIR, REVERSE);                       // Direction of motor

}


void loop() {

#ifdef SERIAL_COMMS
  serial_state_machine();
#endif

#ifdef I2C_COMMS
  i2c_state_machine();
#endif
}


#ifdef STSPIN220
/*~~~~~~~~~~~~~~~~~~~~~~~~ MtrCtrl_init_220()
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    Purpose : initialize or change the low voltage
    md39a controllers step size
    Input : 0x4'b  [ M1, M2, STEP, DIR ] as
    calculated by the requested step setting
    Outut : digital write to pins
    Notes : ~EN pin is held low
*/
void MtrCtrl_init_220(char pin_config) {


#ifdef TESTING
  Serial.print("pin_config : ");
  Serial.println(pin_config, HEX);
#endif

  //The recommended power-up sequence is following:

  //1. Power-up the device applying the VS supply
    voltage but keeping both STBY and EN/FAULT
    inputs low.
  digitalWrite(STBY, LOW);

  //2. Set the MODEx inputs according to the target
    step resolution (see Table 1).
  digitalWrite(DIR, pin_config & 1);
  digitalWrite(STEP, (pin_config >> 1) & 1);
  digitalWrite(M2, (pin_config >> 2) & 1);
  digitalWrite(M1, (pin_config >> 3) & 1);

  //3. Wait for at least 1 microsec (minimum tMODEsu
    time).
  delayMicroseconds(1);

  //4. Set the STBY high. The MODEx configuration is
    now latched inside the device.
  digitalWrite(STBY, HIGH);

  //5. Wait for at least 100 microsec (minimum
    tMODEho time).
  delayMicroseconds(100);

  //6. Enable the power stage releasing the EN/FAULT
    input and start the operation.
}

#endif



#ifdef STSPIN820

/*~~~~~~~~~~~~~~~~~~~~~~~~ MtrCtrl_init_820()
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    Purpose : initialize or change the high voltage
    md37a controllers step size
    Input : 0x4'b  [ M1, M2, M3 ] as calculated by
    the requested step setting
    Outut : digital write to pins
    Notes : ~EN pin is held low
*/
void MtrCtrl_init_820(char pin_config) {



#ifdef TESTING
  Serial.print("pin_config : ");
  Serial.println(pin_config, HEX);
#endif

  //The recommended power-up sequence is following:

  //1. Power-up the device applying the VS supply
    voltage but keeping both STBY and EN/FAULT
    inputs low.
  digitalWrite(STBY, LOW);

  //2. Set the MODEx inputs according to the target
    step resolution (see Table 1).
  digitalWrite(M3, pin_config & 1);
  digitalWrite(M2, (pin_config >> 1) & 1);
  digitalWrite(M1, (pin_config >> 2) & 1);

#ifdef TESTING
  Serial.println(pin_config & 1);
  Serial.println((pin_config >> 1) & 1);
  Serial.println((pin_config >> 2) & 1);
#endif

  //3. Wait for at least 1 microsec (minimum tMODEsu
    time).
  delayMicroseconds(1);

  //4. Set the STBY high. The MODEx configuration is
    now latched inside the device.
  digitalWrite(STBY, HIGH);

  //5. Wait for at least 100 microsec (minimum
    tMODEho time).
  delayMicroseconds(100);

  //6. Enable the power stage releasing the EN/FAULT
    input and start the operation.
```

```
}

#endif

/*~~~~~~~~~~~~~~~~~~~~~~~~ Q_to_Freq()
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    Purpose : Convert the requested Q rate into a
    frequency for the stepper motor
    Input : q_user, syr_radius, Physical quantities
    Outut : freq
*/
float Q_to_Freq( float Q, float Radius) {

  float freq = ( Q * INTERNAL_STEP * GEAR_RATIO) /
    (500 * Radius * Radius * THREAD_PITCH);
  return freq;
}


/*~~~~~~~~~~~~~~~~~~~~~~~~ MicroCali()
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    Purpose : Recalibrate the md39a motor controller
    's microstep setting to accomidate the min freq
    recommendation
              Max freq will result in Full step at
    max freq + warning
    Input : frequency, [global] step_config, [global
    ] MTR_FREQ_MAX, [global] MTR_RES_MIN
    Output : frequency, [global] step_config, [
    global] TICKET
    Notes : Removes decimal point to save memory
*/
double MicroCali( float frequency ) {


  // If greater than the max frequency
  if ( frequency > MTR_FREQ_MAX) {

    step_config = S_FULL;
                          // Set controller to full
    step
    TICKET = false;
                            // Mark ticket invalid

#ifdef SERIAL_COMMS
    Serial.println(" Requested Q value exceeds
    hardware's MAX RPS");  // Inform user
#endif

#ifdef I2C_COMMS
    error_code = LARGE_ERR;
#endif

    return MTR_FREQ_MAX;
                          // Set freq to max
  }

  // Or if less than the min freq
  else if (frequency < MTR_RES_MIN) {

    double microstep = MTR_RES_MIN / frequency;
                         // Microstep size

    if ( microstep <= 2 ) {
                          // Half step
      step_config = S_HALF;
                          // Set controller to half
    step
      TICKET = true;
                          // Approved ticket
      return 2 * frequency;
                          // double freq
    }

    else if ( microstep <= 4 ) {
      step_config = S_QRTR;
                          // Set controller to half
    step
      TICKET = true;
                          // Approved ticket
      return 4 * frequency;
                          // double freq
    }

    else if ( microstep <= 8 ) {
      step_config = S_8;
      TICKET = true;
                          // Approved ticket
      return 8 * frequency;
    }

    else if ( microstep <= 16 ) {
      step_config = S_16;
      TICKET = true;
                          // Approved ticket
      return 16 * frequency;
    }

    else if ( microstep <= 32 ) {
      step_config = S_32;
      TICKET = true;
                          // Approved ticket
      return 32 * frequency;
    }

    else if ( microstep <= 64 ) {

      // the STSPIN820 doesnt have 64 bit -> make it
     128 bit instead
#ifdef STSPIN220
      step_config = S_64;
      TICKET = true;
                          // Approved ticket
      return 64 * frequency;
#else
      step_config = S_128;
      TICKET = true;
                          // Approved ticket
      return 128 * frequency;
#endif

    }

    else if ( microstep <= 128 ) {
      step_config = S_128;
      TICKET = true;
                          // Approved ticket
      return 128 * frequency;
    }

    else if ( microstep <= 256 ) {
      step_config = S_256;
      TICKET = true;
                          // Approved ticket
      return 256 * frequency;
    }

    else {
      step_config = S_256;
                           // Set controller to
    smallest step
      TICKET = false;

#ifdef SERIAL_COMMS
      Serial.println(" Requested Q value is below
    hardware's MIN RPS");  // Inform user
#endif

#ifdef I2C_COMMS
      error_code = SMALL_ERR;
                      // report error to pi
#endif

      return MTR_RES_MIN;
```

```
                              // Set freq to min
449
450     }
451   }
452
453   // No microstepping change needed
454   else {
455     TICKET = true;
                                // Approved ticket
456     return frequency;
457   }
458
459 }
460
461
462 /* ~~~~~~~~~~~~~~~~~~~~~~~~ validDur_Start()
      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
463     Purpose : Confirm volumetric flow rate x
      duration wont exceed the syringe capacity
464     Input : [global]   q_user[L/s], syr_capacity[m],
       duration[s]
465     Output : [global] bool TICKET
466 */
467 void validDur_Start(float Q, float Vol, float Time)
      {
468   if ( (Q * Time) <= Vol ) {
469
470 #ifdef TESTING
471     Serial.println(" Valid dur");
472 #endif
473
474     TICKET = true;
475   }
476   else {
477
478 #ifdef I2C_COMMS
479     if (error_code == NO_ERR) {
480       error_code = DUR_ERR;
481     }
482     else if (error_code == LARGE_ERR) {
483       error_code = LD_ERR;
484     }
485     else if (error_code == SMALL_ERR) {
486       error_code = SD_ERR;
487     }
488 #endif
489
490 #ifdef TESTING
491     Serial.println("Bad Dur");
492     Serial.println(Q, 10);
493     Serial.println(Vol, 10);
494     Serial.println(Time, 15);
495 #endif
496
497     TICKET = false;
498   }
499 }
500
501
502
503
504
505 #ifdef SERIAL_COMMS
506 /* ~~~~~~~~~~~~~~~~~~~~~~~~ serial_state_machine()
      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
507     Purpose : Control the flow of the motor
      controller
508     Input : Serial user input signals
509     Output :
510 */
511 void serial_state_machine() {
512
513   switch (state) {
514
515     case STANDBY:
516
517       digitalWrite(YELLOW, HIGH);         // LED
518     indicator
519
520       if (displayed == false) {
                              // dont print unless its the
      first
521         Serial.println(" ~~~~~~~~~~~~Standby Mode
      ~~~~~~~~~~~ ");
522         Serial.println(" Press S for START mode or J
      for JOG (No line ending) \n\n\n\n");
523         displayed = true;
524       }
525
526       recvOneChar();
527
528       if (receivedChar == START) {
                    // User selects START for normal use
529
530 #ifdef SERIAL_COMMS
531         displayed = false;
                    //graphic already displayed...no
      more pls
532 #endif
533
534         digitalWrite(YELLOW, LOW);
                    // LED indicator
535         state = DATAPULL_S;
536       }
537
538       else if (receivedChar == JOG) {
                     // User selects JOG to jog motor
539
540 #ifdef SERIAL_COMMS
541         displayed = false;
542 #endif
543
544         digitalWrite(YELLOW, LOW);
                    // LED indicator
545         state = DATAPULL_J;
546       }
547
548       else {
549         state = STANDBY; // Or wait until user
      chooses
550       }
551
552
553       break;
554
555     case DATAPULL_S:
556
557       digitalWrite(BLUE, HIGH);
                  // LED indicator
558
559       Serial.println(" ~~~~~~~~~~DATAPULL_S~~~~~~~~~
      ");
560       Serial.println(" What is the fluid capacity of
      the syringe[mL]? (newline)");
561       while (newData_str == false) {
562         recvWithEndMarker();
563       }
564       syr_capacity = userString2Float() / 1000;
565
566       Serial.println(" What is the radius of the
      syringe plunger[um]? (newline)");
567       while (newData_str == false) {
568         recvWithEndMarker();
569       }
570       syr_radius_user = userString2Float() /
      1000000;
571
572       Serial.println(" What is the requested
      volumetric flow rate Q [nL/s] ? Range [69n , 10u
      ] (newline)");
573       while (newData_str == false) {
574         recvWithEndMarker();
575       }
```

```
      q_user = userString2Float() / 1000000000;

      Serial.println(" Duration hours? (newline)");
      while (newData_str == false) {
        recvWithEndMarker();
      }
      hour_user = userString2Float();

      Serial.println(" Duration minutes? (newline)");
      while (newData_str == false) {
        recvWithEndMarker();
      }
      min_user = userString2Float();

      Serial.println(" Duration seconds? (newline)");
      while (newData_str == false) {
        recvWithEndMarker();
      }
      sec_user = userString2Float();

      Serial.println(" Direction? F for forward, R for reverse, or B to back to Standby (No line end)");
      newData_char = false;
      while (newData_char == false) {
        recvOneChar();
      }
      if ( receivedChar == 'R') {
        DIRECTION = REVERSE;

#ifdef TESTING
        Serial.println(" Capacity, raduis, Q, hour, min, sec, Dir ");
        Serial.print(syr_capacity);
        Serial.print(", ");
        Serial.print(syr_radius_user, 8);
        Serial.print(", ");
        Serial.print(q_user, 10);
        Serial.print(", ");
        Serial.print(hour_user);
        Serial.print(", ");
        Serial.print(min_user);
        Serial.print(", ");
        Serial.print(sec_user);
        Serial.print(", ");
        Serial.println("Reverse\n\n");
#endif
      }
      else if ( receivedChar == 'F') {
        DIRECTION = FORWARD;

#ifdef TESTING
        Serial.println(" Capacity, raduis, Q, hour, min, sec, Dir ");
        Serial.print(syr_capacity);
        Serial.print(", ");
        Serial.print(syr_radius_user, 8);
        Serial.print(", ");
        Serial.print(q_user, 10);
        Serial.print(", ");
        Serial.print(hour_user);
        Serial.print(", ");
        Serial.print(min_user);
        Serial.print(", ");
        Serial.print(sec_user);
        Serial.print(", ");
        Serial.println("Forward\n\n");
#endif

      }

      duration = ( (3600 * hour_user) + (60 * min_user) + sec_user);   // entire duration in msec for easy compare later

      freq =  Q_to_Freq(q_user, syr_radius_user);
                                // Get period from Q
      freq = MicroCali(freq);
                                // Calibrate for microstepping + TICKET

      if (duration == 0) {
        TICKET = false;
      }
      else if (TICKET == true) {
        validDur_Start(q_user, syr_capacity , duration); // final TICKET confirm duration
      }

      if (receivedChar == BACK) {
        digitalWrite(BLUE, LOW);
                                // LED indicator
        state = STANDBY;
      }
            // User requests to go back
      else if (TICKET) {
                                // Must lie within Q and duration range
        digitalWrite(BLUE, LOW);
                                 // LED indicator
        state = RUN_S;
      }
              // Valid data, move forward
      else {
        state = DATAPULL_S; // Repeat if the data wasnt RX/right
      }

      break;


    case DATAPULL_J:
      digitalWrite(BLUE, HIGH);
                                // LED indicator
      digitalWrite(WHITE, HIGH);
                                // LED indicator

      Serial.println(" \n\n\n\n~~~~~~~~~~ DATAPULL_JOG~~~~~~~~~ ");
      Serial.println(" What is the radius of the syringe plunger [um]? (newline)");
      while (newData_str == false) {
        recvWithEndMarker();
      }
      syr_radius_user = userString2Float() / 1000000;

      Serial.println(" What is the requested volumetric flow rate Q [nL/s] ? Range [69n , 10u ] (newline)");
      while (newData_str == false) {
        recvWithEndMarker();
      }
      q_user = userString2Float() / 1000000000;

      Serial.println(" Direction? F for forward, R for reverse, or B to back to Standby (No line ending)");
      newData_char = false;
      while (newData_char == false) {
        recvOneChar();
      }
      if ( receivedChar == 'R') {
        DIRECTION = REVERSE;
      }
      else if ( receivedChar == 'F') {
        DIRECTION = FORWARD;
      }

      freq =  Q_to_Freq(q_user, syr_radius_user);
                                // Get period from Q
```

```
#ifdef TESTING
      Serial.println("radius, Q, freq ");
      Serial.print(syr_radius_user, 8);
      Serial.print(", ");
      Serial.print(q_user, 10);
      Serial.print(", ");
      Serial.println(freq);
#endif

      freq = MicroCali(freq);                    // Calibrate microstepping & TICKET validation

#ifdef TESTING
      Serial.println("radius, Q, freq ");
      Serial.print(syr_radius_user, 8);
      Serial.print(", ");
      Serial.print(q_user, 10);
      Serial.print(", ");
      Serial.println(freq);

#endif

      if (receivedChar == BACK) {                // User requests to go back
        //reset metrics and go back
        freq = 0;
        digitalWrite(BLUE, LOW);                 // LED indicator
        digitalWrite(WHITE, LOW);                // LED indicator
        state = STANDBY;
      }

      else if (TICKET == true) {
        digitalWrite(BLUE, LOW);                 // LED indicator
        digitalWrite(WHITE, LOW);                // LED indicator
        state = RUN_J;
      }
              // Valid data, move forward

      else {
        state = DATAPULL_J; // Repeat if the data wasnt RX/right
      }

      break;


    case RUN_J:

      digitalWrite(GREEN, HIGH);                 // LED indicator
      digitalWrite(WHITE, HIGH);                 // LED indicator

      // Allowing arduino to roam through cycles and not wait for user input while running
      if (displayed == false) {
        Serial.println(" ~~~~~~~~~~~~ Run_JOG ~~~~~~~~~~~~ ");
        Serial.println(" Press E for ESTOP (No line ending)");
        displayed = true;
      }

      recvOneChar();

      if (receivedChar == ESTOP) {
        digitalWrite(GREEN, LOW);                // LED indicator
        digitalWrite(WHITE, LOW);                // LED indicator
        state = STOP;
      }

      else if (RUNNING == false) {

#ifdef STSPIN220
        MtrCtrl_init_220(step_config);           // setup motor controller
#else
        MtrCtrl_init_820(step_config);           // setup motor controller
#endif

        digitalWrite(DIR, DIRECTION);            // rewrite DIR pin for direction
        Timer1.initialize((1 / freq) * 1000000); // Init PWM freq[microsec] (mult to conv to usec)
        Timer1.pwm(STEP, DC);                    // Start motor
        RUNNING = true;
        state = RUN_J;
      }

      else {

        state = RUN_J;
      }

      break;


    case RUN_S:

      digitalWrite(GREEN, HIGH);                 // LED indicator

      if (displayed == false) {
        Serial.println(" ~~~~~~~~~~~~ Run_START ~~~~~~~~~~~ ");
        Serial.println(" Press E for ESTOP (No line ending)\n\n\n\n");
        displayed = true;
      }

      recvOneChar();

      if (receivedChar == ESTOP || FINISH == true) {
        displayed = false;
        digitalWrite(GREEN, LOW);                // LED indicator
        state = STOP;
      }

      // init run of motor
      else if (RUNNING == false) {

#ifdef STSPIN220
        MtrCtrl_init_220(step_config);           // setup motor controller
#else
        MtrCtrl_init_820(step_config);           // setup motor controller
#endif

        digitalWrite(DIR, DIRECTION);            // rewrite DIR pin for direction
        Timer1.initialize((1 / freq) * 1000000); // Init PWM freq[microsec] (mult to conv to usec)
        time_stamp_start = millis();             // Start recording time
        Timer1.pwm(STEP, DC);                    // Start motor
        RUNNING = true;
        state = RUN_S;
      }
```

```
      // Running until final time
      else {

        time_stamp_end = millis();
        time_stamp_end = (time_stamp_end -
    time_stamp_start);       // total elapsed time
        if ( time_stamp_end / 1000 >= duration) {
          FINISH = true; // if time had reached user
    's request
        }
        state = RUN_S;
      }

      break;

    case STOP:
      digitalWrite(STBY, LOW);
                            // Place motor controller
     in standby
      Timer1.disablePwm(STEP);
                            // turn off motor
      RUNNING = false;
      TICKET = false;
      displayed = false;
      FINISH = false;
      q_user = 0;
                    // clear user variables
      syr_radius_user = 0;
      freq = 0;
      duration = 0;
      hour_user = 0;
      min_user = 0;
      sec_user = 0;
      state = STANDBY;
      break;

    default:
      state = STOP;
      break;


  }
}



/*~~~~~~~~~~~~~~~~~~~~~~~~~ recvOneChar()
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   Purpose : Receive one char from the user
   Input : serial
   Output : [global] newData
*/
void recvOneChar() {
  if (Serial.available() > 0) {
    receivedChar = Serial.read();
    newData_char = true;
  }
}



/*~~~~~~~~~~~~~~~~~~~~~~~~~ recvWithEndMarker()
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   Purpose : Receive set of char (up to 32) from the
     user
   Input : serial
   Output : [global] receivedChars[]
*/
void recvWithEndMarker() {
  static byte ndx = 0;
  char endMarker = '\n';
  char rc;


  while (Serial.available() > 0 && newData_str ==
    false) {
    rc = Serial.read();

    if (rc != endMarker) {
      receivedChars[ndx] = rc;
      ndx++;
      if (ndx >= numChars) {
        ndx = numChars - 1;
      }
    }
    else {
      receivedChars[ndx] = '\0'; // terminate the
    string
      ndx = 0;
      newData_str = true;
    }
  }


}



/*~~~~~~~~~~~~~~~~~~~~~~~~~ userString2Float()
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   Purpose : convert recvWithEndMarker's output
    array of char receivedChars[ndx] to float &
    continues data flow
   Input : [global] char receivedChars[]
   Output : float
*/
float userString2Float() {
  if (newData_str == true) {
    String user_str;
    user_str = String(receivedChars);
    newData_str = false;
    return user_str.toFloat();
  }
}


#endif




#ifdef I2C_COMMS
/* ~~~~~~~~~~~~~~~~~~~~~~~~~ i2c_state_machine()
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   Purpose : Control the flow of the motor
    controller
   Input : I2C user input signals
   Output :
*/
void i2c_state_machine() {

  switch (state) {



    case STANDBY:

      digitalWrite(YELLOW, HIGH);
                  // LED indicator

      if (estop_user == true) {
        state = STOP;
      }

      else if (poll_user == 2) {
                        // User selects START for
    normal use
        digitalWrite(YELLOW, LOW);
                // LED indicator
        state = DATAPULL_S;
```

```
      }

      else if (poll_user == 1) {
                  // User selects JOG to jog motor
        digitalWrite(YELLOW, LOW);
                // LED indicator
        state = DATAPULL_J;
      }

      else {
        state = STANDBY; // Or wait until user chooses
      }

      break;

    case DATAPULL_S:

      digitalWrite(BLUE, HIGH);
                                  // LED indicator

      if (redo_flag == true || estop_user == true) {
        state = STOP;
      }

      // if all data has been pulled from Pi, process and find errors (if exist)
      else if (dp_checkpoint == false && datapull_flag == true) {


        // Process variables for valid TICKET and check for errors
        duration =  (3600 * hour_user) + (60 * min_user) + sec_user;   // entire duration in msec for easy compare later

#ifdef TESTING
        Serial.print("hour_user : ");
        Serial.println(3600 * hour_user);
        Serial.print("min_user : ");
        Serial.println(60 * min_user);
        Serial.print("sec_user : ");
        Serial.println(sec_user);
        Serial.print("Duration : ");
        Serial.println(duration, 10);
#endif

        freq =  Q_to_Freq(q_user, syr_radius_user);
                                  // Get period from Q

#ifdef TESTING
        Serial.print("Freq : ");
        Serial.println(freq);
#endif

        freq = MicroCali(freq);
                                  // Calibrate for microstepping + error check (TICKET)

#ifdef TESTING
        Serial.print("Freq: ");
        Serial.println(freq);
#endif

        if (duration == 0) {
          TICKET = false;
        }
        else if (TICKET == true) {
          validDur_Start(q_user, syr_capacity, duration); // final TICKET confirm duration + error
        }

        // Go to STANDBY is user sel to go back
        if (motor_dir == 2) {
          poll_user = 0;              // reset user seletion
          digitalWrite(BLUE, LOW);
          state = STANDBY;
        }

        // Proceed with current data
        else {
          // ready to give feedback to Pi
          dp_checkpoint = true;
          state = DATAPULL_S;
        }
      }

      // if all pulled data is valid, Pi has valid TICKET, and is ready to run
      else if (TICKET == true && eventstart == true) {                          // Must lie within Q and duration range
        digitalWrite(BLUE, LOW);
                                  // LED indicator
        datapull_flag = false;
        eventstart = false;
        TICKET = false;
        state = RUN_S;
      }

      // Not all data has been pulled/ Pi not ready to RUN / Errors exist
      else {
        state = DATAPULL_S;
      }

      break;


    case DATAPULL_J:
      digitalWrite(BLUE, HIGH);
                                  // LED indicator
      digitalWrite(WHITE, HIGH);
                                  // LED indicator

      if (redo_flag == true || estop_user == true) {
        state = STOP;
      }

      // if all data has been pulled from Pi, process and find errors (if exist)
      else if (dp_checkpoint == false && datapull_flag == true) {
        freq =  Q_to_Freq(q_user, syr_radius_user);
                                  // Get period from Q
        freq = MicroCali(freq);
                                  // Calibrate microstepping & TICKET validation

        // Go to STANDBY is user sel to go back
        if (motor_dir == 2) {
          digitalWrite(BLUE, LOW);
          state = STANDBY;
        }

        // Proceed with current data
        else {
          // ready to give feedback to Pi
          dp_checkpoint = true;
          state = DATAPULL_J;
        }

      }


      // if all pulled data is valid, Pi has valid TICKET, and is ready to run
      else if (TICKET == true && eventstart == true) {
        digitalWrite(BLUE, LOW);
                                  // LED indicator
```

```
        digitalWrite(WHITE, LOW);
                                  // LED indicator
        datapull_flag = false;
        eventstart = false;
        TICKET = false;
        state = RUN_J;
      }

      // Not all data has been pulled/ Pi not ready to RUN / Errors exist
      else {
        state = DATAPULL_J;
      }

      break;


    case RUN_J:

      // Device order delay
      if (i2c_address == ARD_ADD_1) {
        delay(1800);
      }
      else if (i2c_address == ARD_ADD_2) {
        delay(900);
      }

      digitalWrite(GREEN, HIGH);
                                  // LED indicator
      digitalWrite(WHITE, HIGH);
                                // LED indicator


      if (estop_user == true) {
        digitalWrite(GREEN, LOW);
                                   // LED indicator
        digitalWrite(WHITE, LOW);
                                  // LED indicator
        state = STOP;
      }

      else if (RUNNING == false) {

#ifdef STSPIN220
        MtrCtrl_init_220(step_config);
                    // setup motor controller
#else
        MtrCtrl_init_820(step_config);
                    // setup motor controller
#endif

        if (motor_dir == 1) {
          digitalWrite(DIR, LOW);
        }

        else {
          digitalWrite(DIR, HIGH);
        }


#ifdef TESTING
        Serial.print("freq");
        Serial.println(freq);
#endif

        Timer1.initialize((1 / freq) * 1000000);
                // Init PWM freq[microsec] (mult to conv to usec)
        Timer1.pwm(STEP, DC);
                // Start motor
        RUNNING = true;
        state = RUN_J;
      }

      else {
        state = RUN_J;
      }


      break;


    case RUN_S:

      // Device order delay
      if (i2c_address == ARD_ADD_1) {
        delay(1800);
      }
      else if (i2c_address == ARD_ADD_2) {
        delay(900);
      }

      digitalWrite(GREEN, HIGH);
                                // LED indicator

      if (estop_user == true) {
        digitalWrite(GREEN, LOW);
                                  // LED indicator
        state = STOP;
        Timer1.disablePwm(STEP);
      }

      // keep ard here until finish is sent
      else if (FINISH == true) {
        digitalWrite(GREEN, LOW);
                                  // LED indicator
        Timer1.disablePwm(STEP);
        state = RUN_S;
      }

      // init run of motor
      else if (RUNNING == false) {

#ifdef STSPIN220
        MtrCtrl_init_220(step_config);
                    // setup motor controller
#else
        MtrCtrl_init_820(step_config);
                    // setup motor controller
#endif

        if (motor_dir == 1) {
          digitalWrite(DIR, FORWARD);  // rewrite DIR pin for direction
        }
        else {
          digitalWrite(DIR, REVERSE);
        }

        Timer1.initialize((1 / freq) * 1000000);
                // Init PWM freq[microsec] (mult to conv to usec)
        time_stamp_start = millis();
                // Start recording time
        Timer1.pwm(STEP, DC);
                // Start motor
        RUNNING = true;
        state = RUN_S;
      }

      // Running until final time
      else {

        time_stamp_end = millis();
        time_stamp_end = (time_stamp_end - time_stamp_start);       // total elapsed time
        if ( time_stamp_end / 1000 >= duration) {
          FINISH = true; // if time had reached user's request
        }
        state = RUN_S;
      }


      break;
```

```
    case STOP:

      digitalWrite(STBY, LOW);
                            // Place motor controller in standby
      Timer1.disablePwm(STEP);
                            // turn off pwm

      digitalWrite(GREEN, LOW);
                                // LED indicator
      digitalWrite(WHITE, LOW);
      digitalWrite(YELLOW, LOW);
                                // LED indicator
      digitalWrite(BLUE, LOW);

      q_user = 0;
      syr_radius_user = 0;
      freq = 0;
      duration = 0;
      hour_user = 0;
      min_user = 0;
      sec_user = 0;
      motor_dir = 0;
      step_config = 0;

      estop_user = false;
      dp_checkpoint = false;
      eventstart = false;
      RUNNING = false;
      TICKET = false;
      FINISH = false;
      datapull_flag = false;
      time_flag = false;
      redo_flag = false;

      poll_user = 0;
      error_code = NO_ERR;
      recData = 0;
      strRX[200] = {0};
      dur_mark = 0;

      state = STANDBY;
      break;

    default:
      state = STOP;
      break;


  }
}



void recvEvent(int numBytes) {

  unsigned long temp;

  recData = Wire.read();

  // Ignore data unless recv BLOCK_CODE
  if (block_recv == 0) {
    //Ignore data is not a block (its rollcall or i2cdetect instead)
    if (recData == BLOCK_DATA) {
      //block[0] = recData;
      block_recv++;
    }
  }


  // Recv first data block
  else if (block_recv == 1) {
    block[1] = recData;
    block_recv++;

#ifdef TESTING
    Serial.print("block[1] : ");
    Serial.println(block[1]);
#endif
  }

  // Recv second data block
  else if (block_recv == 2) {
    block[2] = recData;
    block_recv++;

#ifdef TESTING
    Serial.print("block[2] : ");
    Serial.println(block[2]);
#endif

  }

  // Recv third data block
  else if (block_recv == 3) {
    block[3] = recData;
    block_recv++;

#ifdef TESTING
    Serial.print("block[3] : ");
    Serial.println(block[3]);
#endif
  }

  // Recv fourth data block
  else if (block_recv == 4) {
    block[4] = recData;
    block_recv++;

#ifdef TESTING
    Serial.print("block[4] : ");
    Serial.println(block[4]);
#endif
  }

  // Recv fifth data block
  else if (block_recv == 5) {
    block[5] = recData;
    block_recv++;

#ifdef TESTING
    Serial.print("block[5] : ");
    Serial.println(block[5]);
#endif

  }


  // If entire block has been recv, process data
  if (block_recv == 6) {

    //if second factor is non-zero, data uses fac2, rem2
    if (block[4] != 0) {
      temp = 256 * (256 * block[4] + block[5]) + block[3];
    }

    // Only one factor of 256
    else if (block[2] != 0) {
      temp = 256 * block[2]  + block[3];
    }

    // Recv data is below 256
    else {
      temp = block[3];
    }

#ifdef TESTING
    Serial.print("temp : ");
    Serial.println(temp);
#endif
```

```
    //reset index
    block_recv = 0;


    // Read Dict Codes to store in right variable
    switch (block[1]) {

      case POLLDATA:
        poll_user = temp;
        break;

      case Q_DATA:
        q_user = temp;
        q_user /= 1000000000;
        break;

      case R_DATA:
        syr_radius_user = temp;
        syr_radius_user /= 1000;
        break;

      case CAP_DATA:
        syr_capacity = temp;
        syr_capacity /= 1000;
        break;

      case DUR_DATA:
        if (dur_mark == 0) {
          hour_user = temp;
          dur_mark++;
        }
        else if (dur_mark == 1) {
          min_user = temp;
          dur_mark++;
        }
        else if (dur_mark == 2) {
          sec_user = temp;
          dur_mark = 0;
        }

        break;

      case DIR_DATA:
        motor_dir = temp;
        datapull_flag = true;

#ifdef TESTING
        Serial.println(poll_user);
        Serial.println(q_user, 10);
        Serial.println(syr_radius_user, 8);
        Serial.println(syr_capacity);
        Serial.println(hour_user);
        Serial.println(min_user);
        Serial.println(sec_user);
        Serial.println(motor_dir);
#endif

        break;

      case EVENT_DATA:
        eventstart = true;
        break;

      case REDO_DATA:
        redo_flag = temp;
        Serial.println("STOPPED");
        state = STOP;
        break;

      case ESTOP_DATA:
        estop_user = true;
        break;

      default:
        error_code = DICT_ERR;
        break;
    }
  }

}


/* ~~~~~~~~~~~~~~~~~~~~~~~~ reqEvent()
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    Purpose : Write data to main at its request
    Input : I2C request
    Output : TX corresponding error code
*/
void reqEvent() {


  // Tell Pi RUN_S has finished
  if (FINISH == true) {
    Wire.write(FINISH_CODE);
    estop_user = true;
  }


  // If in RUN_S but enough time hasn't elapsed,
    send time_code for next TX
  else if (state == RUN_S && time_flag == false) {
    Wire.write(TIME_CODE);
    time_flag = true;
  }



  // Once Pi has recv time_code, send the time
    metrics
  else if (state == RUN_S && time_flag == true) {

    // convert time_end to sec from ms
    unsigned long time_end = time_stamp_end / 1000;

#ifdef TESTING
    Serial.print("time_end : ");
    Serial.println(time_end);
#endif

    if (dur_mark == 0) {
      hour_user = int(time_end / 3600);
      Wire.write(hour_user);
      dur_mark++;

#ifdef TESTING
      Serial.print("hour_user : ");
      Serial.println(hour_user);
#endif

    }

    else if (dur_mark == 1) {
      min_user = int(( time_end - (3600 * hour_user)
    ) / 60);
      Wire.write(min_user);
      dur_mark++;

#ifdef TESTING
      Serial.print("min_user : ");
      Serial.println(min_user);
#endif

    }

    else if (dur_mark == 2) {
      sec_user = int( time_end - (3600 * hour_user)
    - (60 * min_user));
      Wire.write(sec_user);
      dur_mark = 0;
      time_flag = false;


#ifdef TESTING
```

```
      Serial.print("sec_user : ");
      Serial.println(sec_user);
#endif

    }
  }



  // Tell Pi to wait if ard is still calculating/validating TICKET
  else if (dp_checkpoint == false && state != RUN_S) {
    Wire.write(WAIT_CODE);
           // Ard hasnt had time to validate TICKET yet
  }



  // Return error (or no error) in DATAPULL
  else {
    if (error_code != NO_ERR) {
           // If errors exist, prepare for another datapull
      dp_checkpoint = false;
      datapull_flag = false;
    }
    Wire.write(error_code);
    // TX error
#ifdef TESTING
    Serial.print("Error code : ");
    Serial.println(error_code, DEC);
#endif

    error_code = NO_ERR;

  }
}


#endif
```

Listing 1: Motor Controller Code

B. Hub Controller

```
# Connor Wilson
# Winter 2022
# SMSS Hub Controller V1.0


from time import sleep                    # python delay
from smbus2 import SMBus
from transitions import Machine           # state machine
import curses                             # GUI
from curses import wrapper
from curses.textpad import Textbox
import random



class HubMachine(object):

    states = ['START', 'DATAPULL', 'RUN', 'RESET']

    def __init__(self,name):
        self.name = name

        # States and their corresponding triggers/transitions
        self.machine = Machine(model= self, states = HubMachine.states, initial = 'START')
        self.machine.add_transition(trigger = 'CONFIRM', source = 'START', dest = 'DATAPULL')
        self.machine.add_transition(trigger = 'EVENTSTART', source = 'DATAPULL', dest = 'RUN')
        self.machine.add_transition(trigger = 'DONE', source = 'RUN', dest = 'RESET')
        self.machine.add_transition(trigger = 'RESTART', source = 'RESET', dest = 'START')


# States for GUI
global FRAME, POLL, DATA, RUN, ERROR, USER_CONFIRM, USER_ESTOP, reqGUI
FRAME = 0x0
POLL = 0x1
DATA = 0x2
ERROR = 0x3
RUN = 0x4
USER_CONFIRM = 0x6
USER_ESTOP = 0x7
reqGUI = 0x0          # Working var for GUI states

# Index for device spacing in the GUI
global column
column = 0

# Data between user-curses-arduino
global rollcall, poll_user, q_user, radius_user, cap_user
global hour_user, min_user, sec_user, dir_user, back_flag
global estop_flag, ard_duration
rollcall = [0, 0, 0]
poll_user = [0, 0, 0]
q_user = [0,0,0]
radius_user = [0,0,0]
cap_user = [0,0,0]
hour_user = [0,0,0]
min_user = [0,0,0]
sec_user = [0,0,0]
dir_user = [0,0,0]
                # 1 - REV, 0 - FWD
back_flag = [False, False, False]
                # User sel to go back
estop_flag = [False, False, False]
                # User sel for device estop
ard_duration = [[0,0,0], [0,0,0], [0,0,0]]
                # run time from ard

# Arduino's Requested (Sub) codes
global LARGE_ERR, SMALL_ERR, DUR_ERR, LD_ERR, SD_ERR, DICT_ERR, NO_ERR
global WAIT_CODE, FINISH_CODE, TIME_CODE
global user_code

LARGE_ERR = 0x3                         # User req too large (Q) for hardware constraints
SMALL_ERR = 0x4                         # User req too small (Q) for hardware constraints
DUR_ERR = 0x5                           # User req too long (time) for hardware constraints
LD_ERR = 0x6                            # User too large and long
SD_ERR = 0x7                            # User too small and long
DICT_ERR = 0x8                          # Wrong dictating codes
NO_ERR = 0x10                           # No error to report
WAIT_CODE = 0x15                        # Ard needs time to pull data and validate
FINISH_CODE=  0x16                      # Ard's duration has been reached and finished RUN_S
TIME_CODE =  0x17                       # Ard is not done with RUN_S and will return time(h,m,s) if prompted


user_code = 0                           # working variable

global estop_index
estop_index = 0

def main():

    # Working Variables that pass through main and GUI
    global rollcall, poll_user, q_user, radius_user, cap_user
    global hour_user, min_user, sec_user, dir_user, back_flag
    global estop_flag, ard_duration, reqGUI, user_code

    # I2C Addresses
    PI_ADD = 0x01
    ARD_ADD_1 = 0x14
    ARD_ADD_2 = 0x28
    ARD_ADD_3 = 0x42
    address = [ARD_ADD_1, ARD_ADD_2, ARD_ADD_3]

    # Pi's Dictating (main) codes
    POLLDATA =  1                       # Ard will recv[OFF, JOG, START]
    Q_DATA  = 2                         # Ard will recv[q_user]
    R_DATA  = 3                         # Ard will recv[syr_radius_user]
    CAP_DATA = 4                        # Ard will recv[syr_capacity]
    DUR_DATA = 5                        # Ard will recv[hour_user, min_user, sec_user]
    DIR_DATA = 6                        # Ard will recv[DIRECTION]
    EVENT_DATA = 7                      # Ard will proceed RUN
    ESTOP_DATA = 8                      # Ard will proceed to STOP
    REDO_DATA = 9                       # Ard will revert back to STANDBY from DATAPULL
    BLOCK_DATA = 10                     # Ard will recv a block (4) of data
```

```

    # Arduino's Requested (Sub) codes
    user_code = 0                           # working
    var for reading ERR/CODE from Ard

    #availability of arduino deivces 0 = false, 1 =
    true
    rollcall = [0,0,0]

    # GUI Window constants
    global column
    COL_CONST = 7
    COL_OFFSET = 5


    #curses window
    stdscr = curses.initscr()

    #state machine init
    Hub = HubMachine("SMSS")

    # Loop states
    while True:

        # Inital device check and device run-state
    selection
        if Hub.state == 'START' :

            #Identify devices on the buffer
            rollcall = deviceRollcall(address)

            #Display GUI background frame
            #global reqGUI
            reqGUI = FRAME
            curs(stdscr)

            #reset devices
            for i in range(3):
                if rollcall[i] > 0:
                    transmit_block(address[i],
    ESTOP_DATA, True )


            # Gather device run-state in GUI and TX
    to Ard
            for i in range(3):

                # GUI set to poll user for run-state
                reqGUI = POLL

                #if device is active
                if rollcall[i] > 0:

                    #set the correct column spacing
                    #global column
                    column = COL_CONST * i +
    COL_OFFSET

                    # Go to GUI
                    curs(stdscr)
                    transmit_block(address[i],
    POLLDATA, poll_user[i] )

                else:
                    #print("Dont got'em")
                    pass


            # loop if all devices are off
            if (poll_user[0] == 0 and poll_user[1]
    == 0 and poll_user[2] == 0) :
                pass

            # Change GUI windows to DATAPULL
            else:
                reqGUI = USER_CONFIRM
                curs(stdscr)
                Hub.CONFIRM()

        # Pull data from user using the Hub GUI
        elif Hub.state == 'DATAPULL' :

            #for all three devices
            for i in range(3):

                # if device is active
                if poll_user[i] > 0:

                    loop = True
        # loop to weed out user error
                    data_preserve = False
        # flag to redo user data

                    # update window column on GUI
                    column = COL_CONST * i +
    COL_OFFSET

                    # loop until all errors are out
                    while loop == True:

                        # Present data collection
    GUI
                        reqGUI = DATA
                        curs(stdscr)

                        # if user selected to go
    back to poll GUI instead
                        if back_flag[i] == True:

                            #reset flag
                            back_flag[i] = False

                            # transmit redo flag
                            transmit_block(address[i
    ], REDO_DATA, True )

                            # collect data again
                            reqGUI = POLL
                            curs(stdscr)
                            transmit_block(address[i
    ], POLLDATA, poll_user[i] )
                            reqGUI = DATA
                            curs(stdscr)

                            #loop to transmit again
                            data_preserve = False
            # TX new data to ard again
                            loop = True
            # continue to monitor for user error

                            #if user turns off
    device in redo, set device to standby
                            if poll_user[i] == 0:
                                data_preserve =
    True       # dont write unwanted data


                        # Send data batch (False) or
     keep data (True)
                        if data_preserve == False:

                            # if device is in JOG
                            if poll_user[i] == 1:
                                # transmit variables
                                transmit_block(
    address[i], Q_DATA, int(q_user[i]))
                                transmit_block(
    address[i], R_DATA, int(radius_user[i]) )
                                transmit_block(
    address[i], DIR_DATA, int(dir_user[i]) )


                            # if device is in RUN
```

```
                            elif poll_user[i] == 2:
                                transmit_block(
address[i], Q_DATA, int(q_user[i]) )
                                transmit_block(
address[i], R_DATA, int(radius_user[i]) )
                                transmit_block(
address[i], CAP_DATA, int(cap_user[i]) )
                                transmit_block(
address[i], DUR_DATA, int(hour_user[i]) )
                                transmit_block(
address[i], DUR_DATA, int(min_user[i]) )
                                transmit_block(
address[i], DUR_DATA, int(sec_user[i]) )
                                transmit_block(
address[i], DIR_DATA, int(dir_user[i]) )

                        #if user turns off device in
 redo, dont loop for error removal
                        if poll_user[i] == 0:
                            loop = False

                        else:
                            # read bus for error
codes
                            with SMBus(1) as bus:
                                user_code = bus.
read_byte(address[i], 0)

                            # pass if no errors are
present (exit condition)
                            if user_code == NO_ERR:
                                data_preserve =
False       # reset var
                                loop = False
                    # dont loop for errors (none exist ..
hopefully)

                            # loop if told to wait (
and check again for errors)
                            elif user_code ==
WAIT_CODE:
                                data_preserve = True
                    # dont trash data yet
                                loop = True
                    # loop to check for error though

                            # proceed to error
screen and loop this again
                            else:
                                reqGUI = ERROR
                    # go to GUI screen to show error
                                curs(stdscr)
                                data_preserve =
False       # transmit a new batch of data
                                loop = True
                    # loop to prompt user for better data









                    # if all present devices have tickets,
Hub.EVENTSTART()
                    reqGUI = USER_CONFIRM
                    curs(stdscr)
                    for i in range(3):
                        if poll_user[i] != 0:
                            transmit_block(address[i],
EVENT_DATA, True )
                    Hub.EVENTSTART()

                # Run motors, present estops, and monitor
devices
                elif Hub.state == 'RUN' :

                    # var to save redundant time checking
                    time_saver = [0,0,0]

                    # Print intro RUN window if device is on
                    reqGUI = RUN
                    for i in range(3):
                        if rollcall[i] > 0:
                            column = COL_CONST * i +
COL_OFFSET
                            curs(stdscr)

                    # Loop if at least one device is on
                    while (poll_user[0] != 0 or  poll_user
[1] != 0  or  poll_user[2] != 0) :

                        #Poll for an estop
                        reqGUI = USER_ESTOP
                        curs(stdscr)

                        for i in range(3):

                            # If user has marked one or all
estops during a pull, send tx and stop
                            if estop_flag[0] == True and
poll_user[0] > 0:
                                transmit_block(address[0],
ESTOP_DATA, True )
                                poll_user[0] = 0

                            if estop_flag[1] == True and
poll_user[1] > 0:
                                transmit_block(address[1],
ESTOP_DATA, True )
                                poll_user[1] = 0

                            if estop_flag[2] == True and
poll_user[2] > 0:
                                transmit_block(address[2],
ESTOP_DATA, True )
                                poll_user[2] = 0

                            # if it is time to sample the
RUN_S state for FINISH or TIME
                            if time_saver[i] > 300 and
poll_user[i] == 2:

                                # Read ard status (FINISH or
 TIME_CODE)
                                with SMBus(1) as bus:
                                    user_code = bus.
read_byte(address[i], 0)

                                    #if FINISH, dont repoll
device
                                    if user_code ==
FINISH_CODE:
                                        poll_user[i] = 0
                                        #FINISH SCREEN

                                    # if time_code, read
data and store
                                    elif user_code ==
TIME_CODE:

                                        global ard_duration

                                        with SMBus(1) as bus
:
                                            for j in range
(3):
                                                ard_duration
[i][j]  = bus.read_byte(address[i], 0)

                                        # Print time data
                                        reqGUI = RUN
                                        column = COL_CONST *
 i + COL_OFFSET
                                        curs(stdscr)
```

```
                                #reset var
                                time_saver[i] = 0

                    elif time_saver[i] > 300 and poll_user[i] == 1:
                        # Print time data
                        reqGUI = RUN
                        column = COL_CONST * i + COL_OFFSET
                        curs(stdscr)
                        time_saver[i] = 0


                    time_saver[i] += 1

            #Proceed if all devices are finished
            Hub.DONE()



        #reset variables
        elif Hub.state == 'RESET' :
            rollcall = [0, 0, 0]
            poll_user = [0, 0, 0]
            q_user = [0,0,0]
            radius_user = [0,0,0]
            cap_user = [0,0,0]
            hour_user = [0,0,0]
            min_user = [0,0,0]
            sec_user = [0,0,0]
            dir_user = [0,0,0]
            back_flag = [False, False, False]
            estop_flag = [False, False, False]
            ard_duration = [[0,0,0], [0,0,0], [0,0,0]]
            user_code = 0
            global estop_index
            estop_index = 0

            Hub.RESTART()


# Error_2Str()
# SIGNATURE : int -> String
# PURPOSE : Takes in Error code and returns a string
#           that aids the user in correcting their data
#           requests
def Error_2Str(error):

    global LARGE_ERR, SMALL_ERR, DUR_ERR, LD_ERR, SD_ERR, DICT_ERR, NO_ERR

    if error == LARGE_ERR:
        return "Error: Flow rate too large for hardware"
    elif error == SMALL_ERR:
        return "Error : Flow rate too small for hardware"
    elif error == DUR_ERR:
        return "Error : Run time will ruin machine"
    elif error == LD_ERR:
        return "Error : Too large flow rate + time overflow"
    elif error == SD_ERR:
        return "Error : Too small of Q + time overflow"
    elif error == DICT_ERR:
        return "Error : Ard got lost...GULP"
    else:
        return f"Your error code is unique... : {error}"



# transmit_block()
# SIGNATURE : int, int, int ->
# PURPOSE : Transmits a dictating code +  data to an address
#
def transmit_block(addr, dict_code, int_data):

    # Code for ard to prep for a block
    BLOCK_DATA = 10

    #       [block, dict_code, factor1, remainder1, factor2, remainder2]
    block = [BLOCK_DATA, dict_code, 0, int_data, 0, 0]

    #(can only handle 256 ^3)...increase for larger data set
    i = 2

    while int_data > 255:
        block[i] = int(min(int_data / 256, 256))
    # store factor of 256
        block[i+1] = int(int_data % 256)
    # store remainder of 256
        int_data /= 256
    # break down data
        i += 2

    #transmit data
    with SMBus(1) as bus:
        for i in range(6):
            bus.write_byte(addr, block[i])
            sleep(0.15)


# deviceRollcall()
# SIGNATURE : int_array -> int_array
# Purpose : Sends byte to each device using I2C and
#           Stores 1 in array for a sucessful transmission (device active)
#           Stores 0 in array for unsuccessful transmission (device not active)
def deviceRollcall(myaddress):

    dev_status = [0, 0, 0]

    with SMBus(1) as bus: # write to BUS

        for i in range(3):

            # Try to contact each device
            try:
                bus.write_byte(myaddress[i], 0)
                dev_status[i] = 1
                #print(f"Device {i} on bus")
            except:
                #print(f"Device {i} not on bus")
                dev_status[i] = 0
                #pass

    sleep(0.1)
    return dev_status


# GUI
def curs(stdscr):

    global FRAME, POLL, DATA, RUN, ERROR, USER_CONFIRM, USER_ESTOP
    global reqGUI
    global rollcall, poll_user, q_user, radius_user, cap_user
    global hour_user, min_user, sec_user, dir_user
    global estop_index, user_code


    # Color combinations (ID, foreground, background)
```

```
    curses.init_pair(1, curses.COLOR_RED, curses.COLOR_WHITE)
    RED_AND_WHITE = curses.color_pair(1)

    curses.init_pair(2, curses.COLOR_BLACK, curses.COLOR_WHITE)
    BLACK_AND_WHITE = curses.color_pair(2)

    curses.init_pair(3, curses.COLOR_BLACK, curses.COLOR_GREEN)
    BLACK_AND_GREEN = curses.color_pair(3)

    curses.init_pair(4, curses.COLOR_YELLOW, curses.COLOR_CYAN)
    YELLOW_AND_CYAN = curses.color_pair(4)

    curses.init_pair(5, curses.COLOR_WHITE, curses.COLOR_MAGENTA)
    WHITE_AND_MAGENTA = curses.color_pair(5)

    curses.init_pair(6, curses.COLOR_BLACK, curses.COLOR_RED)
    BLACK_AND_RED = curses.color_pair(6)



    # Display background frame
    if reqGUI == FRAME:

        stdscr.clear()
        stdscr.attron(YELLOW_AND_CYAN)
        stdscr.resize(28,100)
        stdscr.move(0,0)

        # Fill square main in
        for y in range(1,28):
            for x in range(1, 100-1):
                stdscr.addch(" ")
            stdscr.move(y,1)
        stdscr.border()

        #Title block
        stdscr.move(2,30)
        stdscr.addstr("        Smart Motor Syringe Pump        ", BLACK_AND_WHITE)



        stdscr.attroff(YELLOW_AND_CYAN)
        stdscr.attron(BLACK_AND_WHITE)

        #Device one block
        stdscr.move(5,10)
        stdscr.addstr("Device 1 :")
        stdscr.move(5,30)

        for y in range(5,11):
            for x in range(30, 90-1):
                stdscr.addch(" ")
            stdscr.move(y,30)

        # Device two block
        stdscr.move(12,10)
        stdscr.addstr("Device 2 :")
        stdscr.move(12,30)

        for y in range(12,18):
            for x in range(30, 90-1):
                stdscr.addch(" ")
            stdscr.move(y,30)

        # Device three block
        stdscr.move(19,10)
        stdscr.addstr("Device 3 :")
        stdscr.move(19,30)

        for y in range(19,25):
            for x in range(30, 90-1):
                stdscr.addch(" ")
            stdscr.move(y,30)


        #Re-center and refresh
        stdscr.move(21,40)
        stdscr.addstr(25, 35, "
                              ")
        stdscr.move(5,30)
        stdscr.refresh()


    # Poll user for device run modes
    elif reqGUI == POLL:

        global poll_user
        #avoiding a global index here people... forgive me coding gods
        index_gui = int((column-5)/7)


        if rollcall[index_gui] > 0:
            # display window
            winPoll1 = curses.newwin(5, 60, column, 30)
            curses.noecho()
            winPoll1.nodelay(True)
            winPoll1.clear()
            winPoll1.attron(BLACK_AND_GREEN)
            winPoll1.addstr(1, 5,f"Device {index_gui + 1} Mode :      | OFF | JOG | RUN |")
            winPoll1.refresh()

            # collect data
            q = 0
            key = ""

            while key != 'w':

                winPoll1.attroff(BLACK_AND_WHITE)
                winPoll1.attron(BLACK_AND_GREEN)
                winPoll1.addstr(1, 5, f"Device {index_gui + 1} Mode :      | OFF | JOG | RUN |")

                try:
                    key = winPoll1.getkey()
                except:
                    key = None

                if key == 'a':
                    if q > 0:
                        q -= 1

                elif key == 'd':
                    if q < 2:
                        q += 1


                if q == 0:
                    winPoll1.attroff(BLACK_AND_GREEN)
                    winPoll1.attron(BLACK_AND_WHITE)
                    winPoll1.addstr(1, 26, "| OFF |")
                    poll_user[index_gui] = 0
                # disable the device

                elif q == 1:
                    winPoll1.attroff(BLACK_AND_GREEN)
                    winPoll1.attron(BLACK_AND_WHITE)
                    winPoll1.addstr(1, 32, "| JOG |")
                    poll_user[index_gui] = 1
                # jog the device
```

```
            elif q == 2:
                winPoll1.attroff(BLACK_AND_GREEN)
                winPoll1.attron(BLACK_AND_WHITE)
                winPoll1.addstr(1, 38, "| RUN |")
                poll_user[index_gui] = 2
            # run the device


            winPoll1.refresh()
            sleep(0.5)


# Simple enter button
elif reqGUI == USER_CONFIRM:

    # display window
    winPoll4 = curses.newwin(1, 46, 25, 35)
    curses.noecho()
    # dont repeat user input
    winPoll4.nodelay(True)
    # dont wait user input
    winPoll4.clear()
    winPoll4.attron(curses.color_pair(random.randint(1,6))) #randomly change color
    winPoll4.addstr(0, 18, "| CONFIRM? |")
    winPoll4.refresh()

    key = ""

    while key != 'w':

        try:
            key = winPoll4.getkey()
        except:
            key = None

    winPoll4.refresh()
    sleep(0.5)

# Poll metrics for running in START
elif reqGUI == DATA:

        #avoiding a global index here people... forgive me coding gods
        index_gui = int((column-5)/7)

        # User wants to disable the device (leave in standby - no update)
        if poll_user[index_gui] == 0:
            pass

        # User selected JOG function
        elif poll_user[index_gui] == 1:

            winData1 = curses.newwin(5, 60, column, 30)
            winData1.attron(WHITE_AND_MAGENTA)


            winData1.addstr(0, 0, "Volumetric Flow Rate Q [nL/s] in [70n, 10u] : ")
            winData1.refresh()
            win_data_ret = curses.newwin(1, 10, column, 76)
            curses.echo()
            box = Textbox(win_data_ret)
            box.edit()
            q_user[index_gui] = box.gather().strip().replace("\n", "")


            winData1.addstr(1, 0, "Syringe Radius in [mm] : ")
            win_data_ret.clear()
            win_data_ret.mvwin(column + 1,55)
            winData1.refresh()
            box.edit()
            radius_user[index_gui] = box.gather().strip().replace("\n", "")


            winData1.addstr(2, 0, "Direction ? | FWD | REV | REDO | ? ")
            win_data_ret.clear()
            winData1.refresh()

            winData1.attroff(WHITE_AND_MAGENTA)
            winData1.attron(BLACK_AND_GREEN)

                        # collect data
            curses.noecho()
            winData1.nodelay(True)
            q = 0
            key = ""
            while key != 'w':
                try:
                    key = winData1.getkey()
                except:
                    key = None

                if key == 'a':
                    if q > 0:
                        q -= 1

                elif key == 'd':
                    if q < 2:
                        q += 1

                winData1.attroff(BLACK_AND_GREEN)
                winData1.attron(WHITE_AND_MAGENTA)
                winData1.addstr(2, 0, "Direction ?        | FWD | REV | REDO | ? ")
                winData1.attroff(WHITE_AND_MAGENTA)
                winData1.attron(BLACK_AND_GREEN)

                if q == 0:
                    winData1.addstr(2, 19, "| FWD |")
                    dir_user[index_gui] = 1
                    back_flag[index_gui] = False

                elif q == 1:
                    winData1.addstr(2, 25, "| REV |")
                    dir_user[index_gui] = 0
                    back_flag[index_gui] = False

                elif q == 2:
                    winData1.addstr(2, 31, "| REDO |")
                    back_flag[index_gui] = True

                winData1.refresh()
                sleep(0.5)

        # User sel RUN function
        elif poll_user[index_gui] == 2:

            winData1 = curses.newwin(5, 60, column, 30)
            winData1.attron(WHITE_AND_MAGENTA)


            winData1.addstr(0, 0, "Volumetric Flow Rate Q [nL/s] in [70n, 10u] : ")
            winData1.refresh()
            win_data_ret = curses.newwin(1, 10, column, 76)
            curses.echo()
```

```
                box = Textbox(win_data_ret)
                box.edit()
                q_user[index_gui] = box.gather().strip().replace("\n", "")


                winData1.addstr(1, 0, "Syringe Radius in [mm] : ")
                win_data_ret.clear()
                win_data_ret.mvwin(column + 1,55)
                winData1.refresh()
                box.edit()
                radius_user[index_gui] = box.gather().strip().replace("\n", "")

                winData1.addstr(2, 0, "Syringe Capacity in [mL] : ")
                win_data_ret.clear()
                win_data_ret.mvwin(column + 2,58)
                winData1.refresh()
                box.edit()
                cap_user[index_gui] = box.gather().strip().replace("\n", "")

                winData1.addstr(3, 0, "Hour # : ")
                win_data_ret.clear()
                win_data_ret.mvwin(column + 3,40)
                winData1.refresh()
                box.edit()
                hour_user[index_gui] = box.gather().strip().replace("\n", "")

                winData1.addstr(3, 16, "Minute # : ")
                win_data_ret.clear()
                win_data_ret.mvwin(column + 3,58)
                winData1.refresh()
                box.edit()
                min_user[index_gui] = box.gather().strip().replace("\n", "")

                winData1.addstr(3, 35, "Second # : ")
                win_data_ret.clear()
                win_data_ret.mvwin(column + 3,77)
                winData1.refresh()
                box.edit()
                sec_user[index_gui] = box.gather().strip().replace("\n", "")


                winData1.addstr(4, 0, "Direction ?       | FWD | REV | REDO | ? ")
                win_data_ret.clear()
                winData1.refresh()

                winData1.attroff(WHITE_AND_MAGENTA)
                winData1.attron(BLACK_AND_GREEN)

                            # collect data
                curses.noecho()
                winData1.nodelay(True)
                q = 0
                key = ""
                while key != 'w':
                    try:
                        key = winData1.getkey()
                    except:
                        key = None

                    if key == 'a':
                        if q > 0:
                            q -= 1

                    elif key == 'd':
                        if q < 2:
                            q += 1

                    winData1.attroff(BLACK_AND_GREEN)
                    winData1.attron(WHITE_AND_MAGENTA)
                    winData1.addstr(4, 0, "Direction ?        | FWD | REV | REDO | ? ")
                    winData1.attroff(WHITE_AND_MAGENTA)
                    winData1.attron(BLACK_AND_GREEN)

                    if q == 0:
                        winData1.addstr(4, 19, "| FWD |")
                        dir_user[index_gui] = 1
                        back_flag[index_gui] = False

                    elif q == 1:
                        winData1.addstr(4, 25, "| REV |")
                        dir_user[index_gui] = 0
                        back_flag[index_gui] = False

                    elif q == 2:
                        winData1.addstr(4, 31, "| REDO |")
                        back_flag[index_gui] = True

                    winData1.refresh()
                    sleep(0.5)


# JOG-RUN metrics
elif reqGUI == RUN:

        #avoiding a global index here people... forgive me coding gods
        index_gui = int((column-5)/7)

        # User wants to disable the device ( leave in standby - no update)
        if poll_user[index_gui] == 0:
            pass


         # User sel JOG function
        elif poll_user[index_gui] == 1:

            winRun1 = curses.newwin(5, 60, column, 30)
            winRun1.attron(BLACK_AND_RED)

            winRun1.addstr(0, 0, f" Q [nL/s] : {q_user[index_gui]}    Syringe Radius in [mm] : {radius_user[index_gui]}")
            #winRun1.addstr(4, 18, "| ESTOP |")
            winRun1.refresh()

         # User selected RUN function
        elif poll_user[index_gui] == 2:
            winRun1 = curses.newwin(5, 60, column, 30)
            winRun1.attron(BLACK_AND_RED)

            winRun1.addstr(0, 0, f" Q [nL/s] : {q_user[index_gui]}    Syringe Radius in [mm] : {radius_user[index_gui]}")
            winRun1.addstr(1, 0, f" Requested - Hour : {hour_user[index_gui]}    Minute : {min_user[index_gui]}   Second : {sec_user[index_gui]}")
            winRun1.addstr(2, 0, f" Elapsed - Hour : {ard_duration[index_gui][0]}       Minute :  {ard_duration[index_gui][1]}     Second : {ard_duration[index_gui][2]}    ")
            #winRun1.addstr(4, 18, "| ESTOP |")
            winRun1.refresh()
```

```
            #sleep(1)


    # Simple ESTOP button
    elif reqGUI == USER_ESTOP:

        COL_CONST = 7
        COL_OFFSET = 5

        global estop_flag

        # display window
        winStop = curses.newwin(1, 60, 25 , 30)
        curses.noecho()
        winStop.nodelay(True)
        winStop.clear()
        winStop.refresh()

        key = ""
        try:
            key = winStop.getkey()
        except:
            key = None

        if key == 'w':
            if estop_index > 0:
                estop_index -= 1

        elif key == 's':
            if estop_index < 3:
                estop_index += 1


        if estop_index < 3 and poll_user[estop_index
    ] > 0:
            winStop.mvwin(estop_index * COL_CONST +
    9, 30)
            winStop.attron(curses.color_pair(
    estop_index))
            winStop.addstr(0,18, "| ESTOP? |")
            winStop.refresh()

        elif estop_index == 3:
            winStop.mvwin(25, 30)
            winStop.attron(BLACK_AND_RED)
            winStop.addstr(0,18, "| ESTOP ALL? |")
            winStop.refresh()


        if key == 'd':

            # stop indiv device
            if estop_index != 3:
                estop_flag[estop_index] = True
                winStop.attron(BLACK_AND_GREEN)
                winStop.addstr(0,18, "| STOP'D |")
                winStop.refresh()

            # stop all devices
            elif estop_index == 3:
                estop_flag[0] = True
                estop_flag[1] = True
                estop_flag[2] = True
                estop_index = 0


    # Presents error string in device window
    elif reqGUI == ERROR:
        #avoiding a global index here people...
    forgive me coding gods
        index_gui = int((column-5)/7)

        # display window
        winPoll4 = curses.newwin(5, 60, column, 30)
        curses.noecho()
        winPoll4.nodelay(True)
        winPoll4.clear()
        winPoll4.attron(RED_AND_WHITE)
        winPoll4.addstr(2, 10, Error_2Str(user_code)
    )
        winPoll4.refresh()


        sleep(8)


    else:
        print("Requested GUI Error")



wrapper(curs)



if __name__ == "__main__":
    main()
```

Listing 2: Hub Controller Code

APPENDIX D
ADDITIONAL DOCUMENTS

A. Business Model Graphic

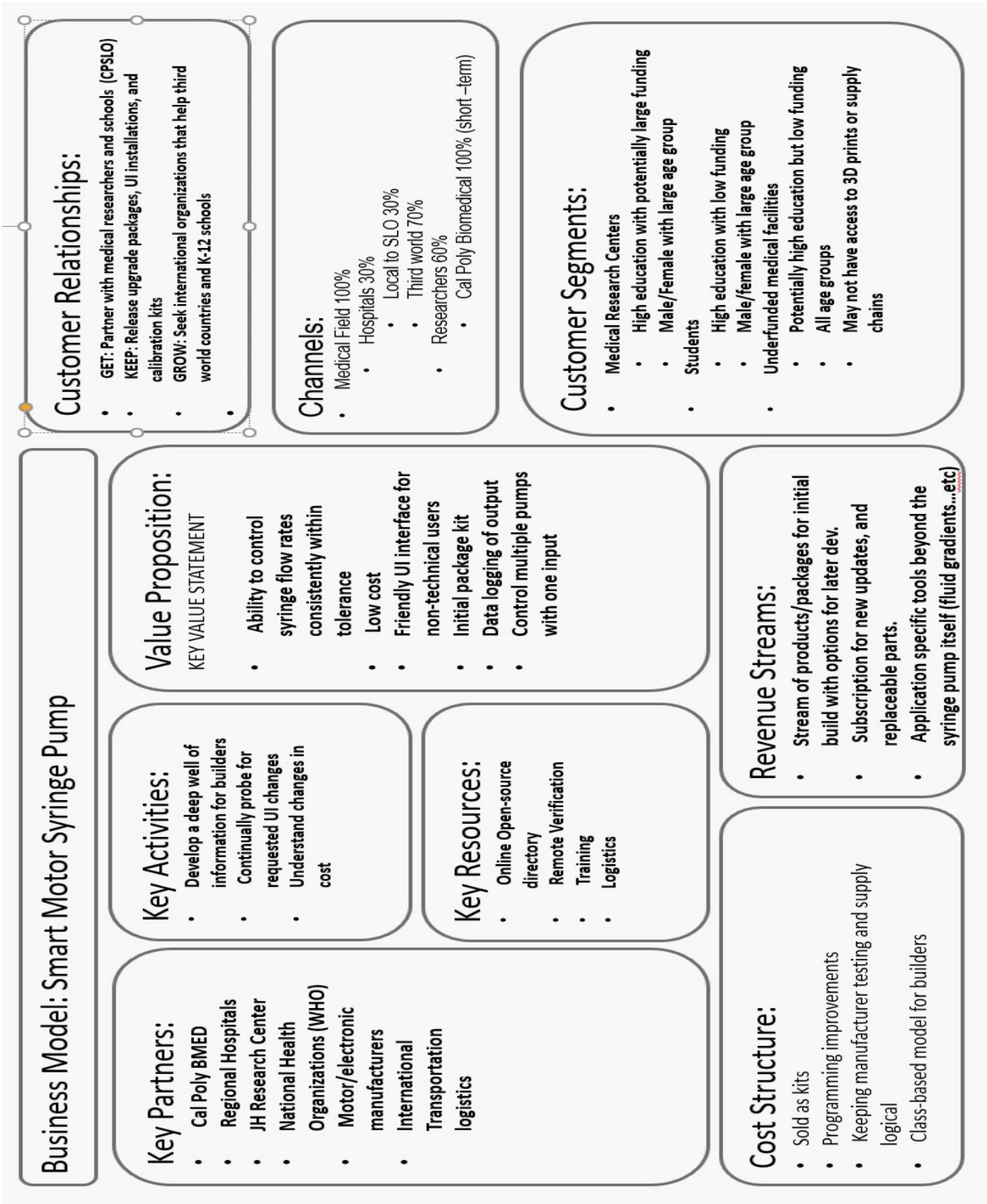

Fig. 47: Business Model Graphic

B. Marketing Data Graphic

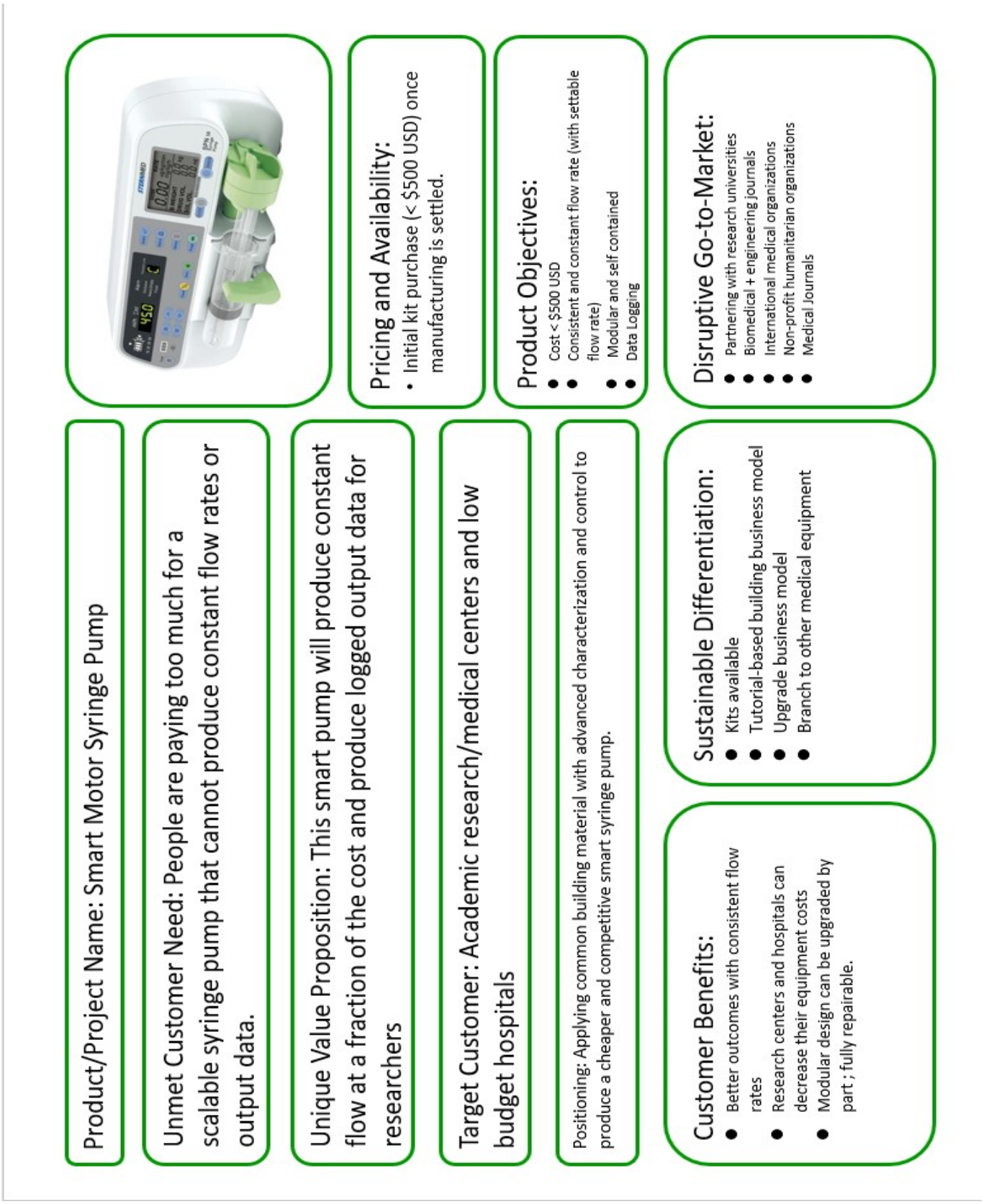

Fig. 48: Marketing Data Graphic

C. GANTT Chart

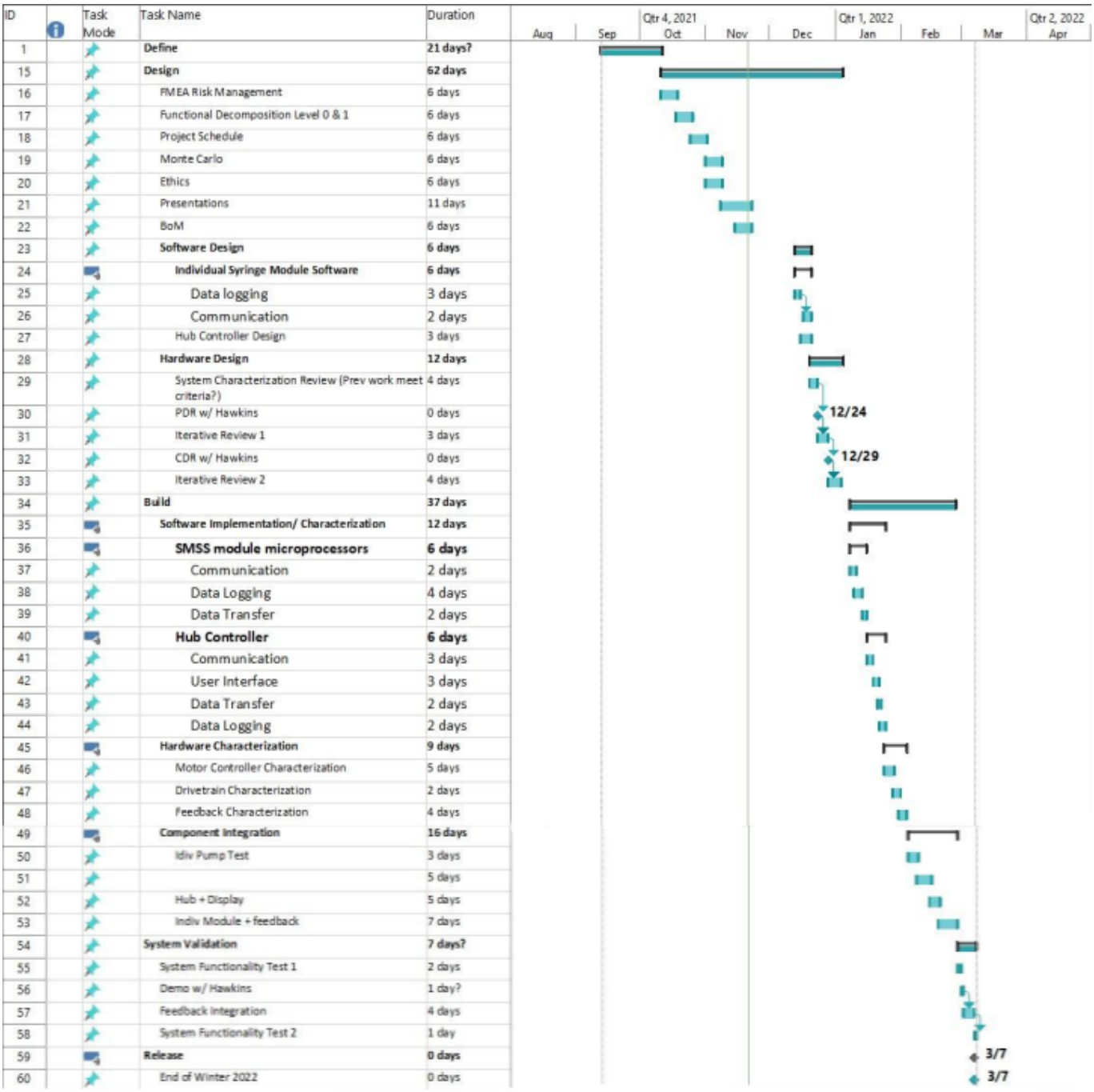

Fig. 49: Gantt Chart for Current Version of the SMSS

D. Bill of Materials

		Senior Project:	Smart Motor Syringe Pump									
		TEAM:	Connor Wilson									
		DATE/REV:	March 1, 2022 VERSION V1.1									
												USA
Sub	Sub	Sub	Sub	Description	Qty	P/N	Document	MFG	MFG P/N	LT - Weeks	1kpcs Price (USD)	
1	Final Assy - Unit Smart Motor Syringe Pump				1	1000						
	1	Modular Switching Power Supply			1	1010		XP Power	LCW150PS24	0	40.0000	
	2	Hub Controller (Raspberry Pi)			1	1020		Raspberry	Model B 4gb	1	45.0000	
		1	Mini-B USB Cables		1	1021		stock		0		
		2	Keyboard		1	1022		stock		0		
		3	Monitor		1	1023		stock		0		
		4	HDMI Cable		1	1024		stock		0		
		5	12.5W Power Supply/Adapter		1	1025		CanaKit	DCAR-RSP-2A5	1	9.9500	
	3	Pump Module				1030						
		1	Arduno Nano		3	1031		Arduino	7630049200173	1	20.7000	
		2	100:1 Planetary Gear Box + Stepper Motor		3	1032		Nema 17	17HS19-1684S-PG1001	0	30.1500	
		3	Screwgear Drivetrain		3	1033		Unknown		0	40.0000	
		4	Motor Mounting Bracket		3	1034		In House		0		
		5	Syring Mounting Bracket		3	1035		In House		0		
		6	1K Resistor		12	1036		stock		0		
		7	LED (YELLOW, GREEN, BLUE, WHITE) 3ea.		12	1037		stock		0		
		8	Electrolytic Capacitor 100uF (35V)		1	1038		stock		0		
		9	Electrolytic Capacitor 47uF (35V)		6	1039		stock		0		
		10	Stepper Motor Driver		3	1040		Pololu		5	8.9700	
	4	I2C Bi-Directional Logic Converter			1	1050		Sparkfun	BOB-12009	1	2.9500	
2	Various screws/nuts/washers (Metric M3-M6)				1	1060		stock		0		
3	Jumper Wire				1	1070		stock		0		
4	Breadboard				4	1080		stock		0		
									Material Total	397.3600		
						If made in USA	0.5hrs x $40/hr			45.00		
						If made in Foreign	0.5hrs x $5/hr		Labor+OH			
						Product Shipping leadtime			Weeks	2		
									Total	444.36		

Fig. 50: Bill of Materials for Current Version of the SMSS

E. CAD Work

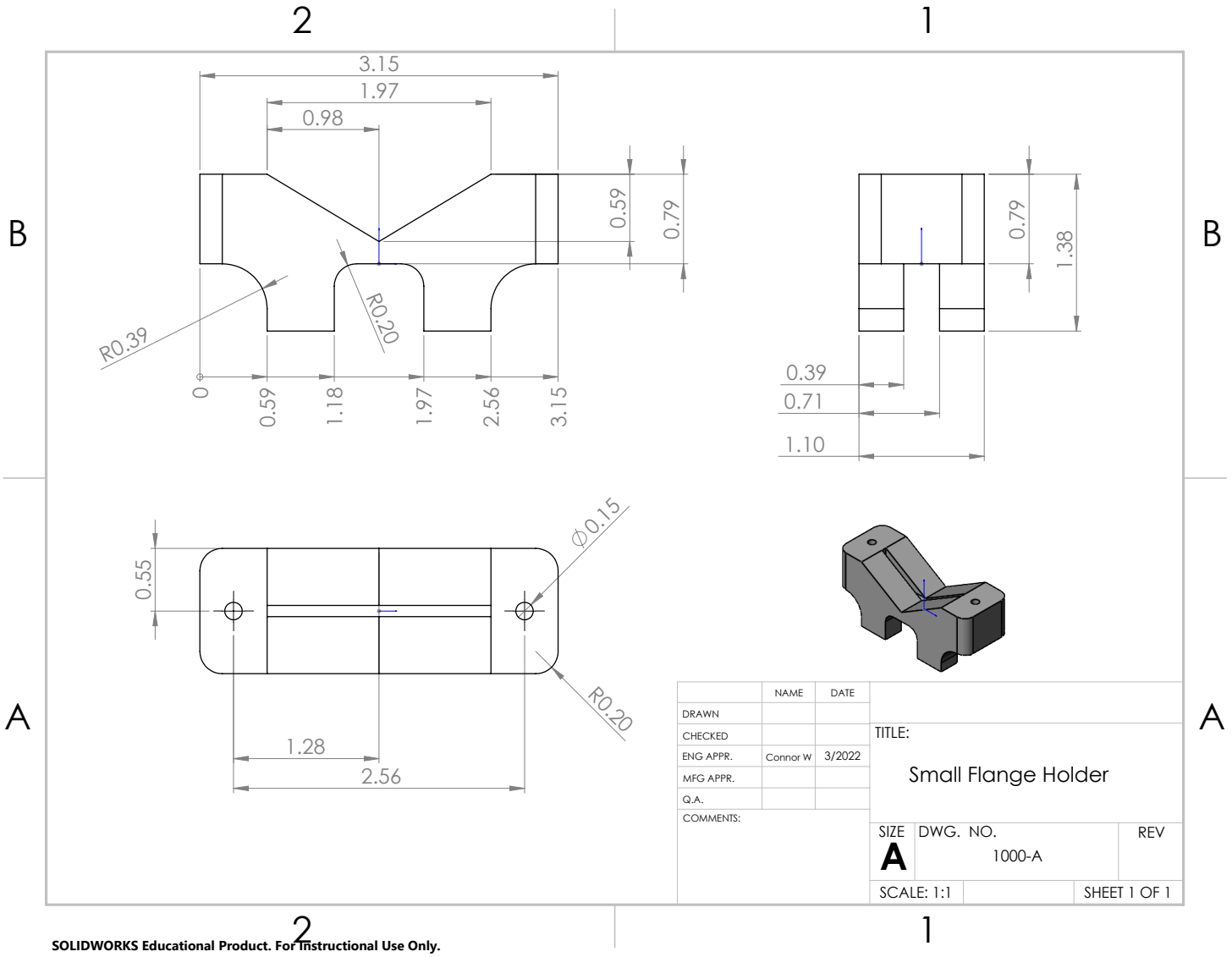

Fig. 51: Flange Holder for Smaller 1205 Ball Screw

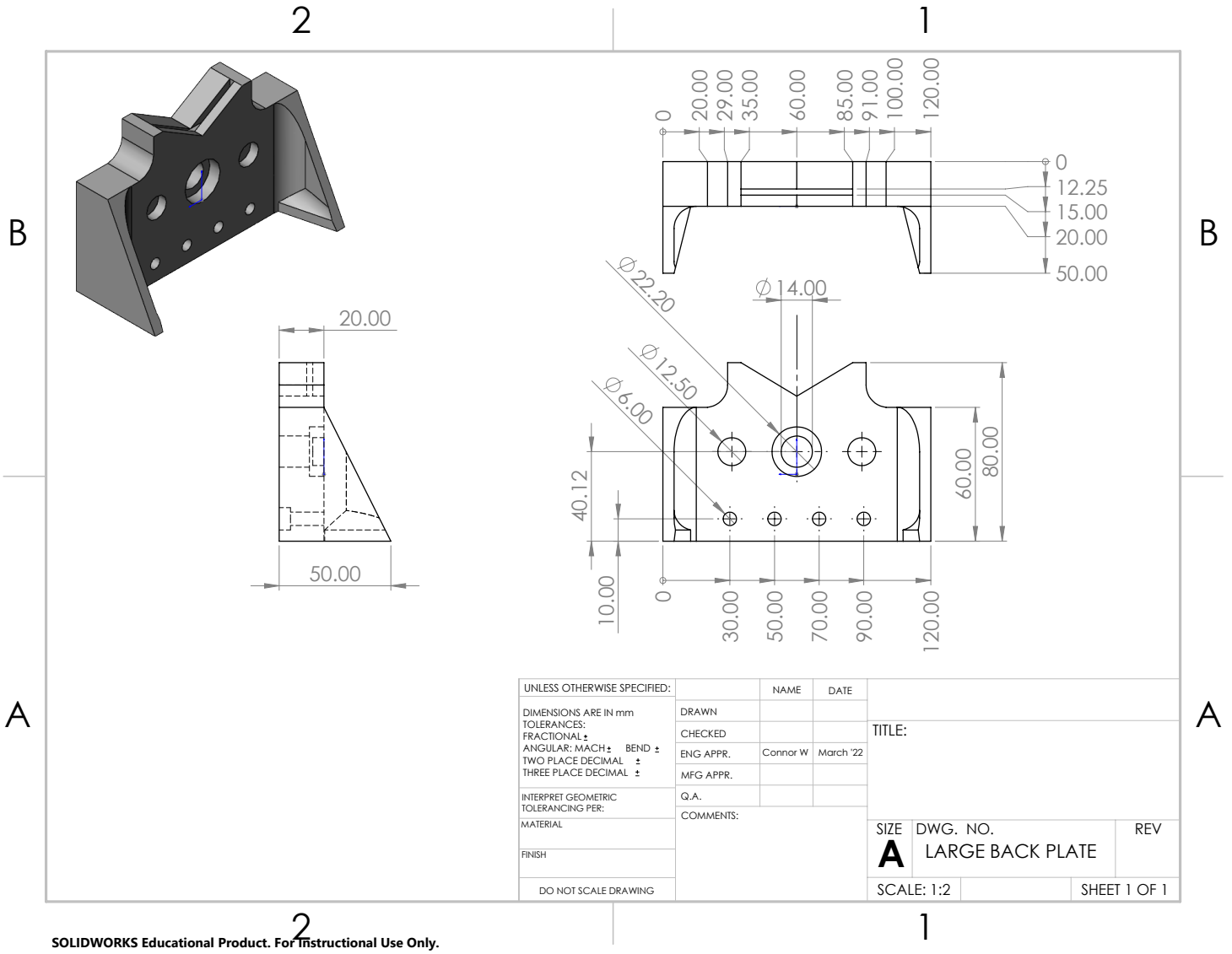

Fig. 52: Flange Holder for Larger 1605 Ball Screw

F. Motor Controller Flow

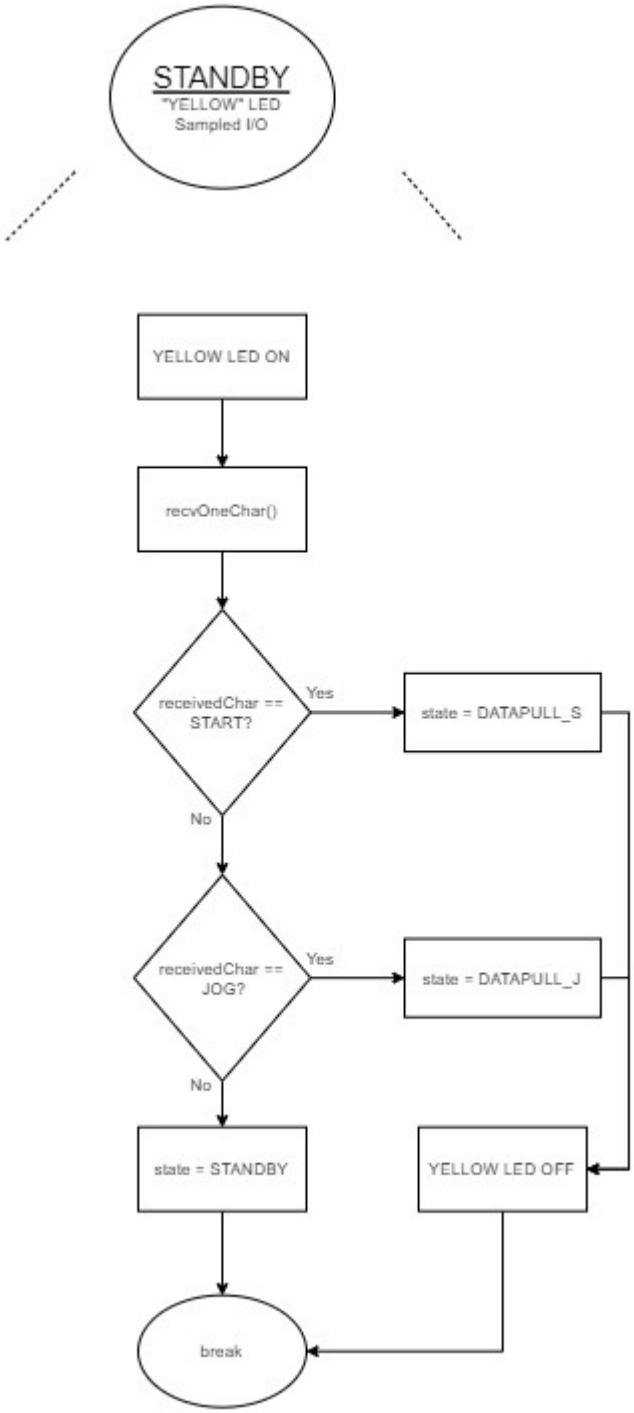

Fig. 53: STANDBY Logic Flow for Motor Controller

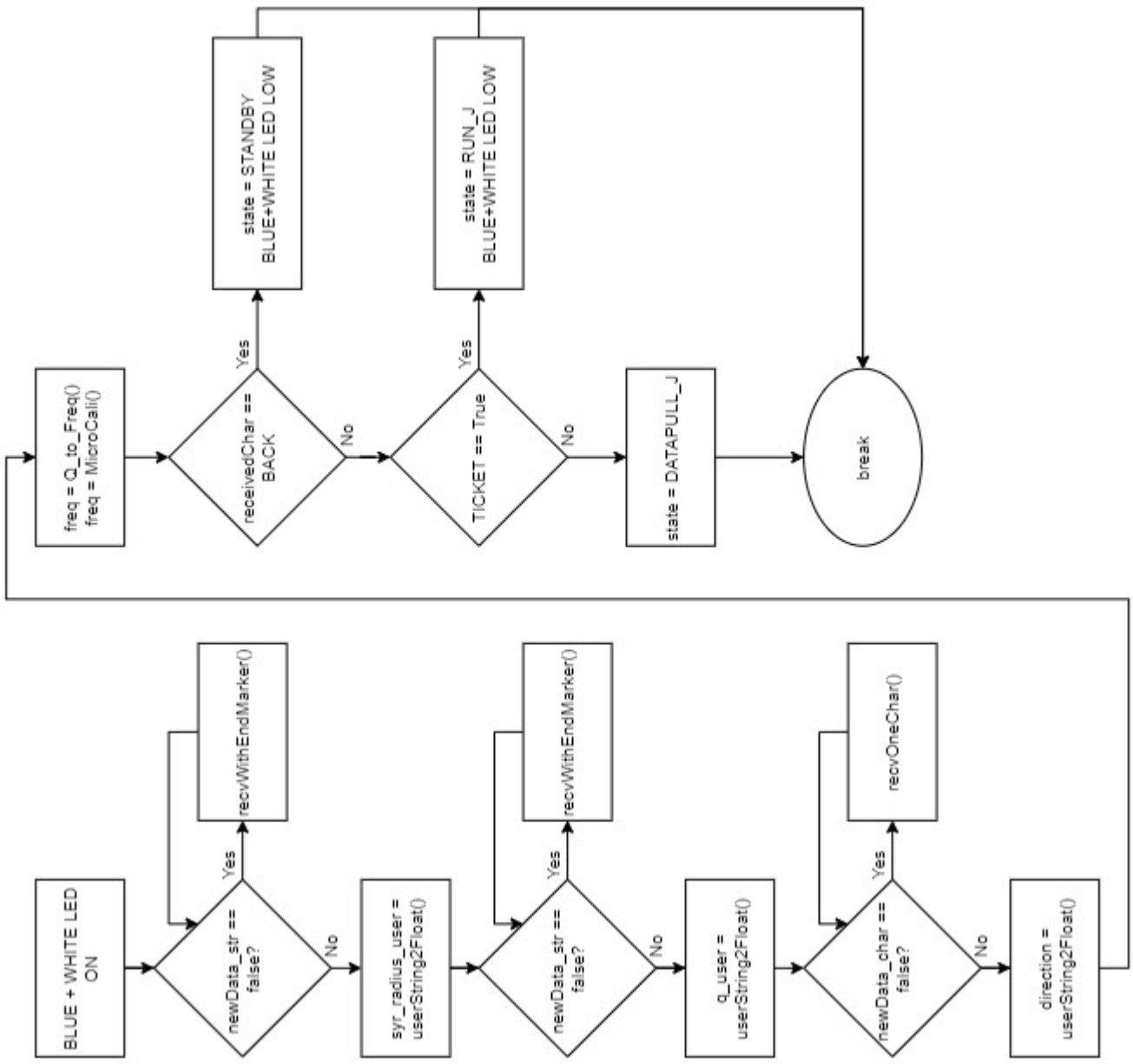

Fig. 54: DATAPULL_J Logic Flow for Motor Controller

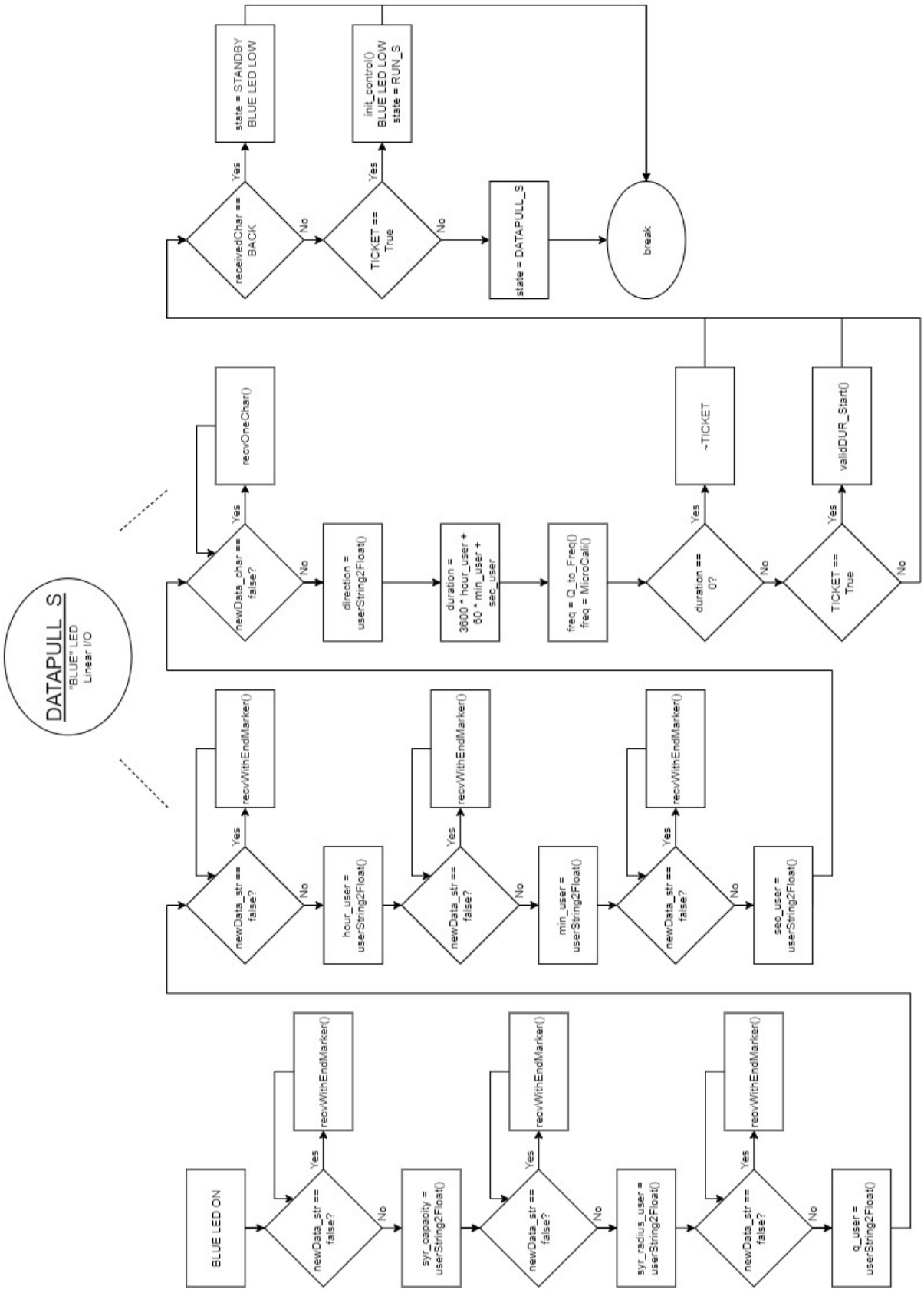

Fig. 55: DATAPULL_S Logic Flow for Motor Controller

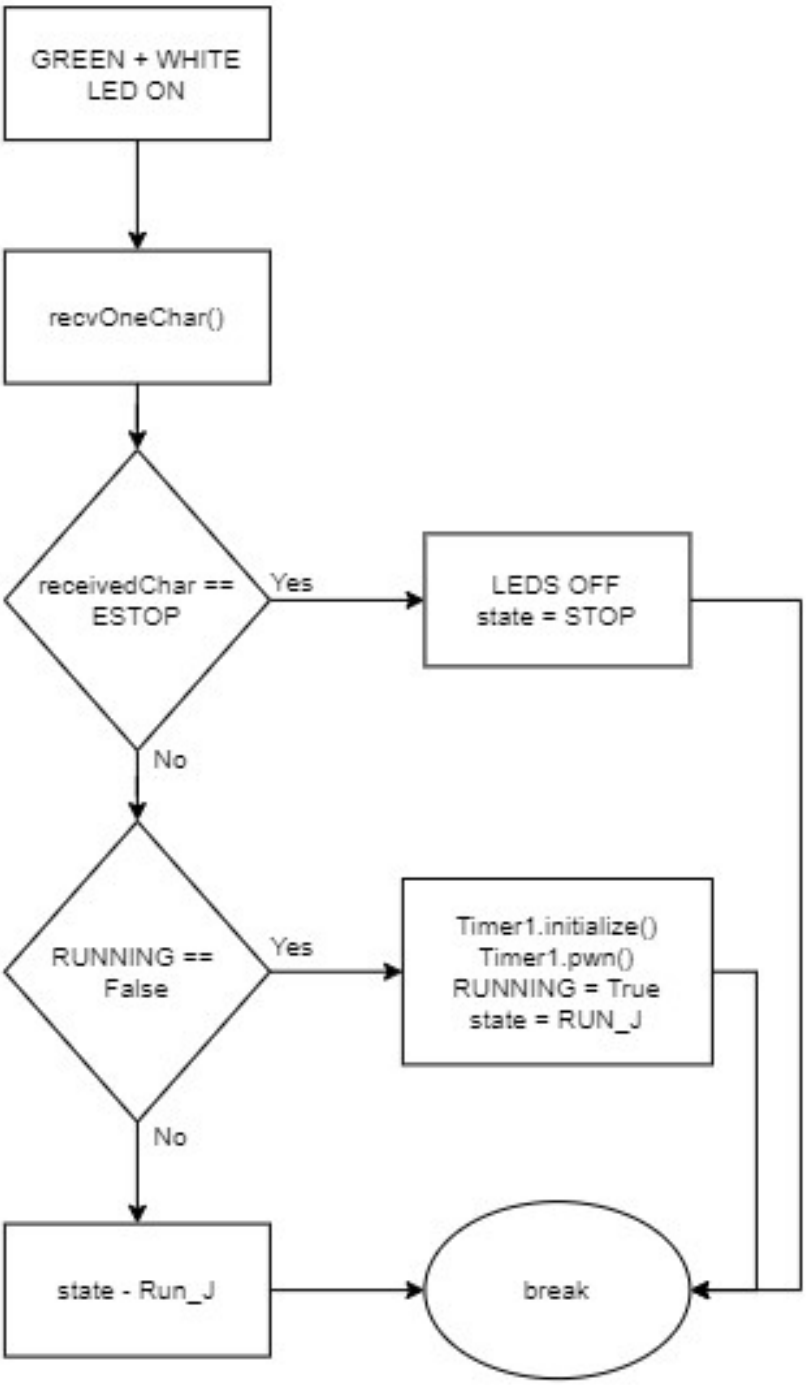

Fig. 56: RUN_J Logic Flow for Motor Controller

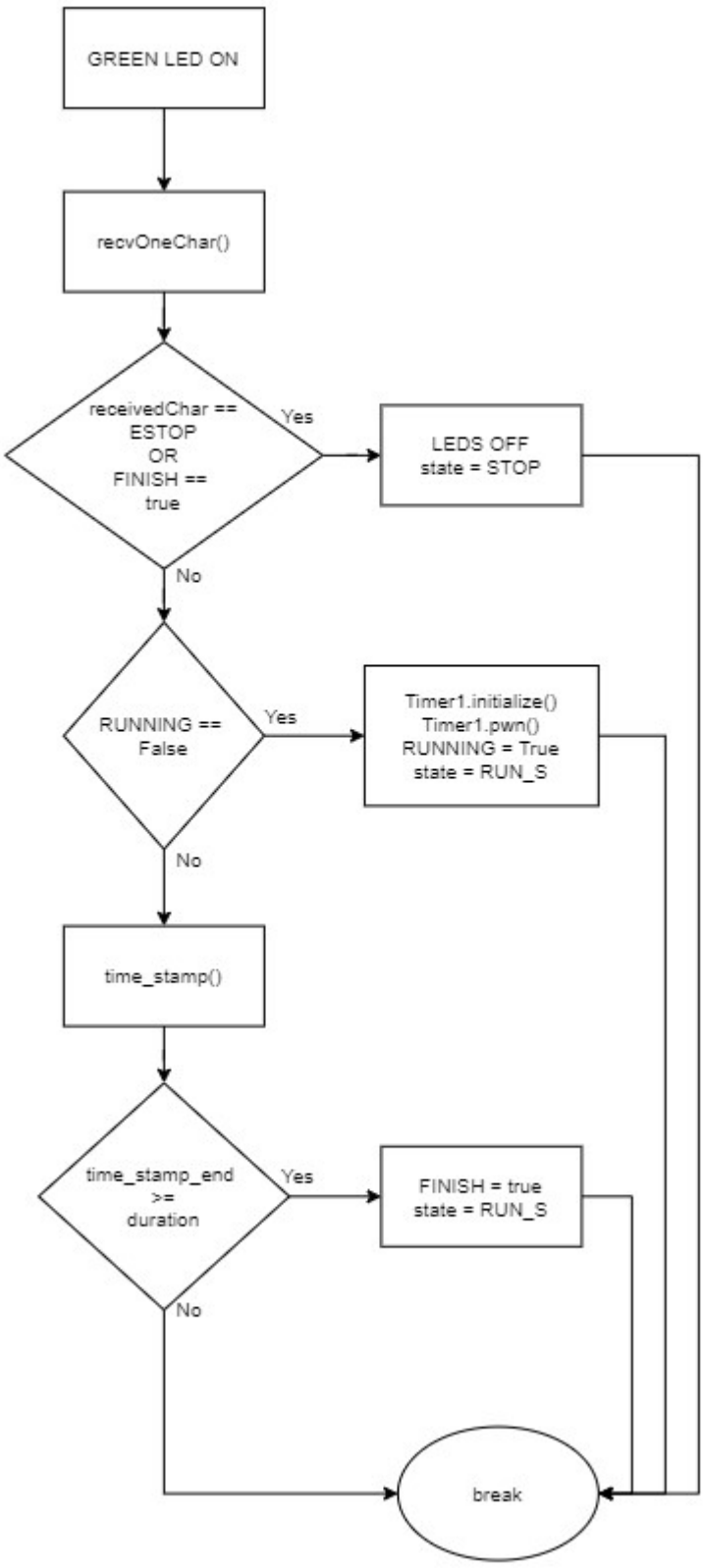

Fig. 57: RUN_S Logic Flow for Motor Controller

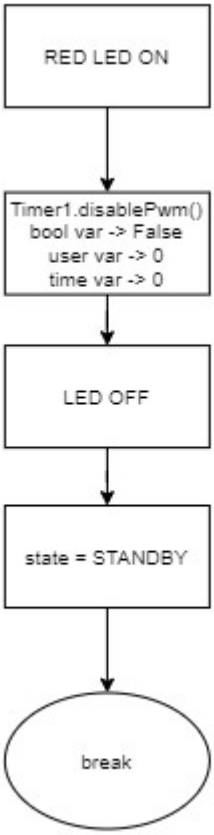

Fig. 58: STOP Logic Flow for Motor Controller

G. External Links

- Github: github.com/Cwilson-WhisperingRock/SMSS22

Acknowledgment

The author would like to thank

- Dr. Ben Hawkins for the guidance and knowledge to achieve success in a project such as this.
- Conrad Chan for putting forth amazing work for which this project stands on.
- My Fiancée for humouring me in my engineering rambles.

References

[1] Cision PR Newswire. *Global Intravenous Infusion Pump Market (2021 to 2030) - Featuring Baxter International, Terumo and Medtronic Among Others*. URL: https://www.prnewswire.com/news-releases/global-intravenous-infusion-pump-market-2021-to-2030---featuring-baxter-international-terumo-and-medtronic-among-others-301416604.html.

[2] Wikipedia. *Syringe Driver*. URL: https://en.wikipedia.org/wiki/Syringe_driver.

[3] AMIS Medical. *What is a Syringe Pump?* URL: https://amismedical.com/what-is-syringe-pump/.

[4] American Institute of Physics. *Do teaching and communicating about microfluidics advances need improvement?* URL: https://phys.org/news/2019-07-microfluidics-advances.html.

[5] Conrad Chan and Jenny Chiao. *Smart Motor Syringe System*. Tech. rep. CA: California Polytechnic San Luis Obispo, Dec. 2020.

[6] CHEMYX. *What is a Syringe Pump?* URL: https://www.chemyx.com/support/knowledge-base/application-reference-by-topic/what-is-a-syringe-pump-3/.

[7] Google Trends. *Google Trend Search for (Syringe Pump)*. URL: https://trends.google.com/trends/explore?q=syringe%5C%20pump&geo=US.

[8] MedicalExpo.com. *Medical Manufactures of Syringe Pumps*. URL: https://www.medicalexpo.com/medical-manufacturer/1-channel-syringe-pump-2074.html.

[9] Heyden Kirk Pittman. *Stepper Motor Theory*. URL: https://www.haydonkerkpittman.com/learningzone/technicaldocuments/stepper-motor-theory.