Spring 2023

Whiteboard Drawing Device

Stefan Ilic
si28@uakron.edu

Andrew Adams
aka73@uakron.edu

Vaughn Richards
vr42@uakron.edu

James Medved
jam529@uakron.edu

Whiteboard Drawing Device

Stefan Ilic – Honors Project Individual Contribution

As the hardware manager, my responsibility was to oversee all hardware components used in the project. Given that we were using three stepper motors, one servo motor, four limit switches, and a pic board, there needed to be sufficient power so all these components could operate appropriately. After various calculations and assessments, we implemented a 120-12 V AC to DC converter. The max current that the converter could provide was 2.5 amps, which was enough for the needed limit switch to kick on while all three stepper motors were working simultaneously. The scenario was chosen as the worst-case scenario as this required the most amount of power used at a single time. A second 120-12 V AC to DC converter was used to drive the pic board separately.

Due to the nature of motor drivers needing logic high to be about 5 Vdc, I implemented a 12 to 5 Vdc converter to ensure that the appropriate motor driver logic pins could be driven with the needed voltage. Also, given that the limit switches need between 4-7 volts, we powered all four limit switches using this converter.

Besides the hardware aspect, I assisted the team with building the apparatus and with the wire management.

Whiteboard Drawing Device

Project Design Report

DT02

Drew Adams

Stefan Ilic

James Medved

Vaughn Richards

Dr. Andrew Milks

4/24/2023

TABLE OF CONTENTS

List of Tables:

List of Figures:

ABSTRACT

Whiteboards have become a staple in many classrooms, especially in the context of STEM courses. However, when it comes to teaching complex topics, it can be difficult or impractical to provide accurate drawings and visualizations by hand, and in extreme cases may lead to misinterpretation of a particular topic. To fix this issue, the Whiteboard Drawing Device provides users with the ability to draw preset shapes to a whiteboard automatically, both quickly and accurately. Using either a smartphone application or a set of on-board controls, the device allows users to automatically draw straight lines, rectangles, triangles, sinusoidal waves, and circles to the whiteboard. The shape, size, and position of each of these shapes on the board are defined by the user, and the device can be retrofitted to a whiteboard of max size 4’ × 4’. With this, the Whiteboard Drawing Device offers a user-friendly, low cost, and fun alternative to other more expensive products with similar functionality. **(VR)**

PROBLEM STATEMENT

(1.1) Need

Whiteboards play a crucial role within the modern education system. Whether it be elementary schools, colleges, or even seminars, whiteboards play a key role in portraying the information visually to whoever it is intended for. They allow teachers to be flexible with their teaching styles, resulting in an increased student engagement, having the ability to quickly erase mistakes and continue the lecture, and managing the class tempo depending on the students' understanding of the material. These boards provide versatility throughout the education system [5]. However, like most things, shortcomings are inevitable. These are in STEM fields (as well as others) where shapes and images are essential. Therefore, having a device where shapes can be drawn with a simple touch of a button would significantly reduce the time the instructor would need to spend doing these tasks manually, and prevent potential confusion if not drawn correctly. **(SI & DA)**

(1.2) Objective

The objective of this project is to design a whiteboard geometry attachment tool to assist teachers with geometric and trigonometric drawings. The device will be able to retrofit to any whiteboard, and allow users to input their desired dry-erase marker. Manual controls will be on the device with preloaded shape presets, while remote control can be performed over bluetooth via an app. Shape, size, and position of the drawing will all be variable by the user. **(JM & VR)**

(1.3) Background

Not everyone is able to learn effectively with the same learning style. The VARK model established by Flemming and Mills (1992) separates learners into four different categories: visual, auditory, reading/writing, and kinesthetic [1]. Visual learners prefer seeing information rather than hearing it. This information could be in the form of drawings, graphs, pictures, videos, etc. Auditory learners are capable of retaining information spoken verbally. In the case of math, these individuals tend to translate the information into a list of steps that could be used as the format for similar problems [2]. Reading/writing learners prefer to learn through text. Kinesthetic learners are most effective when the material has real world or hands-on application. In other words, they

may have an easier time solving a physical problem, such as a Rubik's Cube, rather than a written problem. Students may be partial to only one or any combination of these methods. **(JM)**

Most mathematics can be presented in a variety of ways to fit the various VARK learning styles. For example, assume there is a system of the form:

$$f_1(x) = x$$

and one wants to multiply the system by some constant α, like so:

$$f_2(x) = \alpha x$$

The effect that the constant α has on the system can be conveyed to the students in a few different ways. The instructor could verbally explain the resulting system, write it down, or represent it graphically. For auditory learners, the first or second option may be best. This is because they tend to be better at memorization and application of formulas such as the one above, even if they do not fully understand the underlying mathematical concepts [2]. In contrast, visual learners would be partial to the third option since they would be able to see the original system and how the translation affects the system in graphical form. **(JM)**

The whiteboard geometry attachment device aims to help instructors educate visual learners as effectively as learners inclined to other VARK learning styles. In a quote from Avoiding Math Taboos: Effective Math Strategies for Visual-Spatial Learners, Whitney H. Rapp states, "The visual learner needs to see the information rather than hear it in order to make sense of it. This is not the same as an auditory processing disorder. The visual spatial learner can decipher auditory input, but needs to translate it into visual images if any true learning and application is to occur." [2] In an attempt to help visual learners better understand material taught in class, this device will allow instructors to draw multiple geometric and trigonometric shapes both quickly and accurately. Some of these include basic shapes such as circles and rectangles, sinusoids, and x-y planes. This in turn can help with student's understanding of challenging mathematical concepts by associating said concepts with their illustrated representations. **(JM)**

One existing and effective device that is similar to this proposed project is the SMART board. SMART boards are exceptional at replicating shapes and assisting instructors in the classroom. Many schools have implemented SMART boards and have proven them effective in enhancing the learning experience of students. A study was done in which two groups of seventh grade students were used to observe the effect of SMART boards on knowledge retention and exam performance. In summary, the group of students taught using the SMART board reported greater success than those taught without using the SMART board [3]. **(SI & DA & JM & VR)**

However, these SMART boards have their shortcomings. SMART board attachments are required to be fixed to an existing whiteboard or wall. When considering a college campus that could have hundreds of classrooms, installing a SMART board in every room would be time consuming and extremely costly to the university. Also, teachers who are not very proficient with technology may struggle with these boards. Along with daily startup and general use, these boards are prone to lag, which creates additional difficulties during strict course curriculum schedules, disrupting lesson schedules of instructors and the overall education of students [4]. **(SI & DA)**

Using the proposed device rather than a smart board would free up additional board space. In contrast to SMART boards, the proposed device would not be fixed to the center of a board. By using a set of home coordinates, the proposed device can be tucked away on the side of a board when not in use. Once a task is requested by the user and performed by the device, a return home command can be issued to automatically return the device to its resting spot on the board. **(SI & DA & JM)**

The proposed device would also provide a less expensive and less complicated setup than SMART boards, leading to accessibility for low-income schools. On top of the base price of a SMART board, a projector is also required for use. Projectors add to the total cost of the setup and require frequent calibration, which adds another disruption while teaching. The proposed device would require far fewer components, which would reduce the cost. Also, instead of having to replace the existing white board with a SMART board, the proposed device could simply be attached to a pre-existing board. **(SI & DA & JM)**

Another design similar to the one being proposed is the iBoardbot. The iBoardbot is an internet-controlled whiteboard robot capable of writing text, drawing, and erasing on a small whiteboard. It has a multi-user interface in which graphics can be uploaded to an application. The application then sends a series of commands to the device that allows it to draw these graphics. This device comes disassembled in a kit that can be built by following the assembly instructions [6]. **(SI & DA)**

The iBoardbot and proposed design project have three key differences. The first difference is the size of the two devices. The iBoardbot is very small, only about twelve inches long and six inches wide. In contrast, the proposed device will be much larger, in order to accommodate full scale whiteboards. Also, the iBoardbot is one contained unit with a built-in whiteboard. The proposed device will be able to retrofit onto any wall sized whiteboard, instead of having a board built into it [6]. Lastly, the iBoardbot interfacing is all done using an application. The proposed device will also be controlled using an application, but it will provide physical controls on the unit, as well. These physical controls will be reserved for simple and common commands, such as straight lines, circles, and squares. The application will then be used for more complex instructions, such as drawing the graph of some particular function. **(JM & VR)**

A patent from Aarav Chavda, Isaac Ilivicky, and Winston Soboyejo, shows a device with movement capabilities similar to those we intend to use for the proposed device. This device, known as the Self-Erasing Chalkboard (Patent No. WO2017116995A1), uses "…a transport apparatus…which is an 'X-Y' table, which in a broadest sense, allows for the controlled positioning and repositioning…of a magnetic field generator" [7]. This transport apparatus implements two horizontal tracks, positioned above and below the chalkboard, for X-axis movement. These tracks then guide a vertical beam with a separate apparatus capable of Y-axis movement. Although this patented device is different from the one being proposed because it has been strictly designed for erasing a chalkboard, the positioning technology it uses will be very helpful. For example, the original design project proposed using only one horizontal track positioned above a board, instead of the two horizontal tracks used by the Self-Erasing Chalkboard. This may have resulted in a lack of stability in return for a reduced cost of production. **(VR)**

Another patent known as the Blackboard Circle Drawing Instrument (Patent No. CN202782353U) from Wang Xiuping was found to be particularly useful. This device is not meant to be automatic. Instead, it consists of a telescopic arm that can be attached to a blackboard via a rubber suction cup, similar to how a compass is used to draw a circle on paper [8]. This patent is particularly relevant to the design being proposed because circles are one of the most difficult shapes to draw by hand. Consequently, programming the movement apparatus to draw a circle is rather difficult as well. Using parametric equations, the easiest way to draw a circle would be to have the X-axis motor's movement dictated by a sine wave, and the Y-axis motor's movement dictated by a cosine wave. However, because both motors would need to be properly synchronized and need to approximate the easing of these sinusoids using PWM, it would be difficult to draw a perfect circle. To avoid overcomplication, the idea of adding a rotating arm to the device similar to the one mentioned in this patent may be considered. By attaching the arm to the original transport apparatus and allowing the length of the arm to be specified by the user, the issues presented when drawing a circle with the original apparatus can be avoided altogether. **(VR)**

(1.4) Marketing Requirements

1. The system should be able to retrofit to any whiteboard
2. The system should be able to draw with a marker
3. The system should be able to vary the size and position of the user defined shape
4. The system should have a built in user interface for manual control
5. The system should also be controlled via an external device

ENGINEERING ANALYSIS

(2.1) Circuits

To power the device, two 120-volt AC to 12-volt DC power supplies have been implemented. One of these connects to the PIC board directly, and the other one provides power to the motor driver circuitry attached via solderboard. On the solderboard, the 12 volts are used to power three motor driver chips, as well as a 12 to 5-volt converter. The 5 volts were used to ensure the correct voltage was used to drive the logic high pins on the motor drivers, the servo motor, and the limit switches. **(SI & DA)**

The max current that the 120-volt AC to 12-volt DC converter could provide is 2.5 amps, which is enough for the limit switches to turn on while all three stepper motors are operating. This scenario was chosen as the worst-case scenario, as this requires the largest amount of power used at a single time. The other AC to DC converter plugs directly into the PIC board barrel using the 12 volts. **(SI & DA)**

As mentioned, three motors are used on the apparatus. One stepper motor is used for y-axis movement, and the other two motors are used for movement on the x-axis. Because they always perform the same action, the two *x* motors share an oscillator from the PIC board. Two motors were used on the x-axis because when only one x-axis motor was used at the top of the device, a significant amount of whip would occur at the bottom of the y-axis rail. Once the rig moved left or right, this whip would cause significant inaccuracies while drawing shapes. Therefore, the bottom x and top x-axis motors move in unison to resolve this issue. **(SI & DA)**

(2.2) Electronics

One of the electronic components included is a microcontroller. This microcontroller takes input via onboard controls or Bluetooth, and translates those inputs into the correct motor movements. Along with the microcontroller, the motor driver circuit is the primary hub that bridges what the user inputs with the stepper motors on the apparatus. The microcontroller interfaces with these motor drivers to control the direction the motors spin, the speed at which the motors move, and checks for possible faults. The speed and direction provided to these motors depends on what

shape is being drawn. For instance, a right triangle with two 45° angles will require the x and y motor to be driven at the same speed in order to draw the hypotenuse. However, for any arbitrary triangle, each motor would need to be run at a different speed to ensure that they both reach the correct destination at the right time, while also following the correct drawing path. The mathematics behind such motor movements will be discussed further in the 'Microcontroller Firmware' section. **(SI & VR)**

(2.3) Signal Processing

For this project, all of the necessary signal processing is handled via firmware pre-installed on the microcontroller. The firmware is responsible for reading any incoming user input signals (either from the on-board controls or the smartphone application via Bluetooth), then translating that signal into the intended action. These signals take the form of packets that the firmware can parse and read to perform the desired action. A list of these packet formats can be seen in Table 1. **(JM)**

Table 1: Formats of Available Packet Types

Packet Type	Packet Format
Draw	<d#:##:##:##:##>
Text	<t:####>
Free Move	<m#>
Servo Control	<s>
Return Home	<h#>
Calibrate	<c>

After receiving a 'Draw' or 'Text' packet, the firmware will translate the data received from these signals into the appropriate *x* and *y* motor movements. The firmware is then responsible for using this translated data to send the appropriate signals to the *x* and *y* stepper motors, and the servo motor, in order to draw the desired shape or pattern. Also, it should be noted that the amount of data in the user input signal is not constant, as it depends on which of the preset shapes the user would like to draw. Table 2 lists all of the available shape presets, as well as the parameters each

of them require the user to specify. As indicated by this table, the smallest possible signal the firmware can receive comes from the user choosing to draw a circle, as this only requires one parameter (the radius) to be specified. On the other hand, the largest signal comes from the sinusoid preset, as the user must specify four parameters (amplitude, phase shift, wavelength, and wave count) for the shape to be drawn properly. All other shape presets only require two parameters, meaning their signals will send (roughly) the same amount of data. In addition to these parameters, each shape (except for sinusoids and circles) can also be given a rotation parameter, to rotate the shape around its starting point. **(VR & JM)**

Table 2: List of Required Inputs for each Available Shape Preset

Shape	**Required Parameters**
Line	Length (cm or in), Angle (degrees)
Rectangle	Length (cm or in), Height (cm or in)
Triangle	Base (cm or in), Height (cm or in)
Polygon	Number of Sides (integer), Side Length (cm or in)
Arc	Radius (cm or in), Arc Angle (degrees)
Circle	Radius (cm or in)
Sinusoid	Amplitude (cm or in), Phase (degrees), Wavelength (cm or in), Number of Waves (integer)

In addition to shapes, a select amount of ASCII characters can also be drawn. These include uppercase letters A through Z, along with a few other punctuation symbols, such as commas and periods. As stated, the text packet format can be seen in Table 1, where the hashtags represent the characters to be drawn. Since text drawing functionality was included near the end of the project, said functionality was only implemented within the smartphone application and is not available from the onboard controls. **(JM)**

‘Free Move’ packets have five possible states - move up, move down, move left, move right, and no motion. These packets are shown in Table 3.

Table 3: Free Move Packet Types

Free Move State	Packet Format
Off	`<m0>`
Up	`<m1>`
Right	`<m2>`
Down	`<m3>`
Left	`<m4>`

‘Return Home’ packets have seven possible states, as seen in Table 4. These include packets to set a custom home position, return to a custom home position, and return to one of five predetermined home positions.

Table 4: Return Home Packet Types

Return Home State	Packet Format
Set Custom	`<h0>`
Return Top Left	`<h1>`
Return Top Right	`<h2>`
Return Bottom Left	`<h3>`
Return Bottom Right	`<h4>`
Return Center	`<h5>`
Return Custom	`<h6>`

Lastly, there is a packet to engage/disengage the servo (whose state is tracked in the firmware), along with a calibration packet to recalibrate the device’s positioning. **(JM)**

(2.4) Communications

The Liquid Crystal Display and buttons used by the on-board controls are located on a separate solder board that is wired to the PIC24FJ128GA010 microcontroller. For the smartphone application, the RN4870 Bluetooth module was used as an external device that was connected to the microcontroller using a host interface of Transparent UART serial connection, and a chosen baud rate of 115200. The logic inputs to the motor driver are based on the 5 volts being powered to the pins (as discussed in Section 2.1). The other pins that take direct information from the PIC board include the frequency or oscillator and the direction pin. **(DA & SI)**

(2.5) Electromechanics

The electromechanics of this project make use of three stepper motors and one servo motor. As stated, two stepper motors operated on the *x*-axis, and the other stepper motor operated on the *y*-axis. The servo motor then engages and disengages the marker from the whiteboard.

(2.5.1) Stepper Motors

The stepper motors used for this project are four-wire, bipolar, NEMA17 motors. There are many different variants of stepper motors, but for this project, using a stepper motor similar to that of a 3D printer stepper motor was the best option, as they allow for very precise positioning control when provided the appropriate signals. **(DA & SI)**

A stepper motor works by having coils energized in certain patterns. As in the bipolar case, there are 4 total coils, where two are energized at a time or all four. The pairs are opposite ends of each other and this forces the rotor to align itself with the pairs. Once aligned, all the coils are turned on to move the rotor again, followed by the other pair of coils being energized. This process is repeated and it is this process that results in the motor shaft spinning. Figure 1 provides a visualization of this. [9] **(DA & SI)**

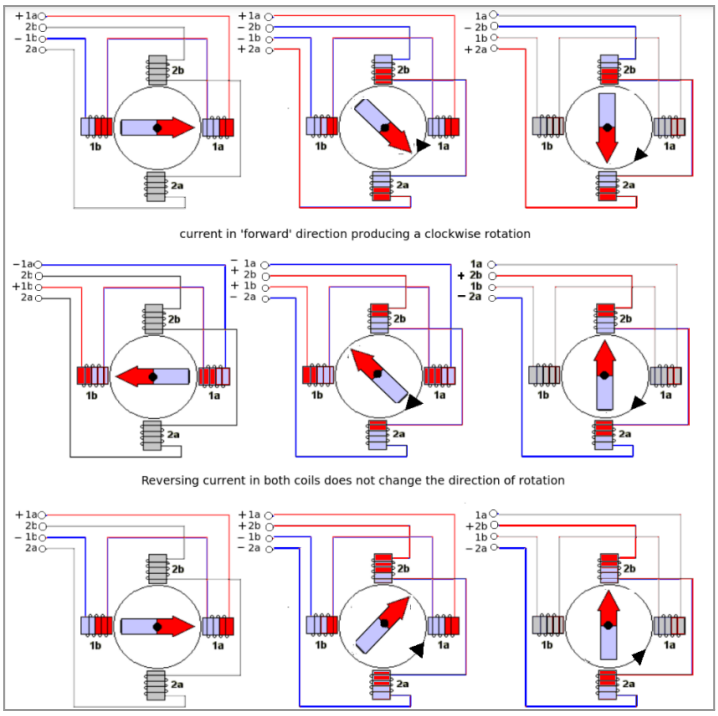

Figure 1: Visual Representation of How a Stepper Motor Operates

Using a stepper motor with a rated voltage of 3.6 V_{DC}, the typical torque is roughly 42 $N \cdot cm$ (60 $oz \cdot in$). The average coefficient of friction for a marker being $\mu_s \approx 0.4$, the average weight of a marker is 0.13344 N, and the average radius of a marker is 0.00889 meters. With that information, the following is derived:

$$\begin{aligned} \text{Torque} &= \mu_s \times w_{marker} \times r_{marker} \\ &= (0.4) \times (0.13344\ \text{N}) \times (0.00889\ \text{m}) \\ &= 0.000475\ \text{N} \cdot \text{m} \\ &= 0.0475\ \text{N} \cdot \text{cm} \end{aligned}$$

where the stepper motors need to overcome the 0.0475 $N \cdot cm$ torque acting against it. Referring to Section 2.1, the stepper motors will be more than capable of doing the task. **(DA & SI)**

(2.5.2) Servo Motor

The servo motor is responsible for engaging and disengaging the marker from the board. This is achieved by using a fixed angle that elevates the marker to and from the whiteboard. This angle is communicated from the microcontroller via a current sent to the motor. Said current produces torque and allows the motor's gears to spin. Once the servo motor's axle is set at the desired angle, the position is held until the gears are turned the opposite direction which would lift the marker from the board. [10] **(DA)**

The positional servo motor being used for this project has an operating voltage of 4.8V to 6V (range of output). This was determined after concluding that the average servo motor, with this voltage range, has a stall torque of 1.8 kg/cm. This means that, with the wheel or attachment radius being 1 cm, the servo motor can move 1.8 kg. Since the average marker weighs 0.0136 kg, the servo motor chosen for this project has more than enough power to reliably engage and disengage the dry erase marker. The other determining factor was the dry erase marker housing weight. Because it weighs under 1.7 kg, the servo motor is sufficient for handling such a load. **(SI)**

The servo is controlled by a PWM (pulse width modulation) signal where the duty cycle, how long the signal is high, determines the position of the servo. The period of the signal is twenty milliseconds, and fifty of these periods are used to position the servo. A pulse width of 500 ms is used to engage the servo, while a pulse width of 1350 ms is used to disengage the servo. **(JM)**

(2.6) Computer Networks

Bluetooth is used to communicate between the smartphone application and the microcontroller. For our final design, the RN4870 Low Energy Bluetooth module is used since it is compatible with the PIC24FJ128GA010 microcontroller. Per the *RN4870/71 Bluetooth Low Energy Module* data sheet, the range of the RN4870 is up to 50m ~ 164 ft [12]. This range is well within the size of most American classrooms, which is where our project is intended to be used. The RN4870 has a maximum data rate of 10 kbps [12]. Assuming that a string is sent from the smartphone

application to the Bluetooth module, the time that it takes for the microcontroller to receive the data can be calculated with:

$$Delay = \frac{(\#\ of\ bytes) \times\ 8}{maximum\ data\ rate}$$

So, if we were to send an example string from the smartphone application to the RN4870 Bluetooth module that represents a square with both length and height of five centimeters, the string would take the following form:

```
<d1:5:5:0>
```

Since each of the ten string characters are represented using one byte, the response time of the drawing apparatus can be calculated with:

$$Delay = \frac{10\ bytes \times 8}{10 \times 10^{3}\frac{bytes}{sec}}$$

$$= 8.0\ \text{ms}$$

When connecting two Bluetooth Low Energy devices, one device must act as the central role while the other acts as the peripheral role. The peripheral device will advertise that it can be connected to, while the central device scans for connectable devices and initiates a connection [13]. In our case, the RN4870 will be the peripheral device and the smartphone will be the central device. **(JM)**

(2.7) Embedded Systems

The implementation of this device requires two particular embedded systems. First, the system will use a microcontroller (in our case, the PIC24FJ128GA010 for our prototype) as a bridge between the user's input interface and the motors that make up the drawing apparatus. Second, the on-board control module will serve as one of the two methods the user can choose to use when interfacing with the device. **(VR)**

(2.7.1) Microcontroller

As stated in Section 2.3, the microcontroller is responsible for reading user input signals, translating these signals into the appropriate motor commands, and sending the translated commands to the stepper motors and servo motor. The microcontroller used for this project operates at voltages between 2.0V and 3.6V, and allows for clock speeds up to 32 MHz (via an 8 MHz oscillator with the option of 4×PLL). The firmware for the microcontroller was written in C, and installed onto the board via a USB connection. The microcontroller also provides a port that can be used to attach our bluetooth module for receiving input from the smartphone application. Lastly, while the microcontroller does provide a built-in liquid crystal display (LCD) that we were able to use for the on-board controls of our prototype, the final version of the device utilizes an external LCD instead. **(VR)**

(2.7.2) On-Board Controls

The on-board controls provide an alternative input method if the user is not able to use the smartphone application to interface with the device. The on-board controls utilize a 16 × 2 external LCD (with ASCII characters of dimensions 5 × 7) to provide a basic visual interface to the user. The first line of the LCD presents a menu title, and the lower line presents the user's current selection. To interface with the LCD, the on-board controls provide the user with 4 arrow buttons (with one arrow pointing in each cardinal direction), and a 5th button in the center of the arrows, normally used for confirming a selection. A 6th button also allows the user to traverse backwards through the menus if they want to change any previously entered selections. Lastly, a 7th button known as the 'macro' button can be programmed by the user in the on-board settings menu. The user can tell this button to return the marker to their specified home position, calibrate the machine, or even draw one of the shape presets with predefined parameters, depending on what the user decides.

Additionally, when selecting the position on the whiteboard where the shape will be drawn, the user will be able to freely move the marker to a starting point using the four arrow buttons. The confirmation button can also be pressed to engage / disengage the marker and free draw straight

lines, but when the device is told to draw a shape, it will make sure to press the marker on the whiteboard before drawing. Figure 2 gives a visualization of the format used when presenting the menu to the user on the LCD, and Figure 3 provides a visualization of the button layout that is used to navigate the on-board menus While the menu prompts suggest that the left and right arrows are used to navigate the menu, with the idea of dependability via redundancy in mind, the up and down arrows can be used as well. **(VR)**

Figure 2: Visualization of LCD Screen Display Format for On-Board Controls

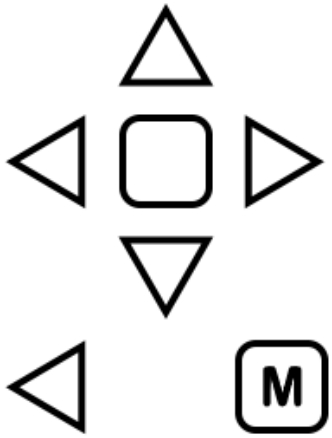

Figure 3: Button Layout for On-Board Controls

(2.8) Controls

User input can be read from an Android smartphone application or pre-programmed onboard controls to achieve the desired output (line, rectangle, polygon, triangle, sinusoid, arc, circle). The smartphone application communicates to the PIC24FJ128GA010 microcontroller through a Bluetooth module while the on-board controls are directly connected to the microcontroller. The microcontroller can give the stepper motors commands to move the marker in the x-direction, y-direction, and toward / away from the whiteboard simultaneously. **(DA & SI)**

In order to implement closed-loop control, the frame of the apparatus contains four limit switches - one on each side. These limit switches communicate to the microcontroller so that, if the user enters a shape that will not fit based on the starting point they want, they will need to re-enter the desired size. The limit switches will act as a barrier so that the microcontroller knows how many revolutions it would take until the end of the apparatus frame reaches its limits, and the amount of revolutions will reflect the size (in cm). While the microcontroller firmware should be able to keep track of the available board space and the current location of the marker, these limit switches act as a failsafe measure to prevent any mishaps that could occur (drawing off the board or beyond the limits of the apparatus). **(DA & SI)**

These limit switches are also used to implement the calibration functionality of the device. During initial calibration, the *x* motor will continuously move to the left until a limit switch rising edge is detected, and then move up until another limit switch rising edge is detected. The firmware will then consider this location as the origin of the board, assigning it the absolute coordinates (0,0) in the firmware. Then, while the device is being used, the software will attempt to keep track of the device's location by keeping track of how many steps each motor has moved along both the *x* and *y* axes. However, after extended use, the accuracy of these numbers may suffer, which is why the user can choose to manually calibrate the machine at any given time. For manual calibration, the machine will use its current *x* and *y* coordinates to predict which of the four corners of the board it is currently closest to, and will then move towards that corner using the limit switches to detect the bounds **(VR)**.

ENGINEERING REQUIREMENTS SPECIFICATION (JM & VR & DA & SI)

Table 5: Engineering Requirement Specification

Marketing Requirement	Engineering Requirement	Justification
2	The device should allow for markers of multiple brands / sizes to be used (at least 3”)	Improves accessibility and the user experience (don’t need one specific type of marker)
1, 2	The device should be able to draw a maximum shape of (3’ x 3’)	Allows for maximum use space while still providing space for the device housing to attach to.
4	The user should be able to specify the start point, choose a shape, and define shape parameters via on-board controls.	Accessibility for users without access to a smartphone
5	The user should be able to specify the start point, choose a shape, and define shape parameters remotely.	Accessibility for users that may have trouble physically interacting with the device
1	The device should be able to retrofit on to a standard whiteboard (4’ x 4’)	Allows the device to be used in most standard classrooms
3	Users should be able to choose between 5 different shape presets (line, rectangle, triangle, circle, sinusoid)	Allows users to draw multiple of the most common geometric shapes and patterns
3	Each shape should have its own set of customizable parameters ($base_3$, $length_{1,2}$, $height_{2,3}$, $orientation_1$, $wavelength_4$, $period\ count_4$, $amplitude_4$, $radius_5$)	Having fixed dimensions for each shape would not suit every scenario for the user (1 = Line, 2 = Rectangle, 3 = Triangle, 4 = Sinusoid, 5 = Circle)
5	Wireless range of remote device control should cover the typical size of a classroom (23 ft^2)	Allows the device to be controlled from anywhere in the room (not just up close)
2, 4, 5	The device should be able to move and draw with a marker at a speed of 1 ft/s	Allows for a drawing speed that is not too slow, and doesn’t raise safety concerns by moving too fast
2, 3	The device should be able to reject a user-specified shape if its size and position exceeds the whiteboard bounds	Ensures the machine doesn’t try to move outside of its range, reducing potential stress on the stepper motors.

Marketing Requirements:

1. The system should be able to retrofit to any whiteboard
2. The system should be able to draw with a marker
3. The system should be able to vary the size and position of the user defined shape
4. The system should have a built in user interface for manual control
5. The system should also be controlled via an external device

ENGINEERING STANDARDS SPECIFICATION

Table 6 provides a list of the standards that will be in use for this project across different categories. The table also provides which aspects of the project will use the specified standards.

Table 6: Engineering Standards Specification

	Standard	Use
Communications	Bluetooth USB UART	App / PIC24 communication Microcontroller firmware installation Bluetooth module / PIC24 communication
Design Methods	Flow Chart Finite State Machine Pseudocode	Smartphone application On-board controls Microcontroller Firmware Motor movement apparatus
Programming Languages	Kotlin C	Smartphone application development Firmware development
Connector Standards	Pitch Connector	Connecting microcontroller to motors

ACCEPTED TECHNICAL DESIGN

Figure 4 shows the Level 0 block diagram for this project, and Table 7 lists the functional requirements of the single module that makes up this diagram. In order for a shape to be drawn, the device requires power (120 V_{AC} supplied by a wall outlet), a dry erase marker to draw with, and a user input command specifying the shape and size, as shown in Figure 4.

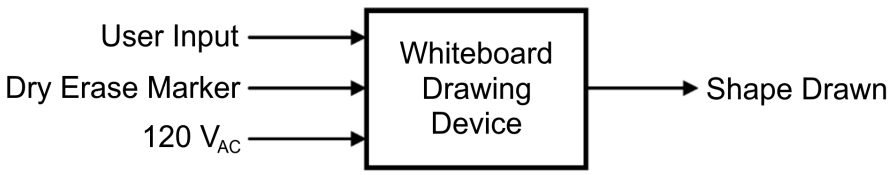

Figure 4: Level 0 Block Diagram

Table 7: Functional Requirement Table for Whiteboard Drawing Device

Module	Whiteboard Geometry Attachment
Designer(s)	Drew Adams, Stefan Ilic, James Medved, Vaughn Richards
Input(s)	User Input: Via on-board controls or smartphone application Dry Erase Marker: Minimum length of 3” Power: $120V_{AC}$
Output(s)	Shape Drawn: With specified position and size
Description	Draws the desired shape on the whiteboard based on user input. The shape, size, and position of the final drawing should have variable user control.

Figure 5 shows a Level 1 block diagram of the device, breaking the module down into the two different modules that handle user input, and the module containing the hardware that draws the desired shape or pattern to the whiteboard. Tables 8, 9, and 10 list the functional requirements for the smartphone application, on-board control, and drawing hardware modules, respectively.

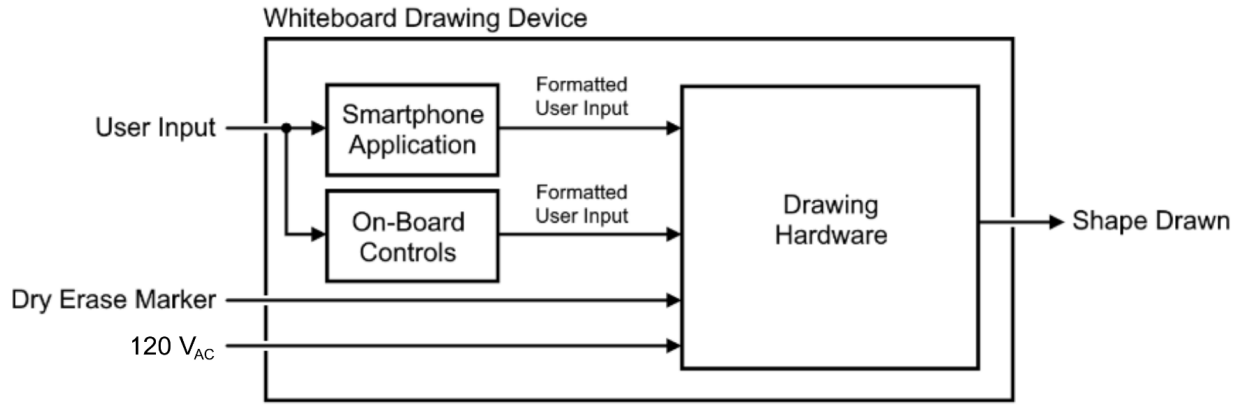

Figure 5: Level 1 Block Diagram

Table 8: Functional Requirement Table for Smartphone Application

Module	Smartphone Application
Designer(s)	James Medved
Input(s)	User Input (Shape and Dimensions via text boxes)
Output(s)	Formatted User Input (in cm)
Description	The Smartphone application allows the user to pick a shape and enter its dimensions. Then that data will be sent to the Bluetooth module.

Table 9: Functional Requirement Table for On-Board Controls

Module	On-Board Controls
Designer(s)	Vaughn Richards
Input(s)	$5V_{DC}$ User Input (via up, down, left, right, and confirm buttons for menu)
Output(s)	Formatted User Input (in cm)
Description	The LCD screen will allow for the user to choose a shape and specify the size and orientation parameters of said shape. This will be an alternative to the smartphone application for more practical use.

Table 10: Functional Requirement Table for Drawing Hardware

Module	Drawing Hardware
Designer(s)	Drew Adams, Stefan Ilic
Input(s)	$12\text{-}24V_{DC}$
Output(s)	Shape Drawn
Description	Draws the desired shape on the whiteboard based on user input. The shape, size, and position of the final drawing should have variable user control.

(5.1) Hardware Design

The stepper motors are the driving force for the marker to be moved in the x and y directions. Stepper motors are very commonly used in the 3D printing world, so they can also be used as a 2D printer for this application. They have many advantages over standard DC motors such as being easily controlled by a computer, allowing for precise rotation, and having high torque at low speeds. A stepper motor is defined as a "brushless, synchronous, electric motor" where it has multiple coils organized into groups called "phases." It rotates one step at a time (hence the name) by converting digital pulses to energize each phase in a given sequence. They are basically the same as DC motors, but they move in discrete steps dictated by mechanical shaft rotations, where they are sent a separate pulse for each step they take. **(DA & SI)**

Each step they take is roughly the same size. These pulses allow the motor to move at a precise angle (1.8° for the stepper motor being used). This allows for the motor's position to be controlled without the need for a feedback mechanism. This is why stepper motors are used in applications that require high precision [12]. As far as engaging and disengaging the marker to the board, a small servo motor seemed most suitable for this application. It requires a low current draw (2A) at a rated voltage of 4.8-6V. The stall torque of the servo is 1.8 kg/cm, which is enough to lift the expo marker and overcome friction when writing. A PIC24FJ128GA010 microcontroller gives commands to the stepper motors and the servo motor. The on-board controls are directly connected to the microcontroller, while the smartphone application interfaces with the Bluetooth module before communicating to the microcontroller. **(DA & SI)**

For further reference, Figure 6 provides the Level 2 block diagram of the drawing hardware module. In this figure, the Marker X Motion and Marker Y Motion modules use stepper motors to guide the motion of the marker, and the Marker Engage / Disengage module uses a servo motor to press and lift the marker on and off of the whiteboard. Tables 11 through 15 provide the functional requirements for the bluetooth, microcontroller, marker *x* motion, marker y motion, and marker engage / disengage modules, respectively. **(VR)**

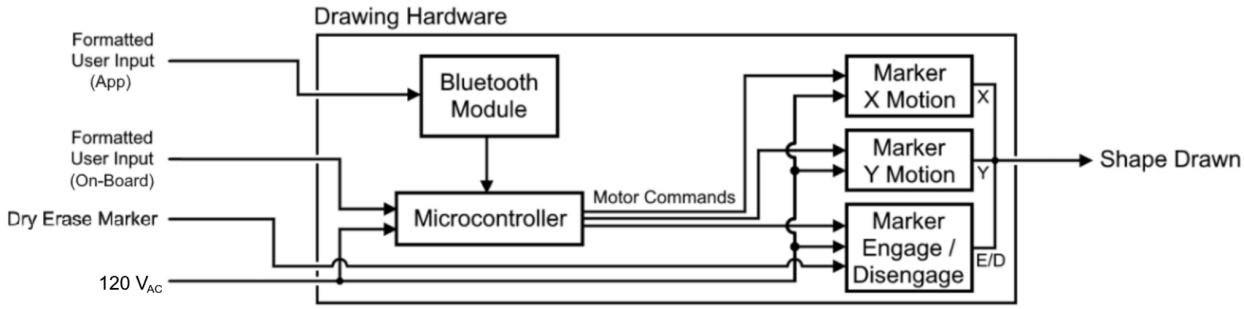

Figure 6: Level 2 Block Diagram for Drawing Hardware

Table 11: Functional Requirement Table for Bluetooth Module

Module	Bluetooth Module
Designer(s)	James Medved
Input(s)	Formatted User Input (received via Smartphone app)
Output(s)	Formatted User Input (sent to microcontroller)
Description	Acts as a bridge between the smartphone application and the microcontroller to allow for user inputs to be read from a smartphone.

Table 12: Functional Requirement Table for Microcontroller

Module	Microcontroller
Designer(s)	Vaughn Richards
Input(s)	Formatted User Input (received via Smartphone app or on-board controls)
Output(s)	Motor Commands (for x-axis stepper, y-axis stepper, and servo motor)
Description	Translates incoming user input into the motor movements required to draw the desired shape with the user-specified parameters.

Table 13: Functional Requirement Table for Marker X Motion

Module	Marker X Motion
Designer(s)	Drew Adams, Stefan Ilic
Input(s)	Motor command (sent from the microcontroller) Power: $12V_{DC}$
Output(s)	Marker movement along the x-axis (via stepper motor)
Description	Moves the dry erase marker along the *x*-axis by rotating the number of steps specified in the input command.

Table 14: Functional Requirement Table for Marker Y Motion

Module	Marker Y Motion
Designer(s)	Drew Adams, Stefan Ilic
Input(s)	Motor command (sent from the microcontroller) Power: $12V_{DC}$
Output(s)	Marker movement along the y-axis (via stepper motor)
Description	Moves the dry erase marker along the *y*-axis by rotating the number of steps specified in the input command.

Table 15: Functional Requirement Table for Marker Engage / Disengage

Module	Marker Engage / Disengage
Designer(s)	Drew Adams, Stefan Ilic
Input(s)	Motor command (sent from the microcontroller) Power: 4 - 6 V_{DC}
Output(s)	Marker is pressed on / lifted off of the board (depending on the command)
Description	Presses the marker against the whiteboard when a shape is being drawn, and lifts the marker away from the whiteboard otherwise.

Figure 7 shows the circuit schematic used for the Whiteboard Drawing Device. The DRV8825 motor driver is responsible for taking the desired input from the user (via on-board controls or Bluetooth) and giving the commands to the stepper motors. The RN4870 Bluetooth module is connected to the PIC24FJ128GA010 board, and the duty of the Bluetooth module is to forward the commands that are inputted in an Android app. **(SI & VR)**

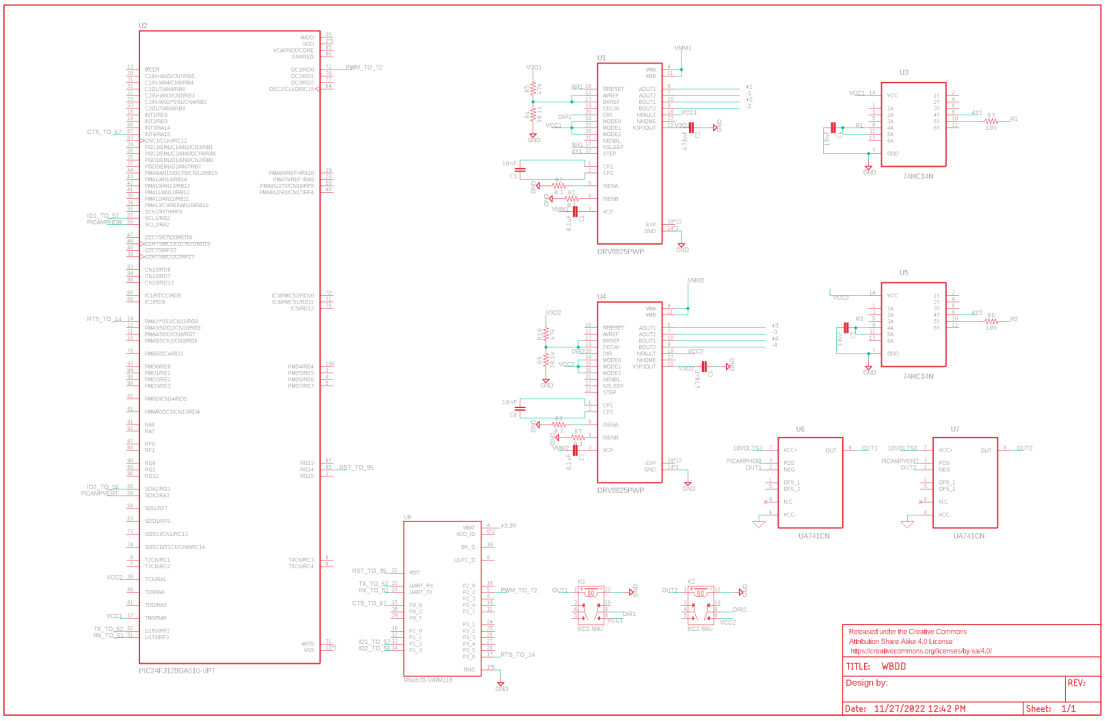

Figure 7: Circuit Schematics

(5.1.1) Limit Switches

The four limit switches placed at the top, bottom, left, and right sides of the board serve multiple purposes. To track the position of the drawing apparatus, the device needs to be initially calibrated. The limit switches allow the device to move until a switch is hit, revealing the position of the device. Another purpose of the limit switches is to avoid damage if the device is calibrated incorrectly. The limit switches disable the motors, preventing the device from moving out of bounds. **(JM)**

Each limit switch requires 5V input, and output an active low signal. This means that each switch will be supplying voltage until it is triggered, stopping the voltage output. On the PIC24, the max voltage input without damage to a pin is ~3.6v. So, a PNP transistor was used to allow a 3.3v source to power the rest of the circuit, rather than the 5v output from the switches. PNP transistors allow current flow when the base is low, so when a switch is hit, current will flow from the collector to the emitter. Each limit switch is connected to one of these transistors and the emitter of each is tapped off to a different pin on PORTA. Tapping off these pins allows the microcontroller to know what specific switch was hit. Lastly, all of the transistor outputs are fed through diodes to avoid the backflow of current, and sent to the microcontroller's interrupt pin. Triggering any of the limit switches would thereby trigger the interrupt, disabling the motors and bouncing off in the correct direction. **(JM)**

(5.1.2) **Onboard Control Hardware**

Implementation of the on-board controls required interfacing with seven tactile buttons used to implement the button layout shown in Figure 3. While each button could have easily been tied to its own pin on the microcontroller, this would require the use of seven total pins, and because the on-board controls use an LCD screen with an 8-bit data line, a good amount of our pins are already used by the screen. Thus, each of the buttons were hardware-encoded using diodes to a specific 3-bit pattern that the firmware can then decode into the appropriate button press. **(VR)**

Table 16 provides the bit patterns assigned to each button. These bit patterns were assigned in such a way that reading only the most significant bit of the pattern is enough to know whether or not a directional arrow is pressed. Figure 8 provides a circuit schematic of the diode circuit used to hardware encode said bit patterns. It should be noted that while the circuit uses a total of 12 diodes, 3 of these diodes are technically unnecessary. Specifically, the diodes used for the confirm, back, and down buttons are not necessary because each of these buttons only drive one of the three bit lines high. Lastly, to prevent ghosting caused by multiple buttons being pressed at once, the software is programmed to read only one button at a time, and will ignore any buttons that are pressed down while another button is already pressed. **(VR)**

Table 16: Hardware Encoded Bit Patterns for On-Board Buttons

Button	b_2	b_1	b_0
None	0	0	0
Confirm	0	0	1
Back	0	1	0
Macro	0	1	1
Down	1	0	0
Up	1	0	1
Left	1	1	0
Right	1	1	1

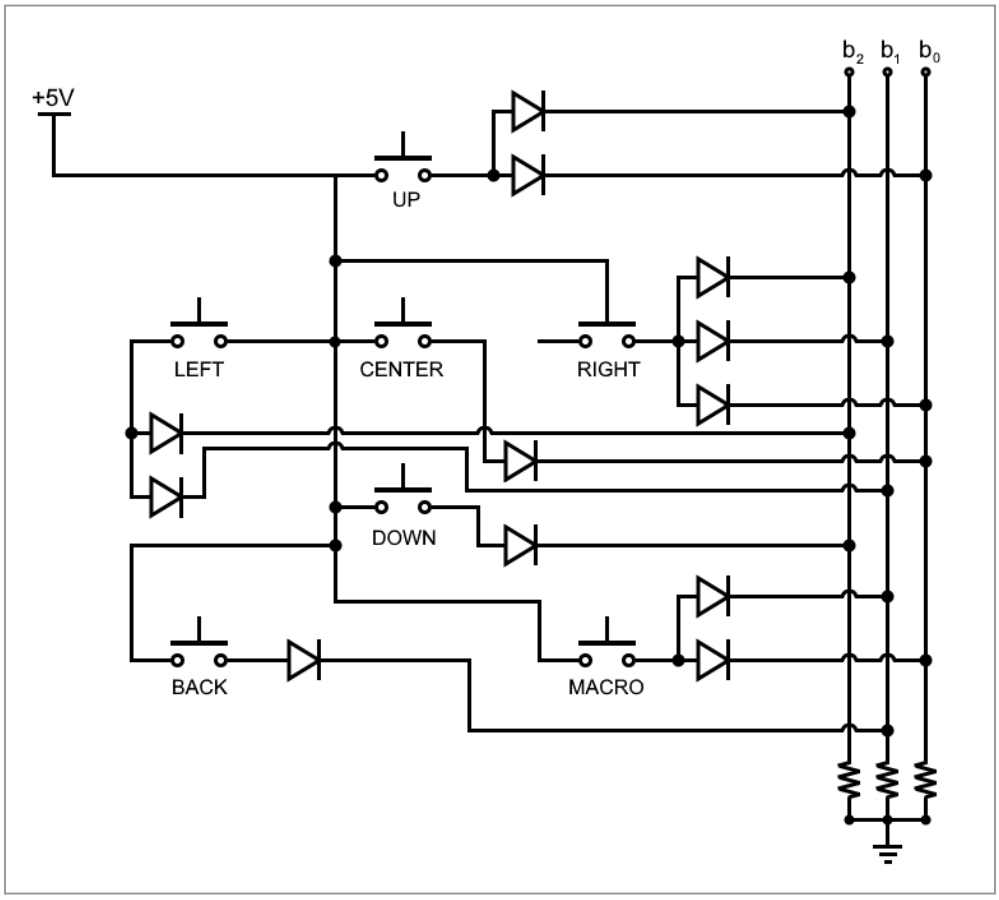

Figure 8: Circuit Schematic for the On-Board Control Buttons

(5.2) Software Design

(5.2.1) Smartphone Application

A smartphone application is used to control the drawing apparatus remotely. This application was programmed for the Android operating system using Kotlin, a statically typed programming language which was designed using the Java Class Library and JVM (Java Virtual Machine). We chose to program our smartphone application in Kotlin for a few reasons. First of all, Android software is open-source, allowing us to program Android applications using Kotlin on any machine. Also, Google has been in support of Kotlin since 2017, and actively pushing it on new developers since 2019. **(JM)**

To draw a shape using the smartphone application, the user will first need to connect to the Bluetooth module of the drawing apparatus. Once connected, they can position the drawing apparatus (using either the on-board controls or the smartphone application), select a shape, enter its dimensions, and send the shape information to the Bluetooth module. Figure 9 depicts this process in the form of a flow chart. **(JM)**

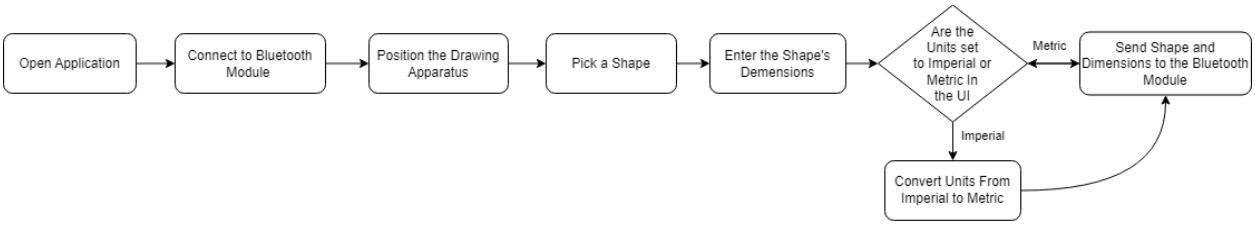

Figure 9: Smartphone Usage Flowchart

The *onCreate* method of the *MainActivity* class is the first activity (app page) loaded when the smartphone application is started. This activity can be seen in Figure 10.

```
class MainActivity : AppCompatActivity() {
   private var serviceConnection : BluetoothServiceConnection = BluetoothServiceConnection(this)

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContentView(R.layout.activity_connect)

       // Connect to RN4870 GATT Server
       btn_connect.setOnClickListener{
           try {
               applicationContext.bindService(Intent(this, BluetoothLeService::class.java),
serviceConnection, Context.BIND_AUTO_CREATE)

               thread {
                   while (serviceConnection.btLeService?.connected != true) {
                       Thread.sleep(1000)
                   }
                   startActivity(Intent(this, HomeActivity::class.java))
               }
           }
           catch (e : Exception) {
               Log.e("BluetoothGatt", "Device was unable to connect to RN4870 GATT server")
               Toast.makeText(this, "Device was unable to connect to RN4870 GATT server",
Toast.LENGTH_LONG).show()
           }}}}
```

Figure 10: Kotlin Implementation of the *MainActivity OnCreate* Method

The main activity calls the serviceConnection class. This class initializes the Bluetooth LE service, along with creating needed Bluetooth adapter, manager, and device variables used in the connection process. Also, from the serviceConnection class, the client (smartphone) is connected to the RN4870 module's GATT server. This is done by instantiating a Bluetooth Gatt callback, found in Figure 11, which can be used to discover the desired Bluetooth GATT service. For the RN4870 Low Energy Bluetooth module, this service is:

49535343-FE7D-4AE5-8FA9-9FAFD205E455

Bluetooth services have characteristics that work similar to passwords, allowing one device to write to another. In our case, the write characteristic is:

49535343-1E4D-4BD9-BA61-23C647249616

And the read characteristic is:

49535343-4C8A-39B3-2F49-511CFF073B7E

```
private val btGattCallback = object : BluetoothGattCallback() {
   @SuppressLint("MissingPermission")
   override fun onConnectionStateChange(gatt: BluetoothGatt, status: Int, newState: Int) {
       val deviceAddress = gatt.device.address

       if (status == BluetoothGatt.GATT_SUCCESS) {
           if (newState == BluetoothProfile.STATE_CONNECTED) {
               toastMessage("Successfully connected to $deviceAddress")
               btGatt!!.discoverServices()
               connected = true
               Log.i("BluetoothGattCallback", "Successfully connected to $deviceAddress")
               toastMessage("Successfully connected to $deviceAddress")

           } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
               Log.i("BluetoothGattCallback", "Successfully disconnected from $deviceAddress")
               toastMessage("Successfully disconnected from $deviceAddress")
               btGatt?.close()
               connected = false
           }

       } else {
           Log.e("BluetoothGattCallback", "Error $status encountered for $deviceAddress! Disconnecting...")
           toastMessage("Error $status encountered for $deviceAddress! Disconnecting...")
           btGatt?.close()
           connected = false
       }
    }
```

Figure 11: Kotlin Implementation of the Bluetooth Gatt Callback

Once the characteristics were found, and their descriptor values were updated, we were able to read and write to the RN4870 module. Reading from the module could be accomplished by checking the read characteristic's value, and converting that value to a string. Similarly, writing to the module could be accomplished by first converting the desired string to a byte array, then setting the write characteristic's value to that byte array. **(JM)**

Clicking the 'Connect' button from the main (connect) activity will check to see if the user is connected to the RN4870 module's GATT server. If the user is not connected, it will attempt connection, then move to the next app page once the connection process has completed. Similar events occur whenever Bluetooth packets are sent to the microcontroller where, if the phone has been disconnected from the RN4870 module's GATT server, it will reconnect before continuing. This ensures that the smartphone application will stay connected to the microcontroller without the need of constant reconnecting from the user. **(JM)**

After a user has connected to the Bluetooth module, they can then choose a shape from within the smartphone application. After the user leaves the connect screen, they will see three tabs: a presets tab used to draw shapes and text, a free move tab used to position the device, and a settings tab where they can set their in-app units and home position. **(JM)**

Users are able to choose their units (Imperial or Metric) from the settings tab. Before a shape and its dimensions are sent to the Bluetooth module, if the application's units are set to Imperial, the shape's dimensions are converted to Metric. This is done to ensure that the firmware is always making calculations with the same type of units. The Imperial to Metric conversion can be calculated with:

$$Centimeters = 2.45 \times Inches$$

Since all shape dimensions are sent as integers, using Imperial units in the app will lead to slightly inaccurate drawings, as they are simply rounded to the nearest centimeter. **(JM)**

The *Presets* tab has buttons for each of the possible objects that can be drawn: line, rectangle, triangle, polygon, arc, circle, sinusoid, and text. When one of these buttons is clicked, the user will be prompted with a screen to enter the dimensions for the shape (which will be different for all shapes), or the desired text if chosen. Lastly, the user can click draw to initiate the drawing. If the shape with its current dimensions is within the bounds of the whiteboard, the shape and its dimensions will be sent to the microcontroller over Bluetooth. If the shape is not in bounds, an error will be displayed to the user. A flow chart representation of this shape selection process is shown in Figure 12. **(JM)**

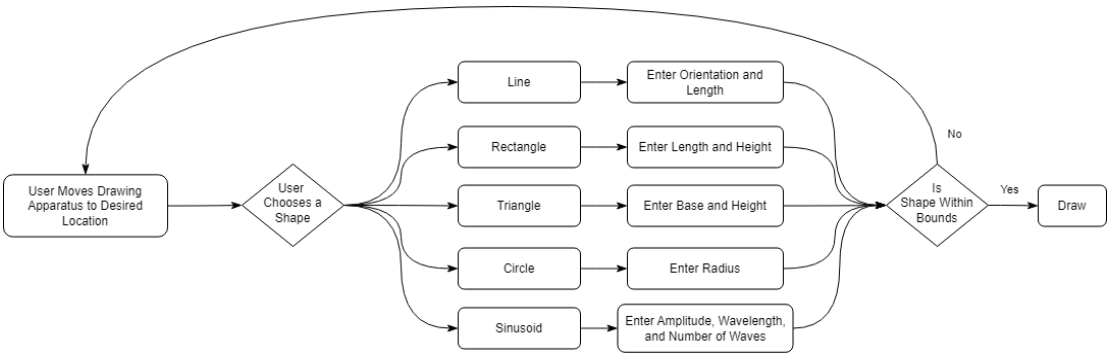

Figure 12: Shape Selection Flow Chart

The *Move* tab contains buttons for different movement options, including: a D-pad to move the device in each of the eight cardinal directions, a *draw* button to engage the servo, a *home* button to return the device to its currently set home position, a *set home* button to set a custom home position, and a *calibrate* button to calibrate the device's known position. **(JM)**

The *Settings* tab contains two drop-downs. The first drop-down allows for the unit selection of Metric or Imperial. The second drop-down allows the user to set their home position. These home positions include the top left, top right, bottom left, bottom right, or center of the whiteboard. There is also an option of a custom home position, set by the user from the *Move* tab. **(JM)**

(5.2.2) On-Board Controls

Since the on-board controls are intended to provide the same functionality as the app, its implementation follows a very similar process with a few key differences. For example, the application allows the user to input numerical values by typing them into a number pad, but the on-board controls don't provide enough buttons to implement a similar approach. To circumvent this, the on-board controls set a default value to the parameters of every shape preset to a value that, on average, is relatively close to the most common input value for that shape. From here, the user can use the arrow buttons to increase (up arrow or right arrow) or decrease (down arrow or left arrow) the magnitude of the input number by increments of 1. Another limitation of the on-board controls is the lack of support for writing text, as this remains an exclusive feature of the smartphone application. **(VR)**

To begin the on-board implementation, firmware code had to be written that could print text to the LCD screen. The particular model of LCD used for this project consists of 8 data pins, a register select pin, a read/write select pin, and an enable pin (along with power and ground, contrast, and backlight cathode and anode). To make the interfacing process easier to read in code form, microcontroller pins on the board were given appropriate names that could be used in all hardware interfacing functions. Figure 13 provides the small snippet of code that provides names for the pins used for LCD interfacing. **(VR)**

```
#define RS PORTGbits.RG12           // LCD Register Select Pin (IR/DR)
#define RW PORTGbits.RG13           // LCD Read/Write Pin
#define EN PORTGbits.RG14           // LCD Chip Enable Pin
#define LCD_DATA PORTE              // LCD Data Pins (Only use PORTE<7..0>)
```

Figure 13: Pin configuration for LCD

Additionally, according to the LCD's datasheet, when sending data to the LCD's data or instruction register, the 8 data lines (as well as the RS and R/W pins) are first set to their desired values, and then the enable (EN) pin can be quickly turned on and off to allow the LCD to read in said data. For this, a simple function (provided in Figure 14) was created that pulls the EN pin high for 1ms. **(VR)**

```
// Pulses the LCD ENABLE pin (RF8)
void LCD_flash_EN()
{
    EN = 1;
    ms_delay(1);   // 1ms > 140ns
    EN = 0;
}
```

Figure 14: Function Used to Flash LCD Enable Pin

Note that the code in Figure 14 uses a function called *ms_delay* to halt the program for one millisecond. To implement this function, we can use one of the timers provided by the PIC24 board. By enabling a timer with a 256 pre-scale on the 16MHz clock used by the board, over the course of t seconds, the timer will have reached some value v such that

$$v = \frac{t}{\frac{1}{1.6\times10^{7}} \times 256} \Rightarrow v = 62500t$$

However, for the purposes of this project, only delays on the order of milliseconds are needed. In this case, by interpreting the value of t as milliseconds, the timer value v that coincides with some arbitrary value of t is

$$v = 62500(\frac{t}{1000}) \Rightarrow v = 62.5t$$

Thus, assuming TMR1 is configured with a prescale of 256, meaning

```
T1CON = 0x8030;
```

Figure 15 provides a C-Code function that observes a specified amount of delay (given in milliseconds). **(VR)**

```
void ms_delay(int N)
{
    TMR1 = 0;
    while(TMR1 < N * 62.5);
}
```

Figure 15: C-Code Implementation of the *ms_delay* Function

With this, the LCD now needs to be initialized for its intended use. Fortunately, the datasheet for our LCD provided a startup sequence diagram that was used as a reference, and the function used to initialize the LCD is shown in Figure 16.

```
void init_LCD()
{
   RS = RW = 0;     // Configure RS/RW bits for writing internal operations

   us_delay(45000);          // 45000us > 40ms
   LCD_DATA = 0b00111000;    // Function set (8-bit DL, 2 lines, 5x8 dots)
   LCD_flash_EN();

   us_delay(50);             // 50us > 39us
   LCD_DATA = 0b00111000;    // Same instruction as before
   LCD_flash_EN();

   us_delay(50);             // 50us > 37us
   LCD_DATA = 0b00001100;    // Enable display/cursor/blinking)
   LCD_flash_EN();

   us_delay(50);             // 50us > 37us
   LCD_DATA = 0b00000001;    // Display clear
   LCD_flash_EN();

   us_delay(2000);           // 2000us > 1.53ms
   LCD_DATA = 0b00000110;    // Entry mode (cursor move direction / shift EN)
   LCD_flash_EN();

   return;          // End of initialization
}
```

Figure 16: C-Code Implementation of the *init_LCD* function

With the LCD fully initialized, a series of simple functions were written to write a particular character to the LCD, set the line of the LCD cursor, and clear the contents of the LCD were all written for basic interfacing. Each of these functions ensures that enough time has passed between instructions before sending its data or command, and properly sets the data, RS, and R/W bits for each operation. These basic interfacing functions are shown in Figure 17. **(VR)**

```
// Prints the provided character to the LCD
void char_to_LCD(char input)
{
   us_delay(2000);
   LCD_DATA = input;         // Set the LCD data bits to the input character
   RS = 1; RS = 1;           // Set RS high for data (done twice due to a bug)
   RW = 0;                   // Set RW lo to indicate a data write
   LCD_flash_EN();           // Flash the enable pin to write the character
}

// Sets the cursor to line 1 / line 2 of the LCD
void set_LCD_line(int line)
{
   us_delay(2000);
   RS = RW = 0;                          // Set RS/RW low to write an instruction
   LCD_DATA = line == 1 ? 0x80 : 0xC0;  // Change address based on line input
   LCD_flash_EN();                       // Flash enable pin to write address
}

// Clears LCD with built-in command (therefore setting cursor at address 0x00)
void clear_LCD()
{
    us_delay(2000);
    RS = RW = 0;
    LCD_DATA = 0x00;
    LCD_flash_EN();
}
```

Figure 17: C-Code Implementation of Basic LCD Interfacing Functions

Using these interfacing functions, a new function was written to print an entire string of characters to the LCD with automatic text wrapping to the second line. The function also allows the ‘absolute value’ character to be used to indicate an early jump to the second LCD line, and the ‘<’ and ‘>’ characters are automatically converted into left and right arrow characters, as this helps convey to the user when a scroll menu is being presented to them on the screen. Lastly, while this abstracted string printing function works well in most cases, a second function was written that simply prints a string to the LCD with no overhead. Both functions are shown in Figure 18. **(VR)**

With the LCD interfacing complete, one last function is required to read the signals of the tactile buttons used for user input. As shown in Table 16, each of the on-board buttons is assigned its own 3-bit pattern for identification, so the function used to interpret the buttons simply reads the bit lines and adds the appropriate values of 2 to a final integer result. The function then returns an integer representation of the 3-bit pattern sent by the on-board buttons, which can later be renamed using a C enumeration for better readability. Figure 19 provides the function used to read the on-board buttons. **(VR)**

```
// Prints a string of characters to LCD, starting at line 1 w/ text
// ('|' moves to line 2, '<' prints a left arrow, and '>' prints a right arrow)
void print_LCD(char* inputString)
{
    short int charCount = 0;         // Keep track of characters being printed

    set_LCD_line(1);                 // Begin letters at first line of LCD
    while(*inputString)              // Iterate through the input string
    {
     // If line 1 is full or a special character is read, move to line 2
        if (charCount == 16 || *inputString == '|')
        {
            set_LCD_line(2);
            charCount = -100;   // (To make sure this doesn't fire again)
        }

     // Print the current character (unless it's the special character)
        if (*inputString != '|')
        {
         // Print arrows for special characters, or just the character itself
            if      (*inputString == '<') char_to_LCD(0x7F);
            else if (*inputString == '>') char_to_LCD(0x7E);
            else                          char_to_LCD(*inputString);
        }

     // Move to the next character and increment the character count
        inputString++;
        charCount++;
    }
}

// Prints to LCD
void print_LCD_raw(char* string)
{
    while(*string) char_to_LCD(*string++);
}
```

Figure 18: C-Code Implementation of Practical LCD Interfacing Functions

```
int read_buttons()
{
    int button_read = 0;                        //  Int | Button
    ms_delay(1);                                // -----+--------
                                                //   0  | None
 // Read the 3 bits of button data input       //   1  | Center
    if (PORTGbits.RG8) button_read += 4;        //   2  | Back
    if (PORTGbits.RG7) button_read += 2;        //   3  | Macro
    if (PORTGbits.RG6) button_read += 1;        //   4  | Down
                                                //   5  | Up
 // Return the button read                     //   6  | Left
    return button_read;                         //   7  | Right
}
```

Figure 19: C-Code Implementation of the *read_buttons* Function

With the hardware interfacing complete, the next onboard implementation step involves designing a menu navigation system in software, which was done by implementing three different types of LCD screen C-structures. First of the three, the message screen structure can be provided with a C-string array of all of the different prompts a particular message screen should scroll between, as well as an integer that tells the structure the size of the prompt array. The structure also uses another integer variable to keep track of which element of the message array is being shown at a given time (in most cases, this is manually initialized to zero). Lastly, a globally-defined constant named GLOBAL_MESSAGE_SCREEN_DELAY_SEC defines how many seconds one particular message in a prompt list should show on screen before moving to the next one in the list. This constant defines the message delay for all message screens. Figure 20 provides the C code that defines the message screen structure and its 'method' functions. **(VR)**

```
typedef struct Message_Screen
{
    const char** SCREEN_PROMPTS;           // Pointer to array of screen prompts
    unsigned short int current_prompt;     // Specifies current screen prompt
    const unsigned short int NUM_PROMPTS;  // Specifies number of screen prompts

 // Function pointers to "init" and "update" methods for message
    const void (*init)(struct Message_Screen*);
    const void (*update)(struct Message_Screen*);

} Message_Screen;


void message_init(Message_Screen* message)
{
 // Reset the timer and message iterator
    TMR4 = 0; second_counter = 0;
    message->current_prompt = 0;

 // Clear the LCD and print the first message
    clear_LCD();
    print_LCD(message->SCREEN_PROMPTS[message->current_prompt]);

    return;
}

void message_update(Message_Screen* message)
{
 // Move to the next message in the array (treating it as a circular array)
    message->current_prompt = (message->current_prompt + 1) %
    message->NUM_PROMPTS;

 // Clear the LCD and print the new message
    clear_LCD();
    print_LCD(message->SCREEN_PROMPTS[message->current_prompt]);

    return;
}
```

Figure 20: Structure Declaration of an LCD Message Screen

Similar to the message screen structure, the scroll menu structure can be provided with a C-string array of all of the different menu options a particular menu should have, as well as an integer that tells the structure the size of the menu options array. The structure also uses another integer variable to keep track of which menu option is selected at a given time, as well as another C-string variable that is printed as a menu title. Typically, the on-board menu titles are in all uppercase, and the menu options have left and right arrows on the sides of each option to indicate that the user is being presented a menu. Both of these features are not directly provided by the structure, and are instead provided manually in the initialization of each of the scroll menus. Figure 21 provides the C code that defines the message screen structure, along with the definition of its initialization and update functions. **(VR)**

```
typedef struct Scroll_Menu
{
    const char** MENU_OPTIONS;            // Pointer to array of menu options
    short int current_choice;             // Specifies current choice
    const unsigned short int NUM_CHOICES; // Specifies number of menu choices
    const char* TITLE;                    // Specifies the title for the menu

 // Function pointers to "init" and "update" methods for menu
    void (*init)(struct Scroll_Menu*);
    void (*update)(struct Scroll_Menu*, const Event);

} Scroll_Menu;

void menu_init(Scroll_Menu* menu)
{
    menu->current_choice = 0; // Reset current menu choice

 // Clear the LCD and print the menu title & default option
    clear_LCD(); print_LCD(menu->TITLE);
    print_LCD(menu->MENU_OPTIONS[menu->current_choice]);

    return;
}

void menu_update(Scroll_Menu* menu, const Event event)
{
 // Only clear the screen & reprint the title if function called w/ no event
    if (event == NONE)
        { clear_LCD(); print_LCD(menu->TITLE); }

 // Move one to the right (circular) if the up/right buttons are pressed
    if (event == B_UP   || event == B_RIGHT)
        menu->current_choice = (menu->current_choice + 1) %
        menu->NUM_CHOICES;

 // Move one to the left (circular) if the down/left buttons are pressed
    if (event == B_DOWN || event == B_LEFT)
        menu->current_choice = (menu->current_choice + menu->NUM_CHOICES - 1)
        % menu->NUM_CHOICES;

 // Update the second line of the menu with the new choice selection
    print_LCD(menu->MENU_OPTIONS[menu->current_choice]);
    return;
}
```

Figure 21: Structure Declaration of an LCD Scroll Menu Screen

Lastly, the parameter prompt structure (arguably the most complicated out of the three) can be provided a C-String array of titles for each parameter prompt, another C-string array specifying the display units for each parameter, an integer specifying the number of parameters (and therefore specifying the size of the previous two arrays), and a 2D array that specifies the minimum, maximum, step, and default values of each parameter. Additionally, because parameter menus often results in the generation a draw packet, the structure provides a 'build packet' function along with the usual 'initialize' and 'update' functions, as shown in Figure 22: **(VR)**

```
typedef struct Param_Prompt_List
{
    const char** PARAM_PROMPTS;          // Array of parameter title prompts
    const char** DISPLAY_UNITS;          // Array of display unit strings
    short int current_param;             // Specifies the current parameter
    short int* param_values;             // Specifies values of all parameters
    const unsigned short int NUM_PARAMS; // Specifies number of total parameters

 // 2D integer array defining the min, max, default, and step for each parameter
    const short int (*MIN_MAX_STEP_DEFAULT)[4];

 // Function pointers to "init" and "update" methods for parameter list
 // (Update function returns the index of the current parameter the list is on)
    void (*init)(struct Param_Prompt_List*);
    short int (*update)(struct Param_Prompt_List*, const Event);

 // Function pointer used to construct a shape packet from parameter values
    void (*build_packet)(struct Param_Prompt_List*, const unsigned short int SHAPE, char* [50]);
}
Param_Prompt_List;

void print_param(short int value, const char* UNITS)
{
 // Convert the value that needs printed from an integer to a C-String
    char temp[5]; sprintf(temp, "%i", value);

 // Clear the second LCD line and set the cursor at the start of line 2
    set_LCD_line(2); print_LCD_raw("                ");
    set_LCD_line(2);

 // Print a right arrow, a space, the converted parameter value, and a space
    char_to_LCD(0x7E);   print_LCD_raw(" ");
    print_LCD_raw(temp); print_LCD_raw(" ");

 // Print the unit label depending on the unit variable
    if (!centimeters && UNITS == "cm") print_LCD_raw("in");
    else                               print_LCD_raw(UNITS);

    return;
}

void param_init(Param_Prompt_List* param_list)
{
 // Reset to the first parameter in list
    param_list->current_param = 0;
    const short int index = param_list->current_param;

 // Reset all parameter values
    short unsigned int i;
    for (i = 0; i < param_list->NUM_PARAMS; ++i)
        param_list->param_values[i] = param_list->MIN_MAX_STEP_DEFAULT[i][3];

 // Print the current parameter list to the screen
    clear_LCD(); set_LCD_line(1);
    print_LCD_raw(param_list->PARAM_PROMPTS[index]);
    print_param(param_list->param_values[index], param_list->DISPLAY_UNITS[index]);

    return;
}
```

```
short int param_update(Param_Prompt_List* param_list, const Event event)
{
 // Save the index of the current parameter to use in the succeeding code
    short int index = param_list->current_param;

 // If the up or right buttons are pressed...
    if (event == B_UP || event == B_RIGHT)
    {
     // Increment current parameter's value by its specified step value
     // (If this exceeds the parameter's maximum value, decrement it back)
        param_list->param_values[index] += param_list->MIN_MAX_STEP_DEFAULT[index][2];
        if (param_list->param_values[index] > param_list->MIN_MAX_STEP_DEFAULT[index][1])
           param_list->param_values[index] -= param_list->MIN_MAX_STEP_DEFAULT[index][2];
    }

 // If the down or left buttons are pressed...
    else if (event == B_DOWN || event == B_LEFT)
    {
     // Decrement current parameter's value by its specified step value
     // (If this falls below the parameter's minimum value, increment it back)
        param_list->param_values[index] -= param_list->MIN_MAX_STEP_DEFAULT[index][2];
        if (param_list->param_values[index] < param_list->MIN_MAX_STEP_DEFAULT[index][0])
           param_list->param_values[index] += param_list->MIN_MAX_STEP_DEFAULT[index][2];
    }

 // If the center button is pressed, move on to the next parameter
    else if (event == B_CENTER) param_list->current_param += 1;

 // If the back button is pressed, move back to the previous parameter
    else if (event == B_BACK)   param_list->current_param -= 1;

    index = param_list->current_param;  // (Resave the index again for logic)

 // If the user moved to a new parameter menu & all parameters are not specified
    if ((event == B_CENTER || event == B_BACK) && param_list->current_param < param_list->NUM_PARAMS)
    {
     // Clear the LCD and print the title of the new parameter
        clear_LCD(); set_LCD_line(1);
        print_LCD_raw(param_list->PARAM_PROMPTS[index]);
    }

 // Print the current parameter list to the screen
 // (only do this if the user has not gone to the shape menu or the draw screen)
    if (param_list->current_param >= 0 && param_list->current_param < param_list->NUM_PARAMS)
        print_param(param_list->param_values[index], param_list->DISPLAY_UNITS[index]);

 // Return the updated parameter index for the menu
    return param_list->current_param;
}

void param_build_packet(Param_Prompt_List* param_list, const unsigned short int SHAPE, char*
packet[50])
{
    unsigned short int i;        // For loop iterator
    char* converted_int[3];      // Holds C-string of integer conversion

 // Begin the shape packet using the sentinel value for the chosen shape
    strcpy(packet, "<d");
    sprintf(converted_int, "%i", SHAPE);
    strcat(packet, converted_int);

 // Add each of the specified parameter values to the packet
    for (i = 0; i < param_list->NUM_PARAMS; ++i)
    {
        strcat(packet, ":");
        sprintf(converted_int, "%i", param_list->param_values[i]);
        strcat(packet, converted_int);
    }

 // Finish the packet
    strcat(packet, ">");
    return packet;
}
```

Figure 22: Structure Declaration for an LCD Parameter Prompt Screen Set

With these structure definitions, a separate C file was populated with declarations of every type of LCD screen or menu that could be displayed to the user at any given time. Figures 23 through 25 provide the declarations of message, scroll menu, and parameter menu screens, respectively.

```
#define GLOBAL_MESSAGE_SCREEN_DELAY_SEC 2    // Two second delay between screens

#define MESSAGE_STARTUP_NUM_PROMPTS 1        // Startup screen
const char* MESSAGE_STARTUP_PROMPTS[MESSAGE_STARTUP_NUM_PROMPTS] = {"Press any button|to
begin..."};
Message_Screen startup_screen = {MESSAGE_STARTUP_PROMPTS, 0, MESSAGE_STARTUP_NUM_PROMPTS,
message_init, message_update};

#define MESSAGE_CALIBRATE_NUM_PROMPTS 2      // Calibration screen
const char* MESSAGE_CALIBRATE_PROMPTS[MESSAGE_CALIBRATE_NUM_PROMPTS] = {"Calibrating,|please
wait... >", "Thank you for|your patience! >"};
Message_Screen calibrate_screen = {MESSAGE_CALIBRATE_PROMPTS, 0, MESSAGE_CALIBRATE_NUM_PROMPTS,
message_init, message_update};

#define MESSAGE_HOME_NUM_PROMPTS 2           // Returning home screen
const char* MESSAGE_HOME_PROMPTS[MESSAGE_HOME_NUM_PROMPTS] = {"Returning home,|please wait...
>", "Thank you for|your patience! >"};
Message_Screen home_screen = {MESSAGE_HOME_PROMPTS, 0, MESSAGE_HOME_NUM_PROMPTS, message_init,
message_update};

#define MESSAGE_FREE_DRAW_NUM_PROMPTS 3      // Free draw screens
const char* MESSAGE_FREE_DRAW_PROMPTS[MESSAGE_FREE_DRAW_NUM_PROMPTS] = {"Use arrows to|move
marker,   >", "Center button to|engage marker, >", "Back button to|exit free draw >"};
Message_Screen free_draw_screen = {MESSAGE_FREE_DRAW_PROMPTS, 0, MESSAGE_FREE_DRAW_NUM_PROMPTS,
message_init, message_update};

#define MESSAGE_SETTINGS_HOME_NUM_PROMPTS 2  // New home confirmation screen
const char* MESSAGE_SETTINGS_HOME_PROMPTS[MESSAGE_SETTINGS_HOME_NUM_PROMPTS] = {"New home
has|been set!      >", "Press any button|to return...   >"};
Message_Screen home_confirm_screen = {MESSAGE_SETTINGS_HOME_PROMPTS, 0,
MESSAGE_SETTINGS_HOME_NUM_PROMPTS, message_init, message_update};

#define MESSAGE_SETTINGS_MACRO_NUM_PROMPTS 2 // New macro confirmation screen
const char* MESSAGE_SETTINGS_MACRO_PROMPTS[MESSAGE_SETTINGS_MACRO_NUM_PROMPTS] = {"New macro
has|been set!      >", "Press any button|to return...   >"};
Message_Screen macro_confirm_screen = {MESSAGE_SETTINGS_MACRO_PROMPTS, 0,
MESSAGE_SETTINGS_MACRO_NUM_PROMPTS, message_init, message_update};

#define MESSAGE_SETTINGS_UNITS_NUM_PROMPTS 2 // New units confirmation screen
const char* MESSAGE_SETTINGS_UNITS_PROMPTS[MESSAGE_SETTINGS_UNITS_NUM_PROMPTS] = {"Display
units|have been set! >", "Press any button|to return...   >"};
Message_Screen units_confirm_screen = {MESSAGE_SETTINGS_UNITS_PROMPTS, 0,
MESSAGE_SETTINGS_UNITS_NUM_PROMPTS, message_init, message_update};

#define MESSAGE_DRAW_SHAPE_NUM_PROMPTS 1     // Drawing shape confirmation screen
const char* MESSAGE_DRAW_SHAPE_PROMPTS[MESSAGE_DRAW_SHAPE_NUM_PROMPTS] = {"Drawing
shape,|please wait..."};
Message_Screen drawing_screen = {MESSAGE_DRAW_SHAPE_PROMPTS, 0, MESSAGE_DRAW_SHAPE_NUM_PROMPTS,
message_init, message_update};

#define MESSAGE_MACRO_NUM_PROMPTS 2          // Macro confirmation screen
const char* MESSAGE_MACRO_PROMPTS[MESSAGE_MACRO_NUM_PROMPTS] = {"Performing macro,|please
wait...", "Press any button|to exit..."};
Message_Screen macro_screen = {MESSAGE_DRAW_SHAPE_PROMPTS, 0, MESSAGE_MACRO_PROMPTS,
message_init, message_update};
```

Figure 23: LCD Message Screen Declarations

```
#define MENU_OP_SIZE 5       // Operation menu
const char* MENU_OP_OPTIONS[MENU_OP_SIZE] = {"|<  Draw Shape  >", "|<  Free
Draw   >", "|<  Go To Home  >", "|<  Calibrate   >", "|<   Settings   >"};
Scroll_Menu op_menu = {MENU_OP_OPTIONS, 0, MENU_OP_SIZE, "CHOOSE AN OPTION",
menu_init, menu_update};
const State MENU_OP_NEXT_STATES[MENU_OP_SIZE] = {SHAPE_SELECT, FREE_DRAW,
RETURNING_HOME, CALIBRATION, SETTINGS};

#define MENU_SHAPE_SIZE 7    // Preset shape menu
const char* MENU_SHAPE_OPTIONS[MENU_SHAPE_SIZE] = {"|<     Line     >", "|<
Rectangle  >", "|<   Triangle   >", "|<   Polygon    >", "|<     Arc
>", "|<    Circle    >", "|<   Sinusoid   >"};
Scroll_Menu shape_menu = {MENU_SHAPE_OPTIONS, 0, MENU_SHAPE_SIZE, " SELECT A
SHAPE ", menu_init, menu_update};
bool came_from_macro = false;

#define MENU_HOMES_SIZE 6    // Home positions menu
const char* MENU_HOMES_OPTIONS[MENU_HOMES_SIZE] = {"|<   Top Left   >", "|<
Top Right  >", "|< Bottom Left  >", "|< Bottom Right >", "|<    Center
>", "|<    Custom    >"};
Scroll_Menu homes_menu = {MENU_HOMES_OPTIONS, 1, MENU_HOMES_SIZE, "  SET
HOME POS  ", menu_init, menu_update};

#define MENU_MACRO_SIZE 3    // Macro options menu
const char* MENU_MACRO_OPTIONS[MENU_MACRO_SIZE] = {"|<  Go To Home  >", "|<
Calibrate   >", "|< Shape Preset >"};
Scroll_Menu macro_menu = {MENU_MACRO_OPTIONS, 0, MENU_MACRO_SIZE, " MACRO
FUNCTION ", menu_init, menu_update};

#define MENU_UNITS_SIZE 2    // Units options menu
const char* MENU_UNITS_OPTIONS[MENU_UNITS_SIZE] = {"|<    Inches    >", "|<
Centimeters  >"};
Scroll_Menu units_menu = {MENU_UNITS_OPTIONS, 0, MENU_UNITS_SIZE, "  SELECT
UNITS  ", menu_init, menu_update};

#define MENU_SETTINGS_SIZE 3 // Settings menu
const char* MENU_SETTINGS_OPTIONS[MENU_SETTINGS_SIZE] = {"|< Set Home Pos
>", "|< Change Macro >", "|< Change Units >"};
Scroll_Menu settings_menu = {MENU_SETTINGS_OPTIONS, 0, MENU_SETTINGS_SIZE,
"    SETTINGS    ", menu_init, menu_update};
```

Figure 24: LCD Scroll Menu Declarations

```
// Min, max, step, and default values for length parameter prompts
#define LENGTH_MIN     0
#define LENGTH_MAX     100
#define LENGTH_STEP    5
#define LENGTH_DEFAULT 20

// Min, max, step, and default values for angle parameter prompts
#define ANGLE_MIN      0
#define ANGLE_MAX      360
#define ANGLE_STEP     15
#define ANGLE_DEFAULT  0

#define PARAM_LINE_NUM_PARAMS 2    // Line parameter menu
const char* PARAM_LINE_PROMPTS[PARAM_LINE_NUM_PARAMS] = {"LENGTH:", "ANGLE:"};
const char* PARAM_LINE_UNITS[PARAM_LINE_NUM_PARAMS] = {"cm", "degrees"};
const short int LINE_MMSD[PARAM_LINE_NUM_PARAMS][4] = {{LENGTH_MIN, LENGTH_MAX, LENGTH_STEP, LENGTH_DEFAULT},
                                                       {ANGLE_MIN,  ANGLE_MAX,  ANGLE_STEP,  ANGLE_DEFAULT }};
short int line_param_values[PARAM_LINE_NUM_PARAMS] = {LENGTH_DEFAULT, ANGLE_DEFAULT};
Param_Prompt_List line_param_list = {PARAM_LINE_PROMPTS, PARAM_LINE_UNITS, 0, line_param_values, PARAM_LINE_NUM_PARAMS,
LINE_MMSD, param_init, param_update, param_build_packet};

#define PARAM_RECT_NUM_PARAMS 3    // Rectangle parameter menu
const char* PARAM_RECT_PROMPTS[PARAM_RECT_NUM_PARAMS] = {"LENGTH:", "WIDTH:", "ROTATION:"};
const char* PARAM_RECT_UNITS[PARAM_RECT_NUM_PARAMS] = {"cm", "cm", "degrees"};
const short int RECT_MMSD[PARAM_RECT_NUM_PARAMS][4] = {{LENGTH_MIN, LENGTH_MAX, LENGTH_STEP, LENGTH_DEFAULT},
                                                       {LENGTH_MIN, LENGTH_MAX, LENGTH_STEP, LENGTH_DEFAULT},
                                                       {ANGLE_MIN,  ANGLE_MAX,  ANGLE_STEP,  ANGLE_DEFAULT }};
short int rect_param_values[PARAM_RECT_NUM_PARAMS] = {LENGTH_DEFAULT, LENGTH_DEFAULT, ANGLE_DEFAULT};
Param_Prompt_List rect_param_list = {PARAM_RECT_PROMPTS, PARAM_RECT_UNITS, 0, rect_param_values, PARAM_RECT_NUM_PARAMS,
RECT_MMSD, param_init, param_update, param_build_packet};

#define PARAM_TRI_NUM_PARAMS 4     // Triangle parameter menu
const char* PARAM_TRI_PROMPTS[PARAM_TRI_NUM_PARAMS] = {"BASE LENGTH A:", "SIDELENGTH B:", "SIDELENGTH C:", "ROTATION:"};
const char* PARAM_TRI_UNITS[PARAM_TRI_NUM_PARAMS] = {"cm", "cm", "cm", "degrees"};
const short int TRI_MMSD[PARAM_TRI_NUM_PARAMS][4] = {{LENGTH_MIN, LENGTH_MAX, LENGTH_STEP, LENGTH_DEFAULT},
                                                     {LENGTH_MIN, LENGTH_MAX, LENGTH_STEP, LENGTH_DEFAULT},
                                                     {LENGTH_MIN, LENGTH_MAX, LENGTH_STEP, LENGTH_DEFAULT},
                                                     {ANGLE_MIN,  ANGLE_MAX,  ANGLE_STEP,  ANGLE_DEFAULT }};
short int tri_param_values[PARAM_TRI_NUM_PARAMS] = {LENGTH_DEFAULT, LENGTH_DEFAULT, LENGTH_DEFAULT, ANGLE_DEFAULT};
Param_Prompt_List tri_param_list = {PARAM_TRI_PROMPTS, PARAM_TRI_UNITS, 0, tri_param_values, PARAM_TRI_NUM_PARAMS, TRI_MMSD,
                                   param_init, param_update, param_build_packet};

#define PARAM_SINE_NUM_PARAMS 4    // Sinusoid parameter menu
const char* PARAM_SINE_PROMPTS[PARAM_SINE_NUM_PARAMS] = {"AMPLITUDE:", "PHASE:", "WAVELENGTH:", "NUMBER OF WAVES:"};
const char* PARAM_SINE_UNITS[PARAM_SINE_NUM_PARAMS] = {"cm", "degrees", "cm", ""};
const short int SINE_MMSD[PARAM_SINE_NUM_PARAMS][4] = {{LENGTH_MIN, LENGTH_MAX, LENGTH_STEP, LENGTH_DEFAULT},
                                                       {ANGLE_MIN,  ANGLE_MAX,  ANGLE_STEP,  ANGLE_DEFAULT },
                                                       {LENGTH_MIN, LENGTH_MAX, LENGTH_STEP, LENGTH_DEFAULT},
                                                       {0,          10,         1,           3             }};
short int sine_param_values[PARAM_SINE_NUM_PARAMS] = {LENGTH_DEFAULT, ANGLE_DEFAULT, LENGTH_DEFAULT, 3};
Param_Prompt_List sine_param_list = {PARAM_SINE_PROMPTS, PARAM_SINE_UNITS, 0, sine_param_values, PARAM_SINE_NUM_PARAMS,
SINE_MMSD, param_init, param_update, param_build_packet};

#define PARAM_CIRCLE_NUM_PARAMS 1  // Circle parameter menu
const char* PARAM_CIRCLE_PROMPTS[PARAM_CIRCLE_NUM_PARAMS] = {"RADIUS:"};
const char* PARAM_CIRCLE_UNITS[PARAM_CIRCLE_NUM_PARAMS] = {"cm"};
const short int CIRCLE_MMSD[PARAM_CIRCLE_NUM_PARAMS][4] = {{LENGTH_MIN, LENGTH_MAX, LENGTH_STEP, LENGTH_DEFAULT}};
short int circle_param_values[PARAM_CIRCLE_NUM_PARAMS] = {LENGTH_DEFAULT};
Param_Prompt_List circle_param_list = {PARAM_CIRCLE_PROMPTS, PARAM_CIRCLE_UNITS, 0, circle_param_values,
PARAM_CIRCLE_NUM_PARAMS, CIRCLE_MMSD  param_init, param_update, param_build_packet};

#define PARAM_POLY_NUM_PARAMS 3    // Regular polygon parameter menu
const char* PARAM_POLY_PROMPTS[PARAM_POLY_NUM_PARAMS] = {"NUMBER OF SIDES:", "SIDE LENGTH:", "ROTATION:"};
const char* PARAM_POLY_UNITS[PARAM_POLY_NUM_PARAMS] = {"", "cm", "degrees"};
const short int POLY_MMSD[PARAM_POLY_NUM_PARAMS][4] = {{0,          12,         1,           6             },
                                                       {LENGTH_MIN, LENGTH_MAX, LENGTH_STEP, LENGTH_DEFAULT},
                                                       {ANGLE_MIN,  ANGLE_MAX,  ANGLE_STEP,  ANGLE_DEFAULT }};
short int poly_param_values[PARAM_POLY_NUM_PARAMS] = {6, LENGTH_DEFAULT, ANGLE_DEFAULT};
Param_Prompt_List poly_param_list = {PARAM_POLY_PROMPTS, PARAM_POLY_UNITS, 0, poly_param_values, PARAM_POLY_NUM_PARAMS,
POLY_MMSD, param_init, param_update, param_build_packet};

#define PARAM_ARC_NUM_PARAMS 3     // Arc paramter menu
const char* PARAM_ARC_PROMPTS[PARAM_ARC_NUM_PARAMS] = {"RADIUS:", "ARC ANGLE:", "ROTATION:"};
const char* PARAM_ARC_UNITS[PARAM_ARC_NUM_PARAMS] = {"cm", "degrees", "degrees"};
const short int ARC_MMSD[PARAM_ARC_NUM_PARAMS][4] = {{LENGTH_MIN, LENGTH_MAX, LENGTH_STEP, LENGTH_DEFAULT},
                                                     {ANGLE_MIN,  ANGLE_MAX,  ANGLE_STEP,  ANGLE_DEFAULT },
                                                     {ANGLE_MIN,  ANGLE_MAX,  ANGLE_STEP,  ANGLE_DEFAULT }};
short int arc_param_values[PARAM_ARC_NUM_PARAMS] = {LENGTH_DEFAULT, ANGLE_DEFAULT, ANGLE_DEFAULT};
Param_Prompt_List arc_param_list = {PARAM_ARC_PROMPTS, PARAM_ARC_UNITS, 0, arc_param_values, PARAM_ARC_NUM_PARAMS, ARC_MMSD,
                                   param_init, param_update, param_build_packet};
```

Figure 25: LCD Parameter Prompt Declarations

With all possible LCD screen states declared, the remaining portion of the implementation involves designing a finite state machine that can reliably transition between LCD states depending on specific events. To begin this process, enumerated names were given to a set of all events that can trigger an on-board state change, as well as a list of possible states that the on-board menu can be in at any given time. Figure 26 provides the declarations of these enumerated events and states. **(VR)**

```
// ENUMERATED EVENTS ////////////////////////////////////////////////////////

// Events (the first 8 events MUST be listed in this order at the beginning)
typedef enum {NONE, B_CENTER, B_BACK, B_MACRO, B_DOWN, B_UP, B_LEFT, B_RIGHT,
              BLE, TIMER, FALLING_EDGE} Event;

// ENUMERATED STATES ///////////////////////////////////////////////////////

// States (listed in nested order of which states can lead to where)
typedef enum
{
    STARTUP,
    OP_MENU,
        SHAPE_SELECT,
            PARAM_SELECT,
            DRAWING_SHAPE,
        FREE_DRAW,
        RETURNING_HOME,
        CALIBRATION,
        SETTINGS,
            SETTING_SUBMENU,
                SET_CUSTOM_HOME,
            SETTING_CONFIRM,
        MACRO_FUNCTION
}
State;
```

Figure 26: Enumerated Events and States for On-Board Controls

As shown in Figure 26, the events that can cause a change in the state of the on-board menu consist of button presses, a bluetooth interruption, a timer going off, or a button's falling edge (in this case, this event triggers when no buttons are pressed). To detect if any of these events have occurred at any given time, a function was written that can detect all of these events in a particular order and return the first event that was detected to have occurred (or return NONE if no such event occurred). Figure 27 provides the implementation of this function. **(VR)**

```
// Event listener function (ideally should be run on each clock cycle)
Event check_events()
{
 // Remember the previous button pressed, then read the on-board buttons
    prev_button = current_button;
    current_button = read_buttons();

 // If a button was pressed (rising edge), return the button that was pressed
    if (prev_button == 0 && current_button > 0)      return current_button;

 // If a button was released (falling edge), return the FALLING_EDGE event
 // (This event only occurs when ALL buttons have been released)
    else if (prev_button > 0 && current_button == 0) return FALLING_EDGE;

 // If TMR4 reaches 1 second...
    if (TMR4 >= 62500)
    {
     // Increment the second_counter and reset TMR4
        second_counter += 1;
        TMR4 = 0;

     // If the message screen delay was reached
        if (second_counter >= GLOBAL_MESSAGE_SCREEN_DELAY_SEC)
        {
         // Reset the second counter and return the TIMER event
            second_counter = 0;
            return TIMER;
        }
    }

 // Otherwise, if no events were detected, indicate no events have occurred
    return NONE;
}
```

Figure 27: C-Code Implementation of the *check_events* Function

To complete the implementation of the on-board controls, one final function can be written that takes in the current state of the on-board menu and the last event that occurred, and properly updates the state of the onboard once invoked. Additionally, the function can be given a character array used to store any packets that may be generated as a result of a state change (normally if the menu is in the FREE_DRAW or DRAWING_SHAPE states). In order to accomplish this, the function consists of a series of nested switch statements that accounts for every possible state of the menu, along with some initialization code at the beginning that handles any bluetooth interruptions or use of the 'macro' button. Figure 28 provides this state-transition function. **(VR)**

```
State update(State current_state, Event last_event, char* packet[50])
{
 // Used as a safeguard (ideally, only call this function when an event occurs)
    if (last_event == NONE) return current_state;

    short int param_index;       // Used in parameter states
    unsigned short int i;        // Used as as for-loop iterator
    bool macro_disabled = false; // Indicates if macro is currently disabled

    if (last_event == B_MACRO)
    {
        for (i = 0; i < NUM_DISABLED_MACRO_STATES; ++i)
            if (current_state == MACRO_DISABLED[i])
                macro_disabled = true;

        if (!macro_disabled)
        {
         // Macro handler
            strcpy(packet, macro_packet); macro_menu.init(&macro_menu);
            return MACRO_FUNCTION;
        }
    }

 // Find the current state in the first switch statement, then
 // perform whatever is specified in that state for the last read event
    switch(current_state)
    {
    case MACRO_FUNCTION:
        switch(last_event)
        {
            case B_UP: case B_RIGHT: case B_DOWN: case B_LEFT:
            case B_CENTER: case B_BACK: case B_MACRO:
                op_menu.init(&op_menu); return OP_MENU;

            case TIMER:
                macro_menu.update(&macro_menu, last_event); return MACRO_FUNCTION;

            default: return MACRO_FUNCTION;
        }
        break;


    case STARTUP:      // Starting screen (message screen)
        switch(last_event)
        {
            case B_UP: case B_RIGHT: case B_DOWN: case B_LEFT:
            case B_CENTER: case B_BACK: case B_MACRO:
                op_menu.init(&op_menu); return OP_MENU;

            default: return STARTUP;
        }
        break;

    case OP_MENU:      // Operation menu (scroll menu)
        switch(last_event)
        {
            case B_UP: case B_RIGHT: case B_DOWN: case B_LEFT:
                op_menu.update(&op_menu, last_event); return OP_MENU;

            case B_CENTER:    // Read selected choice
                came_from_macro = false; // (To account for shape macro)
                switch(op_menu.current_choice)
                {
                    case 0:  shape_menu.init(&shape_menu);             break;
                    case 1:  free_draw_screen.init(&free_draw_screen); break;
                    case 4:  settings_menu.init(&settings_menu);       break;

                    case 3:
                        strcpy(packet,"<c>");
                        calibrate_screen.init(&calibrate_screen);      break;

                    case 2:
                        strcpy(packet, home_pos_packet);
                        home_screen.init(&home_screen);                break;
```

```
                default: break; // (Should never occur)
            }
            return MENU_OP_NEXT_STATES[op_menu.current_choice];

        case B_BACK:      // Return to startup screen
            startup_screen.init(&startup_screen); return STARTUP;

        default: return OP_MENU;
    }
    break;

case FREE_DRAW:
    switch(last_event)
    {
     // NEEDS CLEANING UP!
        case B_UP:         strcpy(packet,"<m1>"); return FREE_DRAW;
        case B_DOWN:       strcpy(packet,"<m3>"); return FREE_DRAW;
        case B_LEFT:       strcpy(packet,"<m4>"); return FREE_DRAW;
        case B_RIGHT:      strcpy(packet,"<m2>"); return FREE_DRAW;
        case B_CENTER:     strcpy(packet,"<s>");  return FREE_DRAW;
        case FALLING_EDGE: strcpy(packet,"<m0>"); return FREE_DRAW;

        case B_BACK:
            op_menu.update(&op_menu, NONE); return OP_MENU;

        case TIMER:
            free_draw_screen.update(&free_draw_screen); return FREE_DRAW;

        default: return FREE_DRAW;
    }
    break;

case RETURNING_HOME:
    switch(last_event)
    {
        case B_BACK:
            op_menu.update(&op_menu, NONE); return OP_MENU;

        case TIMER:
            home_screen.update(&home_screen); return RETURNING_HOME;

        default: return RETURNING_HOME;
    }
    break;

case CALIBRATION:
    switch(last_event)
    {
        case B_BACK:
            op_menu.update(&op_menu, NONE); return OP_MENU;

        case TIMER:
            calibrate_screen.update(&calibrate_screen); return CALIBRATION;

        default: return CALIBRATION;
    }
    break;


case SHAPE_SELECT: // Shape selection menu (scroll menu)
    switch(last_event)
    {
        case B_UP: case B_RIGHT: case B_DOWN: case B_LEFT:
            shape_menu.update(&shape_menu, last_event);
            return SHAPE_SELECT;

        case B_CENTER:
            (*PARAM_MENUS[shape_menu.current_choice]).init
            (PARAM_MENUS[shape_menu.current_choice]);
            return PARAM_SELECT;

        case B_BACK:
            if (came_from_macro)
            {
                (*SETTING_SUBMENUS[settings_menu.current_choice]).update(SETTING_SUBMENUS
```

```
                    [settings_menu.current_choice], NONE);
                return SETTING_SUBMENU;
            }
            else
            {
                op_menu.update(&op_menu, NONE);
                return OP_MENU;
            }

        default: return SHAPE_SELECT;
    }
    break;

case PARAM_SELECT:
    switch(last_event)
    {
        case B_UP:   case B_RIGHT:  case B_DOWN:
        case B_LEFT: case B_CENTER: case B_BACK:
            (*PARAM_MENUS[shape_menu.current_choice]).update(PARAM_MENUS
              [shape_menu.current_choice], last_event);

            param_index = (*PARAM_MENUS[shape_menu.current_choice]).current_param;
            if (param_index == -1)
            {
                shape_menu.update(&shape_menu, NONE);
                return SHAPE_SELECT;
            }

            else if (param_index == (*PARAM_MENUS[shape_menu.current_choice]).NUM_PARAMS)
            {
                (*PARAM_MENUS[shape_menu.current_choice]).build_packet(PARAM_MENUS
                 [shape_menu.current_choice], shape_menu.current_choice, packet);

                if (came_from_macro)
                {
                    strcpy(macro_packet, packet);
                    (*SETTING_CONFIRM_SCREENS[settings_menu.current_choice]).init
                    (SETTING_CONFIRM_SCREENS[settings_menu.current_choice]);
                    return SETTING_CONFIRM;
                }
                else
                {
                    drawing_screen.init(&drawing_screen); return DRAWING_SHAPE;
                }
            }

            else return PARAM_SELECT;

        default: return PARAM_SELECT;
    }
    break;

case DRAWING_SHAPE:
    switch(last_event)
    {
        case B_UP: case B_RIGHT: case B_DOWN: case B_LEFT:
        case B_CENTER: case B_BACK: case B_MACRO:
            op_menu.init(&op_menu); return OP_MENU;

        default: return DRAWING_SHAPE;
    }
    break;

case SETTINGS:     // Settings menu (scroll menu)
    switch(last_event)
    {
        case B_UP: case B_RIGHT: case B_DOWN: case B_LEFT:
            settings_menu.update(&settings_menu, last_event);
            return SETTINGS;

        case B_CENTER:
           (*SETTING_SUBMENUS[settings_menu.current_choice]).init
            (SETTING_SUBMENUS[settings_menu.current_choice]);
            return SETTING_SUBMENU;
```

```
            case B_BACK:
                op_menu.update(&op_menu, NONE); return OP_MENU;

            default: return SETTINGS;
        }
        break;

    case SETTING_SUBMENU:
        switch(last_event)
        {
            case B_UP: case B_RIGHT: case B_DOWN: case B_LEFT:
                (*SETTING_SUBMENUS[settings_menu.current_choice]).update
                (SETTING_SUBMENUS[settings_menu.current_choice], last_event);
                return SETTING_SUBMENU;

            case B_CENTER:
                switch(settings_menu.current_choice)
                {
                    case 0: // Change Home
                        change_home_packet(home_pos_packet, (*SETTING_SUBMENUS
                        [settings_menu.current_choice]).current_choice + 1);
                        break;

                    case 1:
                        switch((*SETTING_SUBMENUS[settings_menu.current_choice]).current_choice)
                        {
                            case 0: strcpy(macro_packet, home_pos_packet); break; // Go to home
                            case 1: strcpy(macro_packet, "<c>");           break; // Calibrate

                            case 2:                                               // Preset shape
                                came_from_macro = true;
                                shape_menu.init(&shape_menu);
                                return SHAPE_SELECT;

                            default: break; // (Should never occur)
                        }

                    case 2: // Change Units
                        centimeters =(*SETTING_SUBMENUS[settings_menu.current_choice]).current_choice;
                        break;

                    default: break; // (Should never occur)
                }

             // Will occur for home and calibrate selections
                (*SETTING_CONFIRM_SCREENS[settings_menu.current_choice]).init
                (SETTING_CONFIRM_SCREENS[settings_menu.current_choice]);
                return SETTING_CONFIRM;

            case B_BACK:
                settings_menu.update(&settings_menu, NONE); return SETTINGS;

            default: return SETTING_SUBMENU;
        } break;

    case SETTING_CONFIRM:
        switch(last_event)
        {
            case B_UP: case B_RIGHT: case B_DOWN: case B_LEFT:
            case B_CENTER: case B_BACK: case B_MACRO:
                settings_menu.update(&settings_menu, NONE);
                return SETTINGS;

            case TIMER:
                (*SETTING_CONFIRM_SCREENS[settings_menu.current_choice]).update
                (SETTING_CONFIRM_SCREENS[settings_menu.current_choice]);
                return SETTING_CONFIRM;

            default: return SETTING_CONFIRM;
        }
        break;
    }
}
```

Figure 28: C-Code Implementation of the State Machine *update* Function

(5.2.3) Bluetooth Firmware

The RN4870 Bluetooth module is controlled using C code and connected to the UART2 pins of the PIC24 microcontroller. To use the Bluetooth module, we first needed to initialize some of the UART2 settings. We define variables for CTS (clear to send) and RTS (request to send). We set the TRIS control for the RTS pin. Lastly, we set the baud rate to 115200 and enabled the UART peripheral and UART transmission. Additionally, in the *init_U2* function, we set the UART2 peripheral to the variables that we defined before. We also make RTS the output and set RTS's default status. **(JM)**

Once the Bluetooth was initialized, we used two main functions to send and receive data between the Bluetooth module and the PIC24 microcontroller. The first of these functions is the *get_U2* function seen in Figure 29. This function is used to receive characters from the Bluetooth module to the microcontroller over the UART2 peripheral. When the function is called, it first asserts a request to send low, then waits for there to be any characters in the UART2 receive buffer. When there are characters in the buffer, request to send is asserted high, the first character of the buffer is removed, and it is returned by the function. **(JM)**

```
// wait for a new character to arrive to the UART2 serial port
char get_U2(void)
{
    RTS = 0;                    // assert Request To Send !RTS
    while (!U2STAbits.URXDA);  // wait for a new character to arrive
    RTS = 1;
    return U2RXREG;             // read the character from receive buffer
}
```

Figure 29: C-Code Implementation of the *get_U2* Function

Similar to the *get_U2* function, we have the *put_U2* function seen in Figure 30. This function is used to send characters to the Bluetooth module from the microcontroller over the UART2 peripheral. When the function is called, it first asserts clear to send low, then waits for there to be any characters in the UART2 transmit buffer. When there are characters in the buffer, clear to send is asserted high. Then the character passed into the function is pushed into the transmit buffer and returned from the function. **(JM)**

```
// send a character to the UART2 serial port
char put_U2 ( char c)
{
      while ( CTS);                // wait for !CTS, clear to send
      while ( U2STAbits.UTXBF);    // wait while Tx buffer full
      U2TXREG = c;
      return c;
}
```

Figure 30: C-Code Implementation of the *put_U2* Function

We used these two low level functions to create two more functions with the ability to transmit full strings of characters. The *write_BLE* function, seen in Figure 31, takes in a character pointer (c-string), and writes one character at a time using the *put_U2* function. **(JM)**

```
void write_BLE(char* cbuff, int size){
    int j;
    for (j=0; j<size; j++) {
        put_U2(cbuff[j]);
    }
}
```

Figure 31: C-Code Implementation of the *write_BLE* Function

Similar to the *write_BLE* function, the *read_BLE* function seen in Figure 32, takes in a c-string. This string should start with the null terminator, meaning that it is empty. The *get_U2* function is then called to get the first character out of the Bluetooth module's receive buffer. If the user sends more than one Bluetooth command in quick succession, the Bluetooth packets may clobber each other, causing corrupt packets. To counteract this, the function will clear everything in the character buffer, and restart reading if '<' is seen. **(JM)**

```
void read_BLE(char* cbuff) {
    // check for BLE transmit and echo if there is anything in the buffer
    if (U2STAbits.URXDA)
    {
        cbuff[0] = get_U2();                // Get first character
        if (cbuff[0] != '<') { return;}     // Return if not start of package
        for (int i=1; cbuff[i-1] != '>'; ++i) {
            cbuff[i] = get_U2();
            if (cbuff[i] == '<') i=0;
        }
    }
}
```

Figure 32: C-Code Implementation of the *read_BLE* Function

Since all packets follow the form:

<####>

the first character of a packet must be '<' or else the packet has been corrupted. So, if this character is not returned by the *get_U2* function, the *read_BLE* function will return without reading the rest of the buffer. If this character is returned by the *get_U2* function, the rest of the characters will be read out of the receive buffer until the ending '>' character is reached. Figure 33 shows this in the form of a flow chart. After reading the packet into the c-string, the character pointer representing the start of the string is passed into the *process_package* function. At the end of ever *main* loop the c-string is cleared. This is to avoid package corruption if Bluetooth commands are sent while the device is busy. **(JM)**

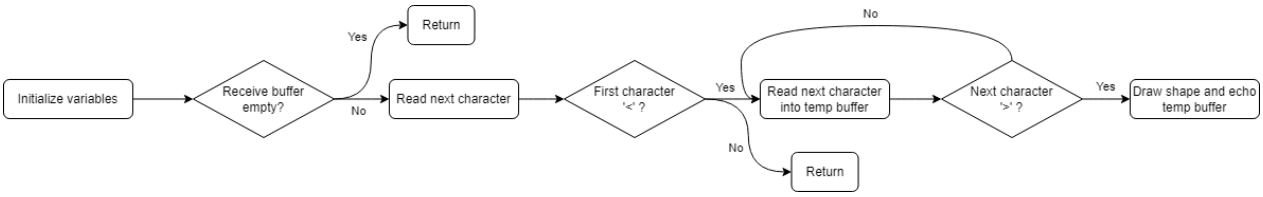

Figure 33: Bluetooth Firmware Flowchart

To allow for concurrent use of the onboard and smartphone application controls, the *U2RX* interrupt was required. Whenever characters are received over Bluetooth, the *U2RX* interrupt will be triggered, calling the *read_BLE* function to update the global character buffer. This interrupt can be seen in Figure 34.

```
void __attribute__((interrupt, no_auto_psv)) _U2RXInterrupt(void) {
    read_BLE(cbuff);

    // handle UART2 RX interrupt here
    IFS1bits.U2RXIF = 0; // clear interrupt flag
}
```

Figure 34: C-Code Implementation of the *U2RX* Interrupt

(5.2.4) Microcontroller Firmware

When the microcontroller is powered on, the servo is disengaged and the device is calibrated before it starts reading user input via Bluetooth or the onboard controls. When the servo is disengaged, a boolean is set, allowing the device to track the servo's orientation. This is accomplished by using the *engage_servo* and *disengage_servo* functions seen in Figure 35 and Figure 36 respectively. Both functions call the *rotate_servo* function, seen in Figure 37. This function uses PWM (pulse width modulation) on a PORTC pin, treating it as an oscillator. The orientation of the servo can then be changed depending on the PWM duty cycle. **(JM)**

```
void engage_servo (void) {
    rotate_servo(500, 19500);
    servo_engaged = true;
}
```

Figure 35: C-Code Implementation of the *engage_servo* Function

```
void disengage_servo (void) {
    rotate_servo(1350 , 18650);
    servo_engaged = false;
}
```

Figure 36: C-Code Implementation of the dis*engage_servo* Function

```
void rotate_servo (int high, int low) {
    ms_delay(100);

    for(int i=0; i<50; i++)
    {
        SERVO_SIGNAL = true;
        us_delay(high);
        SERVO_SIGNAL = false;
        us_delay(low);
    }
}
```

Figure 37: C-Code Implementation of the *rotate_servo* Function

Calibration allows the device to track the position of the drawing apparatus. This initial calibration will always start by positioning the drawing apparatus at the top-left corner of the whiteboard. After this, the device will engage its *x* motor until the right limit switch is hit, then do the same for the *y* motor, until the bottom limit switch is hit. This allows the device to set its bounds so it knows the whiteboard's size. From this point forward, any subsequent calibrations will move the drawing apparatus to the closest corner, allowing for a quicker calibration. Figure 38 shows the *inital_calibrate* and Figure 39 shows the *calibration* function. **(JM)**

```
void initial_calibrate() {
    calibrate();
    drive_motors(1000, 10000, RIGHT, 0,0,0);
    RIGHT_BOUND = global_x - BOUNCE_STEPS;
    drive_motors(0,0,0, 1000, 10000, DOWN);
    LOWER_BOUND = global_y - BOUNCE_STEPS;
}
```

Figure 38: C-Code Implementation of the *inital_calibrate* Function

```
void calibrate (void) {
    x_step = 0, y_step = 0;
    // Calibrate horizontal position
    bool x_direction = global_x > RIGHT_BOUND/2;
    move_to_xlimit(x_direction);
    drive_motors(MAX_FREQ, x_step, !x_direction, 0, 0, 0);

    // Calibrate vertical position
    bool y_direction = global_y > LOWER_BOUND/2;
    move_to_ylimit(y_direction);
    drive_motors(0, 0, 0, MAX_FREQ, y_step, !y_direction);
}
```

Figure 39: C-Code Implementation of the *calibrate* Function

While the machine is running, the firmware on the microcontroller will be listening for user input every millisecond. When it reads in user input from the smartphone application or the on-board controls, the input packet is then sent to the *process_package* function to be split into its command and parameters. Figure 40 illustrates this in the form of a Level 0 Flowchart. **(JM)**

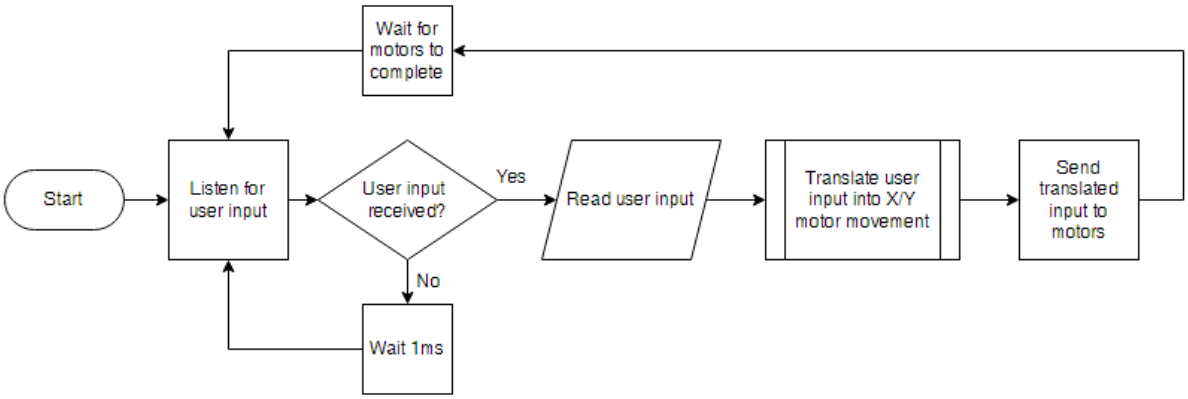

Figure 40: Level 0 Flowchart for Microcontroller Firmware

All of the possible packets can be seen in Table 1 (Section 2.3). Packets starting with 'd' are draw packets. These include the shape to draw, along with the shape's parameters. All parameters are integers, separated by colons within the packet. The *parse_packet* function is used to parse the c-string and return the parameters as a list of integers. This function can be seen in Figure 41. **(JM)**

```
void parse_packet(char* cbuff, int* params) {
    int param_num = 0;

    for (int i=4; cbuff[i] != '>'; ++i){
        if (cbuff[i] == ':') {
            ++param_num;    // Continue to next parameter
            continue;       // Parameters separated by colons
        }

        // Multiply current parameter by 10 then add the next least
        // significant digit
        params[param_num] = params[param_num]*10 + (cbuff[i] - 48);
    }
}
```

Figure 41: C-Code Implementation of the *parse_packet* Function

Regardless of the shape that is chosen, the firmware will always need to calculate how many steps each of the motors needs to take in order to move a specified distance. If we assume that the stepper motors require n steps to complete a full revolution (for our particular motors, n = 200

steps), and the marker's *x* and *y* movement is being driven by these motors with a gear of radius *r* cm, then *N* steps can be used to move the marker distance *D* cm, where **(VR)**

$$N = \frac{Dn}{2\pi r}$$

In our case, we calculated the conversion to be ~49.739262 steps per centimeter, and saved that conversion as a constant named CMSTEPS. **(JM)**

Two functions, *engage_xmotor* and *engage_ymotor*, are used to engage the x and y motors respectively. Both functions work similarly and start by enabling the specific motor's dedicated timer interrupt and oscillator. These are both disabled while the motors are not in use to avoid unwanted movement. Each of these functions takes in two parameters, a frequency and direction of which to drive the motors. The motor's dedicated timer's period register is set according to the following calculation:

$$Period = \frac{Internal\ Clock}{Prescaler * Frequency}$$

We found the actual maximum driving frequency to fall ~1200 Hz, so we set the code's maximum driving frequency to be 1000Hz. The direction parameter gets set into a global direction boolean for each motor, allowing the microcontroller to track how motors are moving at any time. These engage functions can be seen in Figure 42 and Figure 43. **(JM)**

```
void engage_xmotor(int freq, bool direction)
{
    _T3IE = 1;                          // Enable interrupt
    OC1CON = 0x000E;                    // Enable OC1CON with PR3
    int period = 62500/freq;     // Convert frequency to PR3 value
    PR3 = period;                       // Set PR3
    OC1RS = period/2;                   // Set 50% duty cycle using half of PR3
    HORIZONTAL_DIRECTION = global_x_dir = direction;
}
```

Figure 42: C-Code Implementation of the *engage_xmotor* Function

```
void engage_ymotor(int freq, bool direction)
{
    _T2IE = 1;                          // Enable interrupt
    OC2CON = 0x0006;                    // Enable OC2CON with PR2
    int period = 62500/freq;            // Convert frequency to PR3 value
    PR2 = period;                       // Set PR2
    OC2RS = period/2;                   // Set 50% duty cycle using half of PR2
    VERTICAL_DIRECTION = global_y_dir = direction;
}
```

Figure 43: C-Code Implementation of the *engage_ymotor* Function

In each of these functions, the specified motor frequency is converted into an integer period value that can be stored in the PR register for TMR2 or TMR3 (depending on which motor is being driven), which allows motor control to be interrupt-driven, therefore refraining from halting the entire program. Also, it should be noted that OC1RS and OC2RS are always set to 50% duty cycle, as changing stepper motor speed requires frequency modulation instead of pulse width modulation. **(VR)**

Using these two motor functions, the *drive_motors* function can drive the motors for a specific amount of steps. This function takes in drive frequency, steps to drive, and direction for both the *x* and *y* motors. One or both of the motors are then engaged, and kept engaged until either the desired step count is reached, or a limit switch is triggered. This function can be seen in Figure 44. **(JM)**

```
void drive_motors(int x_freq, int x_steps,
bool x_dir, int y_freq, int y_steps, bool y_dir){
    limit_flag = true;      // Initialize limit flag
    TMR2 = 0, TMR3 = 0;
    if (x_freq != 0) { engage_xmotor(x_freq, x_dir); }      // Enable horizontal motor
    if (y_freq != 0) { engage_ymotor(y_freq, y_dir); }      // Enable vertical motor
    x_step = 0, y_step = 0;                                 // Set step count to zero
    while(y_step <= y_steps && x_step <= x_steps && limit_flag);
    disable_motors();        // Trigger interrupt to stop motors
}
```

Figure 44: C-Code Implementation of the *drive_motors* Function

The motors are able to track the amount of steps they have moved using their oscillator and timer interrupt. Each oscillation, the motor will move one step and trigger its respective timer interrupt. In this interrupt both the current step variable and global step variables are incremented, keeping track of the current drive's progress and global device position. **(JM)**

A generic line drawer, seen in Figure 45, is used to draw all shapes, except sinusoids. This function takes in two main parameters, those being the length and angle of the line. First, the length of the line is converted into steps so it can be used in the *drive_motors* function. Trigonometric functions are then used to calculate the individual x and y steps of the line. The speeds of each motor are then calculated. The motor required to move farther is set to the maximum driving frequency of 1000 Hz, and the other motor is set to the following ratio:

$$Motor\ Speed\ = \frac{Max\ Frequency * Smaller\ Step\ Count}{Larger\ Step\ Count}$$

After calculating the speed and step count of each motor, the direction of each motor is calculated based on the angle of the line. Before sending the *x* and *y* motor parameters to the *drive_motors* function, these parameters are checked to ensure that the desired drawing will be within bounds. **(JM)**

```
bool line(float length, int angle, int delay){
    int steps = length*CMSTEPS;
    angle = (int)angle % 360;
    // Calculate horizontal and vertical steps
    int x_steps = abs(steps * cos((float)angle * PI/180));
    int y_steps = abs(steps * sin((float)angle * PI/180));
    int x_speed, y_speed;
    bool x_dir, y_dir;

    // Calculate speed of each motor based on which is designated more steps
    x_speed = x_steps >= y_steps ? MAX_FREQ : MAX_FREQ*((float)x_steps/y_steps);
    y_speed = y_steps >= x_steps ? MAX_FREQ : MAX_FREQ*((float)y_steps/x_steps);

    // Calculate the direction of motors based on the angle
    x_dir = (angle > 90 && angle < 270) ? LEFT : RIGHT;
    y_dir = (angle > 0 && angle < 180) ? UP : DOWN;

    // Drive the motors to draw the line
    if (!in_xbounds(x_steps, x_dir) || !in_ybounds(y_steps, y_dir)) { return false; }
    drive_motors(x_speed, x_steps, x_dir, y_speed, y_steps, y_dir);
    ms_delay(delay);

    return true;
}
```

Figure 45: C-Code Implementation of the *line* Function

As said before, this generic line function is used to draw all shapes except for sinusoids. Using the line function, we wrote functions to draw triangles, rectangles, and regular polygons. These can be seen in Figure 46, Figure 47, and Figure 48 respectively. Though the *draw_rectangle* function is self explanatory, the *draw_triangle* and *draw_polygon* functions were more complex. The *draw_triangle* function takes in three side parameters, and draws a triangle based on their length and order. For example, a (3,7,5) triangle would have a different orientation than a (5,7,3) triangle. This allows us to easily draw equilateral, isosceles, and scalene triangles. The *draw_polygon* function takes in two parameters, number of sides and side length. A regular polygon is then drawn by drawing a side, then increasing the next sides angle by the external angle of the polygon. **(JM)**

```
void draw_rectangle(int length, int height, int rotation)
{
    if (!servo_engaged) engage_servo(); // Engage servo
    if (!line(length, 0 + rotation, DRAW_DELAY)) { return; }
    if (!line(height, 270 + rotation, DRAW_DELAY)) { return; }
    if (!line(length, 180 + rotation, DRAW_DELAY)) { return; }
    if (!line(height, 90 + rotation, DRAW_DELAY)) { return; }
    if (servo_engaged) disengage_servo();   // Disengage servo
}
```

Figure 46: C-Code Implementation of the *draw_rectangle* Function

```
void draw_triangle(int a, int b, int c, int rotation)
{
    if (!servo_engaged) engage_servo(); // Engage servo
    float atheta = law_of_cosine(c, b, a);
    float ctheta = law_of_cosine(a, b, c);

    if (!line(b, ctheta + rotation, DRAW_DELAY)) { return; }
    if (!line(c, 180 + atheta + ctheta + rotation, DRAW_DELAY)) { return; }
    if (!line(a, 180 + rotation, DRAW_DELAY)) { return; }
    if (servo_engaged) disengage_servo();   // Disengage servo
}
```

Figure 47: C-Code Implementation of the *draw_triangle* Function

```
void draw_polygon(int num_sides, int length, int rotation)
{
    if (!servo_engaged) engage_servo(); // Engage servo
    // Calculate the angle at each vertex of polygon
    float num = 180*(num_sides-2), den = num_sides;
    float external_angle = 180 - num/den;
    // Draw each line of polygon by increasing the angle
    for (int i=1; i<= num_sides; i++) {
        if (!line(length, external_angle*i + rotation, DRAW_DELAY)) { return; }
        ms_delay(100);
    }
    if (servo_engaged) disengage_servo();   // Disengage servo
}
```

Figure 48: C-Code Implementation of the *draw_polygon* Function

Similar to regular polygons, circles are drawn using the same logic as the *draw_polygon* function. Some CNC machines draw circles as polygons with many sides. As the sides increase, the circle becomes smoother. We decided to take a similar approach, drawing circles as a many sided polygon, similar to CNC machines. The *arc* function allows us to draw partial and full circles by specifying the radius and arc angle. In terms of a regular polygon, the number of sides can be calculated as a function of the radius, and the number of sides to draw as a function of the arc angle. The *arc* function can be seen in Figure 49. To draw a circle, an arc is drawn with an arc angle of 360 degrees. **(JM)**

```
void arc(int rotation, int radius, int arc_angle){
    // Increasing side count increases circle accuracy
    int num_sides = radius+25, arc_sides = num_sides*((float)arc_angle/360);
    if (!arc_sides) { return; }
    // Calculate the angle at each vertex of polygon
    float internal_angle = 180*(num_sides-2)/num_sides;
    float external_angle = 180 - internal_angle;
    // radius * (2cos(internal_angle/2) -> Law of Cosine
    float side_length = (2 * cos((internal_angle/2) * PI/180)) * radius;
    // Draw each line of polygon by increasing the angle
    for (int i=3; i<= arc_sides+1; i++) {
        if (!line(side_length, external_angle*i + rotation + 90, 0)) { return; }
    }
}
```

Figure 49: C-Code Implementation of the *draw_arc* Function

As previously stated, sinusoids are the only shape that don't use the straight line drawing function in order to be observed. While we did attempt a method of sampling a sine wave and using straight lines to connect sampled points, this resulted in a very poor whiteboard drawing, and

took a significant amount of time to draw. Instead, the *draw_sinusoid* function computes a constant speed for the *x* motor, and then continuously varies the speed of the *y* motor while changing its direction at the appropriate time. However, following this approach to have the motors observe true sinusoidal curves proves to be rather difficult, as the speed of the *y* motor must continuously be changed at a rate proportional to the derivative of a sinusoidal curve - which just so happens to be another sinusoidal curve. Thus, a parabolic approximation was used instead, as the derivative of a parabolic function results in a straight line, meaning that the *y* motor speed could simply be incremented or decremented continuously to observe a close approximation of the desired result. Specifically, the following parabolic approximation of a sinusoidal curve was derived:

$$f(x) = \frac{-4A}{\pi^2} \times \left(\frac{2\pi x}{w} - \frac{\pi}{2}\right)^2 + A$$

This approximation fits a parabola along the points (0,0), (*w*/2, A), and (*w*, 0) for any arbitrary sine wave with wavelength *w* and amplitude *A*. This approximation can then be periodically repeated along the *x*-axis, and every other parabola can then be mirrored across the x-axis to achieve an approximation of the entire desired wave, resulting in the following modified version of what is now a repeated parabolic approximation:

$$f(x) = (-1)^{\text{floor}(2x/w)} \times \left[\frac{-4A}{\pi^2} \times \left(\frac{2\pi}{w} \bmod(x, w/2) - \frac{\pi}{2}\right)^2 + A\right]$$

Unlike a true sinusoid, this approximation's derivative resembles a triangle wave, meaning that the speed of the *y* motor needs to be changed at a linear rate to fully observe the approximation. To begin, the maximum value of $f'(x)$ can be proven to be $\frac{8A}{w}$, and using the calculations similar to the ones done to draw any arbitrary line, this max slope can be used to compute the highest possible speed at which the *x* and *y* motors can be driven. In the case where $\frac{8A}{w}$ is larger than one, the *y* motor will need to be observe the fastest possible speed of the stepper motors, while the *x* motor would then need to be driven at a speed equal to the ratio between the highest frequency and the value of $\frac{8A}{w}$. In the case where $\frac{8A}{w}$ is less than one, the *x* motor will instead need to be

observe the fastest possible speed of the stepper motors, while the *x* motor would then need to be driven at a speed equal to the product of the highest frequency and the value of $\frac{8A}{w}$. In code form,

```
// Compute the largest slope that the sinusoid graph will
const float MAX_SLOPE = 8.0 * (amplitude * 1.0) / (wavelength * 1.0);

// Compute the x speed and y max speed based on the largest slope of the plot
const int X_SPEED = MAX_SLOPE > 1 ? MAX_FREQ / MAX_SLOPE : MAX_FREQ;
const int Y_SPEED_MAX = MAX_SLOPE > 1 ? MAX_FREQ : MAX_FREQ / MAX_SLOPE;
```

Figure 50: C-Code Used to Compute the *x* and *y* Motor Speeds for Sinusoids

Next, the code needs a way to check if the speed of the *y* motor should be increasing or decreasing at any given time. To accomplish this, note that $f'(x)$ alternates between a positive and negative slope at constant intervals of $w/4$. By converting this length into a number of motor steps, the C code can use the modulus operator to detect if a corner of the derivative has been reached, with an additional computation required to shift these corner locations depending on the phase shift specified by the user. Additionally, to compute the value that the speed of the *y* motor should be incremented or decremented by at any given time, we can divide the maximum *y* speed (rise) by the number of steps equivalent to a quarter of the specified wavelength (run). Lastly, another variable can store the current *y* motor speed at a given time, the initial value of which will solely depend on the phase shift specified by the user. In code form,

```
// Set the appropriate global variables that the interrupts use for sinusoids
transition_point = CMSTEPS * wavelength / 4;
transition_offset = (CMSTEPS * wavelength / 360.0) * (phase % 180);

speed_step = Y_SPEED_MAX / (CMSTEPS * wavelength / 4);
current_y_speed = abs((-Y_SPEED_MAX / 90.0) * (phase % 180) + Y_SPEED_MAX);

const bool Y_START_DIR = (phase >= 90 && phase < 270) ? DOWN : UP;
if (phase % 180 >= 90) speed_step *= -1;
```

Figure 51: C-Code Used to Control *y* Motor Speed and Direction for Sinusoids

With this setup complete, a global boolean variable can be set to communicate to each of the motor interrupts that a sinusoid is being drawn to ensure that the appropriate interrupt handling code is run. For the *y* motor interrupt, it will simply grab the next motor speed it needs to observed whenever the interrupt is invoked, like so:

```
void _ISRFAST _T2Interrupt(void)
{
 ...

    if (drawing_sine)
    {
        PR2 = 62500/current_y_speed;
        OC2RS = PR2 / 2;
    }

    _T2IF = 0;        // Clear interrupt
}
```

Figure 52: C-Code interrupt logic for the *y* Motor for Sinusoids

For the *x* motor, every step the x motor moves will result in a different speed that the *y* motor needs to observe at that particular time. Thus, the *x* motor interrupt will update the speed (either by incrementing or decrementing the current speed by the predetermined speed step value), check whether or not the program needs to swap between incrementing or decrementing (depending on if the motor step distance has hit a corner of $f'(x)$), and lastly, it will check to see if the direction of the *y* motor needs to be toggled (which only needs to occur at the peaks and troughs of the wave being drawn). In code form,

```
void _ISRFAST _T3Interrupt(void)
{
    ...

    if (drawing_sine)
    {
     // Update the current y speed, and revert the change if it's below minimum
        current_y_speed -= speed_step;
        if (current_y_speed < MIN_SPEED) current_y_speed = MIN_SPEED;

     // If we've reached a corner of the approx. derivative, invert speed step
        if ((x_step + transition_offset) % transition_point == 0)
        {
            speed_step *= -1;
            current_y_speed -= speed_step;
        }

     // If we've reached a peak/trough of the curve, invert y direction
        if ((x_step + transition_point + transition_offset) % (2 * transition_point) == 0)
        {
            global_y_dir = !global_y_dir;
            VERTICAL_DIRECTION = global_y_dir;
        }
    }

    _T3IF = 0;        // Clear interrupt flag
}
```

Figure 53: C-Code Interrupt Logic for the *x* Motor for Sinusoids

MECHANICAL SKETCH

Figure 49 shows the 4’ x 4’ whiteboard with the following attachments: three stepper motors (two x-axis and one y-axis), pulleys attached to the stepper motors with belts to move the drawing apparatus (in the y-direction) and y-stepper motor (in the x-direction), single rail for the y-stepper motor to move in the x-direction and two rails for the drawing apparatus to move in the y-direction, and two wheels attached at the bottom so that mechanism can move freely along the bottom of the whiteboard. These attachments to the whiteboard are not to scale, as they will be much smaller on the 4’ x 4’ whiteboard, but they were drawn larger for a better visualization. Figure 50 shows the more detailed servo/marker configuration (drawing apparatus). The servo arm will move roughly 20 degrees where it will touch a device screwed into the back of the block, which holds the marker, where it will compress the springs and bring the marker to the whiteboard. **(DA)**

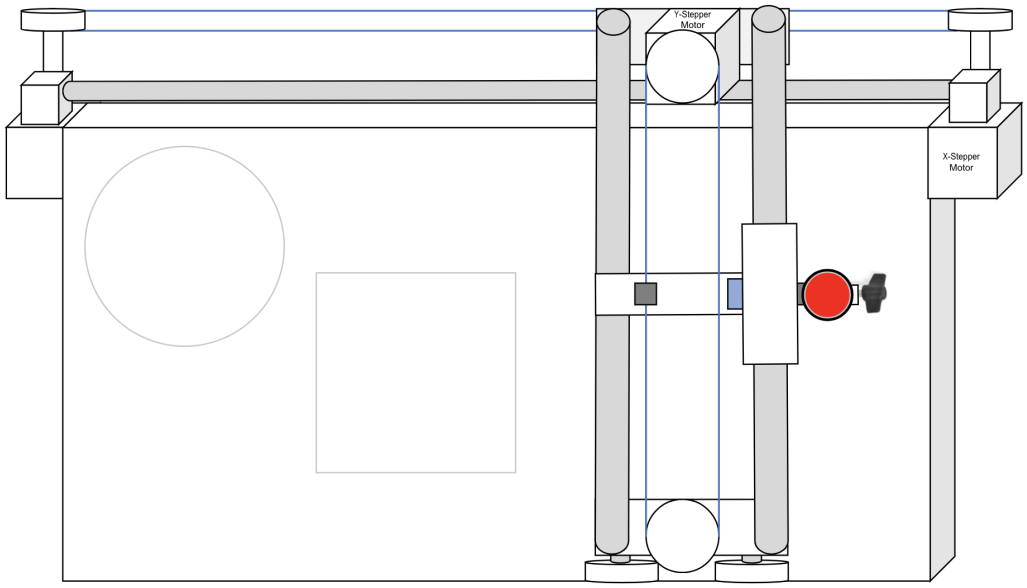

Figure 54: Whiteboard Drawing Apparatus

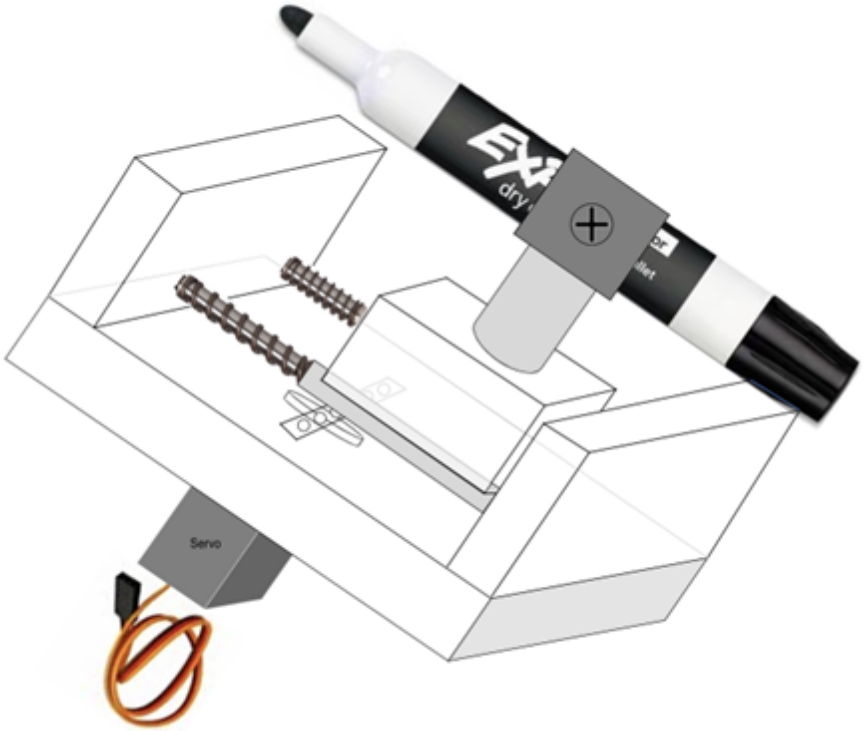

Figure 55: Detailed Servo/Marker Configuration

TEAM INFORMATION

Drew Adams, Electrical Engineer

Responsibilities: Project Leader - Power Analysis and Physical Design

Stefan Ilic, Electrical Engineer

Responsibilities: Hardware Manager - Hardware and Motor Components

James Medved, Computer Engineer

Responsibilities: Software Manager - Smartphone Application and Bluetooth

Vaughn Richards, Computer Engineer

Responsibilities: Engineering Data Manager - On-Board Controls / Firmware

PARTS LISTS

Tables 17 and 18 provide the parts list for the device (corresponding to the schematics shown in Figure 7) and the materials budget list corresponding to the project expenses, respectively.

Table 17: Parts List Corresponding to Schematics

Quantity	Reference Designators	Part Number	Description
2	U1, U4	DRV8825	Motor Driver
2	AOUT1, AOUT2, BOUT1, BOUT2	1-17HS15-1504S-X1	Stepper Motor
2	U3, U5	74HC14N	Schmitt Trigger
2	U6, U7	UA741CN	Op-Amp
2	K1, K2	EC2-5NU	5V Relay
1	U8	RN4870	Bluetooth Module
1	U2	PIC24FJ128GA010	PIC24 Board

Table 18: Materials Budget List Corresponding to Project Expenses

Quantity	Part Number	Description	Unit Cost	Total Cost
1	JF0027	ON/OFF Switch Pack	5.99	5.99
1	10nf-103	Ceramic Capacitor Pack	7.99	7.99
1	EL-CP-021	PCB Pack	11.99	11.99
5	296-28915-5-ND	Motor Driver	6.86	34.30
1	BJ-1OHM-1M OHM	Resistor Pack	10.99	10.99
1	B08L5ZL87P	Capacitor Pack	19.99	19.99
3	1-17HS15-1504S-X1	Nema 17 Stepper Motor	11.99	35.97
1	BITPULLEY-20T-3-2	Idler Pulley	8.49	8.49
1	B07CDDPCM8	Timing Belt	14.66	14.66

1	ZE006	Isolation Columns	9.99	9.99
1	EA611	Limit Switches	9.99	9.99
1	ZR-200527001	Timing Belt	10.99	10.99
1	B01018DB2E	Zip Ties	5.49	5.49
1	B01NCE3ZW1	DRV8825	14.99	14.99
3	B07R1VWY5P	Drag Chain	9.99	29.97
1	EA611	Limit Switches	9.99	9.99
1	4336304932	12V Power Supply	12.99	12.99
1	500212-US1	Belt and Pulley Wheels	11.99	11.99
1	CO-2-VLXC-2020-1500-HEI	2020 59 inch Linear Rails	65.99	65.99
1	CO-2-TLXC-2040-T2-1500-HEI	2040 59 inch Linear Rails	123.99	123.99
1	N/A	SS Bolts, Nuts, & Washer Kit	23.99	23.99
1	HELIFOUNER-T-Nut-240	M3, M4, M5 T Nuts Pack (240 PCS)	9.98	9.98
1	BITPULLEY-20T-3-2	Idler Pulley	8.49	8.49
1	EX235	Plastic Pulley Wheels	10.99	10.99
4	PIC24FJ128GA010-I/PF	Microcontroller	6.93	27.72
8	1119	Switch Buttons	2.50	20.00
1	AMC1602AR-B-Y6WFDY	LCD Screen	10.50	10.50
1	N/A	Thumb Screws	13.99	13.99
1	N/A	Expo Markers	11.87	11.87
			Total:	594.27

.

PROJECT SCHEDULES

A Gantt Chart for the creation of the report and project implementation has been created and is pictured below in Figure 50. Note that this covers the Fall 2022 semester only.

CONCLUSIONS AND RECOMMENDATIONS

In conclusion, our whiteboard drawing assistant tool provides a user friendly, low cost, and fun alternative to more expensive SMART boards. Though our design (similar to that of a 3D-Printer) requires some mechanical engineering experience, it seems very achievable to implement due to the research we have completed and the information we can gather from various online sources. Also, with our backgrounds in computer and electrical engineering, powering, connecting, and interfacing between the hardware and software should be manageable within the current scope of our project. **(JM)**

	Task Mode	Task Name	Duration	Start	Finish	Predecessors	Resource Names
1		SDP I 2022					
2		**Project Design**	**95 days**	**Wed 8/24/22**	**Sun 11/27/22**		
3		**Midterm Report**	46 days	Wed 8/24/22	Sun 10/9/22		
4		Cover page	46 days	Wed 8/24/22	Sun 10/9/22		Drew Adams
5		T of C, L of T, L of F	46 days	Wed 8/24/22	Sun 10/9/22		Stefan Ilic
6		**Problem Statement**	**46 days**	**Wed 8/24/22**	**Sun 10/9/22**		
7		Need	46 days	Wed 8/24/22	Sun 10/9/22		Vaughn Richards
8		Objective	46 days	Wed 8/24/22	Sun 10/9/22		James Medved
9		Background	46 days	Wed 8/24/22	Sun 10/9/22		Drew Adams
10		Marketing Requirements	46 days	Wed 8/24/22	Sun 10/9/22		Stefan Ilic
11		Engineering Requirements Specification	46 days	Wed 8/24/22	Sun 10/9/22		Vaughn Richards
12		**Engineering Analysis**	**46 days**	**Wed 8/24/22**	**Sun 10/9/22**		
13		Circuits (Battery Power)	46 days	Wed 8/24/22	Sun 10/9/22		Drew Adams
14		Electronics (Stepper Motors)	46 days	Wed 8/24/22	Sun 10/9/22		Stefan Ilic
15		Signal Processing (bluetooth to motors)	46 days	Wed 8/24/22	Sun 10/9/22		Vaughn Richards
16		Communications (application and bluetooth)	46 days	Wed 8/24/22	Sun 10/9/22		James Medved
17		Electromechanics	46 days	Wed 8/24/22	Sun 10/9/22		Stefan Ilic
18		Computer Networks	46 days	Wed 8/24/22	Sun 10/9/22		James Medved
19		Embedded Systems	46 days	Wed 8/24/22	Sun 10/9/22		Vaughn Richards
20		Controls	46 days	Wed 8/24/22	Sun 10/9/22		Drew Adams
21		**Accepted Technical Design**	**46 days**	**Wed 8/24/22**	**Sun 10/9/22**		
22		**Hardware Design: Phase 1**	46 days	Wed 8/24/22	Sun 10/9/22		
23		Hardware Block Diagrams Levels 0 thru N (w/ FR ta	46 days	Wed 8/24/22	Sun 10/9/22		Vaughn Richards
24		**Software Design: Phase 1**	46 days	Wed 8/24/22	Sun 10/9/22		
25		Software Behavior Models Levels 0 thru N (w/FR ta	46 days	Wed 8/24/22	Sun 10/9/22		Vaughn Richards
26		**Mechanical Sketch**	46 days	Wed 8/24/22	Sun 10/9/22		Vaughn Richards
27		**Team information**	46 days	Wed 8/24/22	**Sun 10/9/22**		
28		**Project Schedules**	**46 days**	**Wed 8/24/22**	**Sun 10/9/22**		
29		Midterm Design Gantt Chart	19 days	Wed 8/24/22	Mon 9/12/22		Drew Adams,James Medved,Stefan Ilic,Vaughn Richards
30		**References**	46 days	Wed 8/24/22	Sun 10/9/22		
31		**Midterm Parts Request Form**	50 days	Wed 8/24/22	Thu 10/13/22		Drew Adams
32		Midterm presentation file submission	33 days	Wed 8/24/22	Mon 9/26/22		Drew Adams
33		**Midterm Design Presentations Day 1**	0 days	Wed 9/28/22	Wed 9/28/22		Drew Adams,James Medved,Stefan Ilic,Vaughn Richards
34		**Midterm Design Presentations Day 2**	0 days	Wed 10/5/22	Wed 10/5/22		
35		**Project Poster**	14 days	Tue 10/11/22	Tue 10/25/22		Vaughn Richards
36		**Final Design Report**	47 days	Tue 10/11/22	Sun 11/27/22	3	Drew Adams,James Medved,Stefan Ilic,Vaughn Richards
37		**Abstract**	47 days	Tue 10/11/22	Sun 11/27/22	3	James Medved
38		**Hardware Design: Phase 2**	47 days	Tue 10/11/22	Sun 11/27/22	**3**	
39		Modules 1...n	47 days	Tue 10/11/22	Sun 11/27/22	**3**	
40		Simulations	47 days	Tue 10/11/22	Sun 11/27/22	3	Drew Adams,Stefan Ilic
41		Schematics	47 days	Tue 10/11/22	Sun 11/27/22	3	Drew Adams,Stefan Ilic
42		**Software Design: Phase 2**	47 days	Tue 10/11/22	Sun 11/27/22		
43		**Modules 1...n**	47 days	Tue 10/11/22	Sun 11/27/22		
44		Code (working subsystems)	47 days	Tue 10/11/22	Sun 11/27/22	3	James Medved
45		System integration Behavior Models	47 days	Tue 10/11/22	Sun 11/27/22	3	Vaughn Richards
46		**Parts Lists**	47 days	Tue 10/11/22	Sun 11/27/22		
47		Parts list(s) for Schematics	47 days	Tue 10/11/22	Sun 11/27/22	3	Stefan Ilic
48		Materials Budget list	47 days	Tue 10/11/22	Sun 11/27/22	3	Drew Adams
49		**Proposed Implementation Gantt Chart**	47 days	Tue 10/11/22	Sun 11/27/22	3	
50		**Conclusions and Recommendations**	47 days	Tue 10/11/22	Sun 11/27/22	3	Stefan Ilic
51		Parts Request Form for Subsystems	32 days	Wed 9/21/22	Sun 10/23/22	33	Drew Adams
52		Subsystems Demonstrations Day 1	0 days	Wed 11/9/22	Wed 11/9/22		Drew Adams,James Medved,Stefan Ilic,Vaughn Richards
53		Subsystems Demonstrations Day 2	0 days	Wed 11/16/22	Wed 11/16/22		
54		Parts Request Form for Spring Semester	0 days	Fri 12/2/22	Fri 12/2/22	36	Drew Adams

Figure 56: Design Gantt Chart (Final)

REFERENCES

[1] Husmann, Polly R., and Valerie Dean O'Loughlin. “Another Nail in the Coffin for Learning Styles? Disparities among Undergraduate Anatomy Students' Study Strategies, Class Performance, and Reported VARK Learning Styles.” Anatomical Sciences Education, vol. 12, no. 1, 1 Jan. 2019, pp. 6–19. ERIC, EBSCOhost, https://login.ezproxy.uakron.edu:2443/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=eric&AN=EJ1202374&site=eds-live. Accessed 12 Mar. 2022.

[2] Rapp, Whitney H. “Avoiding Math Taboos: Effective Math Strategies for Visual-Spatial Learners.” TEACHING Exceptional Children Plus, vol. 6, no. 2, 1 Dec. 2009. ERIC, EBSCOhost, https://login.ezproxy.uakron.edu:2443/login?url=https://search.ebscohost.com/ login.aspx?direct=true&db=eric&AN=EJ875427&site=eds-live. Accessed 12 Mar. 2022.

[3] AKTAS, Sinan, and Abdullah AYDIN. “The Effect of the Smart Board Usage in Science and Technology Lessons.” Eurasian Journal of Educational Research, 15 July 2016, https://dergipark.org.tr/en/pub/ejer/issue/24398/258631.

[4] Chow, Paul. “Teacher's Attitudes towards Technology in the Classroom.” TSpace, 6 May 2015, https://tspace.library.utoronto.ca/handle/1807/68680.

[5] Baker, Kalena. “How to Harness the Power of Whiteboards.” Teaching Made Practical, 18 June 2021, https://teachingmadepractical.com/using-whiteboards-in-the-classroom/.

[6] Holland, Charles, et al. “IBoardbot. the Internet Controlled Whiteboard Robot > Jjrobots.” Jjrobots, 21 Oct. 2021, https://www.jjrobots.com/product/iboardbot-the-internet-controlled-whiteboard-robot/.

[7] Chavda, A. C., Ilivicky, I. I., & Soboyejo, W. S. (2016). Self-erasing chalkboard (WO2017116995A1). World Intellectual Property Organization. https://patents.google.com/patent/WO2017116995A1/en?oq=chalkboard+drawing#patentCitations

[8] Xiuping, W. X. (2012). Blackboard circle drawing instrument (CN202782353U). China National Intellectual Property Administration. https://patents.google.com/patent/CN202782353U/en?oq=circle+drawing+tool

[9] *The Complete Guide to Stepper Motors*. RS. (n.d.). https://uk.rs-online.com/web/generalDisplay.html?id=ideas-and-advice%2Fstepper-motors-guide#:~:text=The%20stepper%20motor%20converts%20a,increment%20of%20a%20full%20turn.

[10] *How servo motors work*. Kollmorgen. (2022, September 28). https://www.kollmorgen.com/en-us/blogs/how-servo-motors-work/#:~:text=How%20does%20a%20servo%20motor,device%20to%20close%20the%20loop.

[11] Lavaa, A. (2021, September 1). *Types of servo motors and their working principles*. Linquip. https://www.linquip.com/blog/servo-motor-types/

[12] Microchip, "RN4870/71 Bluetooth Low Energy Module", https://ww1.microchip.com/downloads/aemDocuments/documents/WSG/ProductDocuments/DataSheets/RN4870-71-Bluetooth-Low-Energy-Module-DS50002489.pdf. Accessed 10/10/12022

[13] Microchip. "RN4780." Microchip.com, https://www.microchip.com/en-us/product/rn4870. Accessed 10/10/12022

[14] George. "Choosing a Stepper Motor Power Supply: Circuit Specialists." *Simply Smarter Circuitry Blog*, 12 Feb. 2021, https://www.circuitspecialists.com/blog/stepper-motor-power-supplies/#:~:text=Each%20motor%20needs%20%C2%BD%20amp,needed%20current%20is%202%20amps.

[15] *Department of Health*. Emergency Medical Services and Trauma Systems. (n.d.). https://www.health.ny.gov/professionals/ems/