ARM (Actuation via Real-Time Myoelectric Signals) ProsthEEsis

Final Report

EE Senior Design 2024-25, University of Notre Dame

K Blouch, Cassidy Chappuis, Owen Nettles, Madeline Prugh, Andrew Sovinski

Table of Contents

1 Introduction

e-NABLE is a global network of volunteers committed to combating the medical device inaccessibility gap by providing functional 3D-printed prosthetic devices at no charge to users. Over the last decade, this community has democratized access to low-cost, scalable, and customizable prosthetic devices through the development of open-source, 3D-printed prosthesis designs. The University of Notre Dame chapter of this network, e-NABLE ND, has laid a strong foundation for this work here on campus, focusing mainly on manufacturing scaled versions of existing, mechanical prosthesis designs and creating devices specific to certain users or tasks.

The ARM ProsthEEsis project seeks to build on e-NABLE ND's past projects by addressing specific challenges in the design and desired functionality of these devices, pushing beyond existing designs through the implementation of advanced features such as myoelectric control. In addition to myoelectric integration, we intend to improve the overall actuation design to better accommodate typical functionalities while minimizing the weight and cost of the device. The ultimate goal of the project is to provide e-NABLE ND, and potentially the greater e-NABLE community, with a myoelectric prosthesis design which can be replicated and adapted to different users with a maximum degree of functionality at a minimal production cost.

Problem Statement:

For children in search of prosthetic devices, the current market presents several challenges. Traditional prostheses are costly and require frequent replacement as children grow. e-NABLE ND, an organization on campus dedicated to creating affordable, 3D-printed prosthetics, provides a solution to this problem. However, their designs are predominantly mechanical, which limits functionality for users who require more advanced, ergonomic solutions. These devices rely on body-powered mechanisms, which can require substantial effort to actuate. For example, elbow-actuated designs require the user to bend their arm to create a fist, limiting their ability to reach or grasp objects naturally. Additionally, the effort needed to keep a fist closed can lead to discomfort and fatigue, making daily tasks cumbersome and reducing the overall functionality of the prosthetic.

Solution Overview:

A promising alternative is the development of a myoelectric prosthesis that uses electromyography (EMG) sensors to detect muscle signals, interpret them, and use these to control the hand position.
The input uses single-use adhesive electrodes that are commonly found in clinics for EKG and EMG studies.This sensing method allows for the ability to use the response of a single muscle, which can be chosen to be independent of elbow flexion. This allows more intuitive functionality with less effort for the user, in addition to a level of

customization based on the chosen muscle. Electrodes are attached to the user on either side of the center of the designated control muscle parallel to the muscle fibers. A reference electrode will be placed on a nearby bony area that should have little to no electrical signaling; this is essentially electrical ground. The exposed snaps on the outside of the electrodes allow snap wire connection to the processing center.

A combination of hardware and software, including multiple filtering steps, are utilized to distill the signal and allow for clear interpretation of whether the user has flexed their muscle or not. We selected the ESP32-S3 for our project due to the fact that it has multiple GPIO pins and has processing characteristics conducive to signal processing. A calibration function was developed for the processor to confirm threshold values when the device is powered on. This ensures that the device properly responds to user input despite changes in the user's muscles over time. An OLED display is used to quickly and efficiently walk the user through this process.

Lastly, for the output, the microcontroller interfaces with a DC geared motor. A DC geared motor, as opposed to servo motors that have been used in the past, provides higher torque even at lower speeds due to gear reduction and allows for continuous rotation, which was necessary for our design as the stringing of the hand required significant torque to be pulled. A motor driver is used so that optimal speed and different directions (opening and closing) can be executed. The hand was 3-D printed using the EIH, and was a greatly modified design from what e-NABLE ND typically uses due to the unique nature of having a PCB, battery mount, motor mount, spool, and eye bolt needing to be held securely and compactly.

Overall, this device is more accessible and more intuitively operated than existing prostheses. The device minimizes user fatigue in holding the hand closed when the motor is holding it in the closed position.This device is battery-powered, and throughout an entire semester of continuous testing, the batteries never needed recharging, another indicator of how practical our solution is.

Project Success:

Overall, the project was very successful, especially from an electrical design standpoint. Our goal was to integrate EMG sensing and calibration with a microcontroller and motor that could successfully be powered and actuate a modified design of existing 3-D printed hands to provide e-NABLE ND to further their powerful and very-needed mission. Our team was successfully able to present a device on demonstration day that achieved our goals: sense muscle signals using electromyography, filter and process the input, calibrate the device via setting threshold values based on the input, and use these signals to prompt a motor to actuate the hand to either the open or closed position

upon muscle flexion input. Furthermore, this was all held together and powered in a strong and smartly-designed 3-D printed hand and socket.

From an electrical standpoint, our design met our expectations exceedingly well. The EMG sensors were able to successfully detect signals, and the microprocessor and filters were able to distinguish between them and move the motor in the proper direction at the proper speed when necessary. In the demonstration day room, Stinson-Remick 109, there were abnormally high amounts of noise due to multiple projects, some with high-powered electrical components, running in a very tight, constrained environment. Aside from a few abnormalities, the device still sensed and actuated well. In a room with standard consumer electronics running only, it worked even better.

From a mechanical standpoint, our project was still successful, though with room for improvement. Due to supply chain issues, we were forced to demonstrate our device utilizing our PCB that wasn't our latest revision. We persevered and had a successful demonstration day, but it had an insufficient voltage regulator on it. Thus, we demonstrated using one battery instead of the design-required two. Since the motor was given less voltage than planned, it provided less torque than planned. Additionally, the joints in the hand were stiffer than expected. We worked hard to replace them with softer ones, but it still required a lot of torque to properly pull the string and close the hand. Furthermore, though softer joints made it easier to move the hand, it also made it easier for the hand to "flop around" when in the open position. On demonstration day, we were able to successfully demonstrate to multiple professors the ability for the hand to pick up and hold a can of Dr. Pepper. However, at times, the opening and closing of the fingers wasn't as pronounced as we would have liked.

Using one motor was a careful design decision made to reduce the amount of voltage needed to be provided by batteries and the weight of the device. However, in the future, having two motors or one stronger motor would provide more force in the right places to increase opening and closing power. Additionally, the PCB design provided to e-NABLE ND will have a working voltage regulator and allow for two batteries to be used, increasing torque on the one motor.

Our team has succeeded in this goal and are proud to pass off our designs, code, PCB designs, and hardware component selections to e-NABLE so that they can be emulated and used for good and to improve the quality of life for a limitless amount of people in the process. Ultimately, e-NABLE ND is a club with lots of mechanical engineers and expertise in that area. However, they did not have the electrical engineering talent or manpower to integrate EMG sensing, power, user calibration, motors and actuation, and improved hand and socket design all together, end-to-end. Our team was able to accomplish this objective successfully and will be providing e-NABLE ND with much-needed tools for success.

2 Detailed System Requirements

2.1 Overall System Requirements

The prosthetic device must provide intuitive, reliable, and comfortable control of a prosthetic hand by utilizing electromyographic (EMG) signals captured from the user's residual limb. It must operate continuously for at least a standard workday, ideally for more than eight hours, on a single battery charge. The system shall emphasize user safety, ease of use, accessibility, and long-term durability. All subsystems, including EMG acquisition, power management, motor control, mechanical structure, and user interface, should function cohesively to deliver a seamless user experience.

Component compatibility across mechanical, electrical, and software domains is essential. The device must be lightweight and robust enough for daily use while also allowing easy maintenance and servicing by the user without specialized technical knowledge. Above all, the prosthetic must meet the fundamental goal of restoring functional hand movements to the user in a manner that is natural, dependable, and efficient.

Additionally, the final design along with any and all design decisions, ideas, relevant files, etc. should be well-documented in a manner easy for the non-electrical engineer to understand, so that eNABLE ND can build upon and implement this project design for real users in the future.

2.2 EMG Signal Acquisition & Processing Requirements

The system must be capable of reliably detecting electromyographic (EMG) signals generated by voluntary muscle contractions in the user's residual limb. Surface electrodes shall be positioned to maintain consistent contact with the skin, ensuring signal stability without impeding socket fit or user comfort.

Following acquisition, the raw EMG signal must be processed through a series of hardware and software processing steps. Initially, the hardware must isolate the differential signal of the two input electrodes, and amplify the 0.1-1mV signal to a level suitable for further processing. After amplification, a bandpass filter must be applied to remove unwanted noise outside the typical EMG frequency range, particularly filtering out low-frequency motion artifacts and high-frequency electrical noise. Subsequently, an envelope detector must be used to extract the overall energy of the muscle signal over time, producing a smooth signal suitable for digital analysis.

Upon digitization, the processed signal must be analyzed in real time on the microcontroller using software libraries to implement additional noise filtering. The software must determine the presence or absence of meaningful EMG activity, enabling appropriate control signals to be sent to the mechanical actuation subsystem.

2.3 Power Subsystem Requirements

The prosthetic must be powered in a manner that supports continuous operation for a reasonable amount of time under typical usage conditions. The power system must be built around batteries that are safely removable, allowing users to quickly swap batteries if necessary without requiring specialized tools or technical expertise.

To ensure operational safety and to maximize the longevity of the batteries, appropriate charging and discharging management protocols must be implemented. This includes protections against overcharging, over-discharging, and thermal damage. The power distribution architecture must account for and meet the demands of all integrated subsystems, including the microcontroller, signal processing circuits, motors, and any additional electronics.

Accessibility considerations must ensure that the battery compartment is easily reachable and manageable by the user, allowing for efficient daily maintenance routines.

2.4 Motor Subsystem Requirements

The prosthetic hand must utilize a tendon-based actuation system, relying on strong and durable materials such as high-strength string or fishing line. This material must withstand repeated mechanical stresses involved in opening and closing all five fingers during regular daily use, maintaining consistent performance without fraying, snapping, or excessive stretching.

Motor selection must prioritize performance and reliability. A DC motor will be employed, featuring a spool mechanism connected to its shaft to securely manage the wrapping and unwrapping of the ‘tendons’ during actuation. The motor should reliably turn both directions to close and/or open the hand as the user desires. The motor must be capable of receiving command signals from the microcontroller with minimal latency.

2.5 Hand & Socket Design Requirements

The mechanical design of the hand, forearm, and socket must prioritize functionality, structural integrity, and user comfort. The prosthetic hand must incorporate articulated joints that allow all five fingers to open and close effectively. Internal tunnels or pathways within each finger must be included to properly guide and protect the ‘tendon’ system, ensuring efficient force transmission and minimizing wear.

The arm socket must serve as a secure housing for all major components, including the servo motors, batteries, microcontroller, and any associated printed circuit boards (PCBs). The arrangement of these components within the socket must consider thermal management, mass distribution, and ergonomic comfort.

The battery must be located in a manner that allows easy user access for charging or replacement, without requiring removal of the socket from the limb.

Any controls such as buttons, switches, or sensors that facilitate operation, calibration, or locking functions must be positioned to be easily accessible by the user's intact hand. Additionally, the entire mechanical assembly must be sufficiently lightweight to permit all-day wear without causing fatigue or discomfort.

While the goal is to successfully hold objects which may be useful in day-to-day tasks, the focus of this project is primarily to assist eNABLE ND with the electrical design, applying electrical engineering skill sets which the club has reported difficulty with in the past.

2.6 User Interface Requirements

The user interface must support straightforward and intuitive operation of the prosthetic device. Electrode placement must be carefully designed to ensure that electrodes do not interfere with the connection between the user's residual limb and the socket interior, while simultaneously ensuring strong and stable EMG signal detection. Strategic placement will help to minimize signal noise and maximize the system's detection accuracy. The arm should be able to stay in a position without constant input from the user, minimizing user fatigue.

A calibration process must be incorporated to tailor the device's EMG signal detection thresholds to individual users. Calibration must be initiated either automatically at startup or through a simple user-initiated command. The system must guide the user through a series of clear and simple steps that collect resting and flexing EMG signal data. This collected data must then be used to dynamically adjust the detection thresholds, ensuring optimal sensitivity and reliability throughout daily use.

Additionally, user manuals should be created for both the end-user and eNABLE ND. The end-user manual should include information regarding calibration, EMG sensor placement, battery information regarding recharging or replacement, and design information relevant to non-technical maintenance. The eNABLE user manual should include information relevant to those wishing to build upon the existing design (e.g., make updates to the mechanical housing), assembling the arm, and more technical maintenance procedures.

3 Detailed Project Description

3.1 System Theory of Operation

This section will summarize a high level overview of the system. The overall operation of the system is when the user flexes the targeted muscle, the arm prosthesis will actuate, causing the hand to open or close. Electrodes are placed on the user's muscle, which are connected to a hardware processing circuit on the PCB. The circuit normalizes, filters, and amplifies the signal to optimize it for ADC conversion by the microcontroller. Once the signal reaches the microcontroller, it is further filtered in software. The MCU ultimately outputs a boolean, determining whether the muscle is currently contracted or not. The user is able to calibrate the prosthesis, tuning it to function reliably with their unique electrode placement and level of noise.

Based on the MCU output, the MCU will actuate a motor placed within the 3D arm, which opens or closes the hand. To adequately power the motor, the supplied voltage is split into two lines, 8V for the motor, and 3V3 for the rest of the hardware. The 3D printed arm uses flexible resin for the finger joints, allowing for smoother hand movements. The PCB and motor are contained with the 3D printed forearm, with the batteries placed in an external holder for easier user access.

3.2 System Block Diagram

Figure 1 shows the overall system block diagram, illustrating how the system works from a high-level overview. The diagram shows the system divided into subsystems, as well as the interfaces between the subsystems.

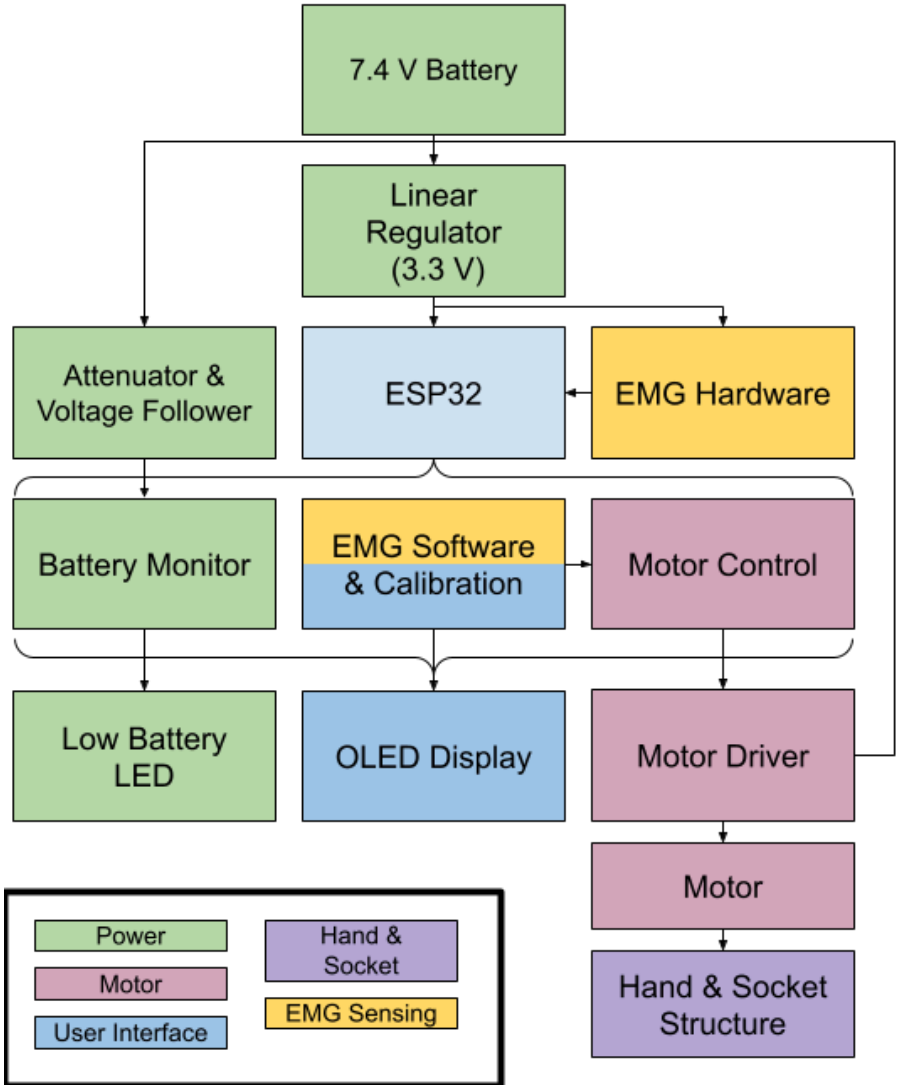

Figure 1. System Block Diagram

3.3 Detailed Operation of Electromyogram (EMG) Sensing Subsystem

Figure 2 shows the detection and processing of the EMG signal, focusing on the key areas of hardware and software processing.

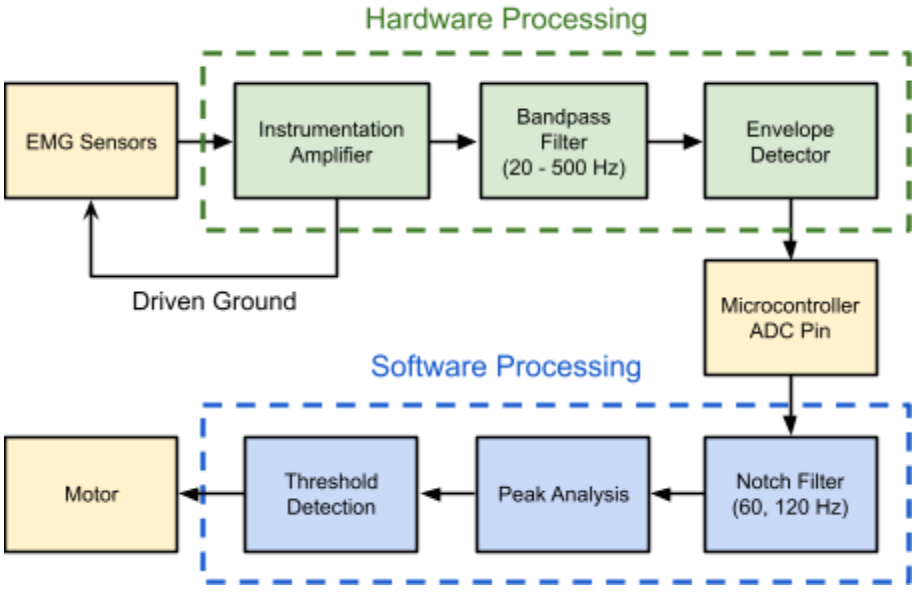

Figure 2. EMG Sensing Subsystem Overview

Requirements

The requirements of the EMG Sensing subsystem are:

1. Acquire an EMG signal from the user's muscle
2. Process the EMG signal in hardware to ensure the input to the ESP32 is at a high resolution and within the specifications of the ADC
    - ➢ Apply filters for noise
    - ➢ Amplify the signal to improve ADC resolution
    - ➢ Smooth the signal for easier ADC sampling
3. Process the EMG signal in software to output a binary decision of whether the muscle is contracted or not
    - ➢ Apply a notch filter to remove 60 Hz noise and harmonics
    - ➢ Implement averaging or signal analysis to determine whether a signal is present
    - ➢ Allow for an adjustable detection threshold for user calibration

Hardware

The usable frequency of EMG signals ranges from 20-500 Hz, so the instrumentation is very sensitive to power line noise at 60 Hz, in addition to electrocardiogram (ECG/EKG) interference around 60 Hz. The energy of the EMG signal is not evenly spread throughout the frequency band, and is instead concentrated towards lower frequencies.

Additionally, EMG signals are at very low amplitudes, a maximum of 0.1-1mV when using skin electrodes. The majority of hardware processing in the EMG Sensing subsystem aims to minimize noise and amplify the desired signal before it reaches the microcontroller. Initial breadboard versions of the subsystem were unreliable and very sensitive to external noise, demonstrating that minimizing wire and trace lengths should be a key factor in component selection and PCB design. **Figure 3** shows the full schematic for the EMG sensing subsystem.

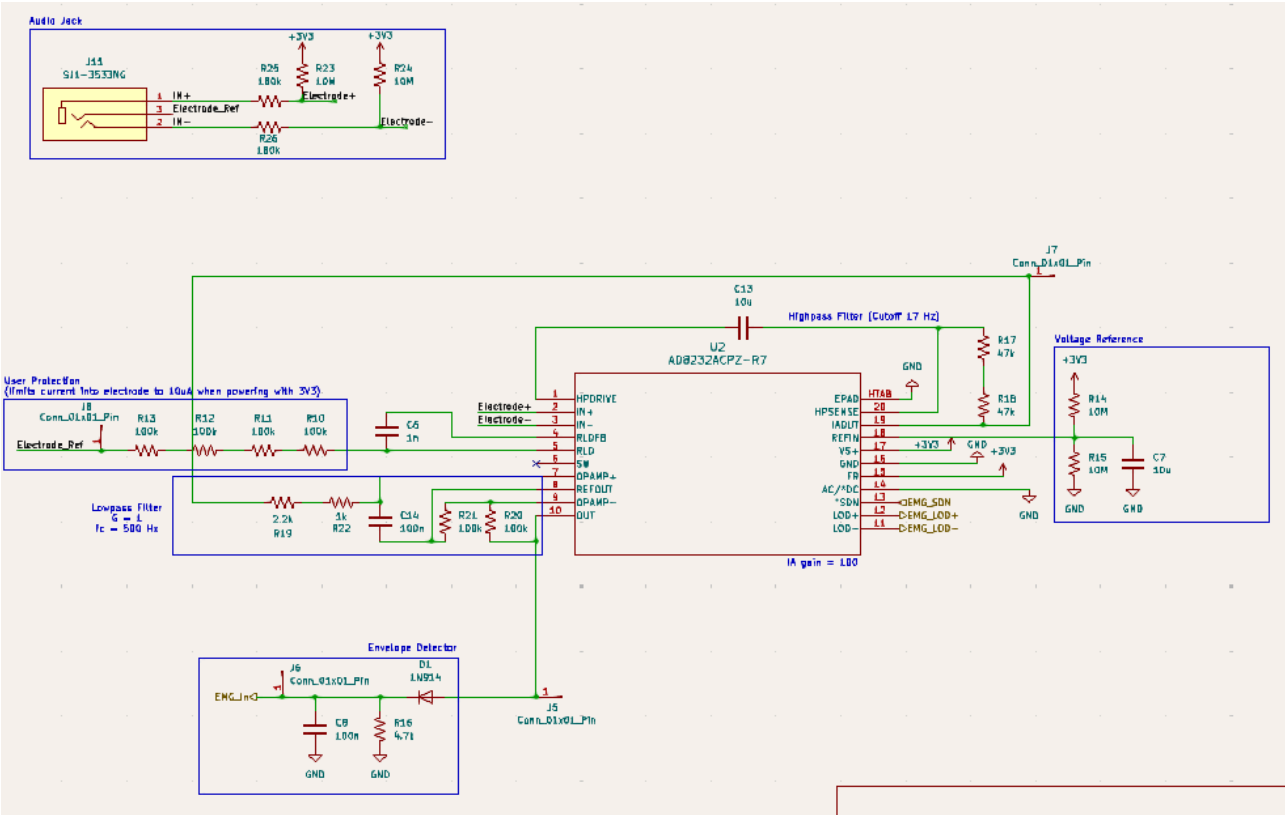

Figure 3. Full Schematic of EMG Sensing Subsystem

The following subsections describe the full hardware design of the EMG Sensing Subsystem. This subsystem utilizes the AD8232, an integrated circuit for heart rate monitoring. The AD8232 was selected because it contains all the necessary signal processing components in a compact, low-cost package. A single chip is only $6, and contains an instrumentation amplifier, driven ground circuit, leads off detection, and extra op-amps to construct a bandpass filter. Though the chip was created for ECG processing, the passive components to set frequency cutoffs are external to the chip, allowing us to tune the processing for EMG frequencies.

a. Electrodes

The standard configuration for detecting biopotential signals is to use a minimum of three electrodes, two for differential signal acquisition, and one as a driven ground. EMG sensing can also be implemented with a two electrode configuration, but the signal will be more noisy and less stable. Electrodes are

relatively standard, with minimal differences between different products. Note that ECG electrodes can also be used for EMG due to the overlap in frequency. This project utilized Covidien 31050522 electrodes, pictured in **Figure 4**, because they were the most available. The electrodes have an adhesive side, with a metal pad coated in an electrolyte jelly in the middle to improve the connection to the user. On the other side, the electrodes have a snap clip.

Figure 4. Covidien 31050522 Electrodes

The two signal electrodes should be placed about 1 cm apart along the direction of the muscle fiber. The ground electrode should be placed on electrically unrelated tissue, preferably close to a bone to minimize noise. The three electrodes are connected via snap clips to a three-wire cable, which connects to the board through a 3-pin audio jack.

At the input terminals of the audio jack, shown in **Figure 5**, there are 180k resistors, recommended by the chip manufacturer to protect the user from fault conditions.

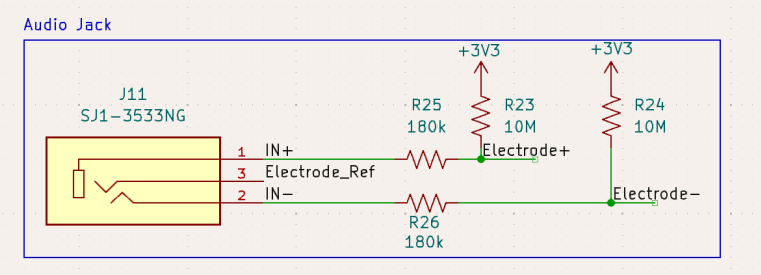

Figure 5. Electrode Input Schematic

The electrode connection to the user has high impedance, making it challenging to adequately ground the user. The input electrodes are biased to 3V3, which helps to ensure the input signal is operating in the same voltage range as the PCB. The 3V3 bias also allows for “leads off detection” where the AD8232 chip detects if the input electrodes are appropriately connected to the user. Pin 11 (LOD-) of the AD8232 will be high when the IN- electrode is disconnected and is connected to GPIO 7 on the MCU. Similarly, Pin 12 (LOD+) of the AD8232 will be high when the IN+ electrode is disconnected and is connected to GPIO 6 on the MCU. The leads off detection feature is configured in the PCB hardware, but is

not currently implemented in the project software as it wasn't a high priority feature.

b. Driven Ground

In a three electrode EMG setup, the third electrode is used to ground the user. However, the ground electrode cannot be directly connected to the ground plane of the PCB. Due to the high impedance of the electrode-user connection, there will be a voltage difference across the ground connection, which will drift over time. If the user is not adequately grounded, the input EMG signal may drift outside the operating range of the instrumentation amplifier within the AD8232, creating an unreliable output. Additionally, directly connecting the user to the board without a protection circuit could be dangerous for the user if any hardware faults occur, as current could travel back through the electrode into the user.

Our design utilizes the third electrode as a driven ground, also referred to as an active ground or a Right-Leg Drive (RLD) circuit. The RLD terminology originates from ECG configurations, where the ground electrode is typically placed on the patient's right leg, which is the furthest body part from the patient's heart. The RLD circuit takes the common-mode signal from the instrumentation amplifier, which is the noise common to both input electrodes, and feeds it back into the user's body through the third electrode, reducing the noise variations within the user.

For user safety, the output of the right-leg drive circuit has a 400k resistor, which limits the maximum current to the user to 8.25uA when the chip is powered by 3V3.

The driven ground circuit also includes a 1 nF capacitor, which forms an integrator when connected between the RLD FB and RLD pins on the AD8232. The capacitor value can be varied to balance gain and noise rejection, but 1 nF is optimal to reject noise in the range of 50-60 Hz.

c. Instrumentation Amplifier

The signal from the two input electrodes are passed through an instrumentation amplifier, which outputs the difference between the two signals. Because the two electrodes are at different distances along the muscle fiber, the EMG signal will be contained within the differential signal, while the common-mode signal is common noise. The instrumentation amplifier is within the AD8232, with the differential signal output on pin IAOUT. It is configured with 100x gain. **Figure 6** depicts the signal at the output of the instrumentation amplifier.

Figure 6. Relaxed vs. Flexed Signal at Output of Instrumentation Amplifier

d. Reference Buffer

The AD8232 allows the configuration of an internal reference buffer, shown in **Figure 7**, enabling all the op-amps to receive a positive and negative voltage input from the supplied system power. As a result, the output of the bandpass filter has a DC bias of half of the supply voltage. The AD8232 datasheet recommends using high resistor values, specifically 10 MΩ, to minimize the power consumption of the reference buffer voltage divider. However, higher resistor values also increase the likelihood of interference at the input of the reference buffer.

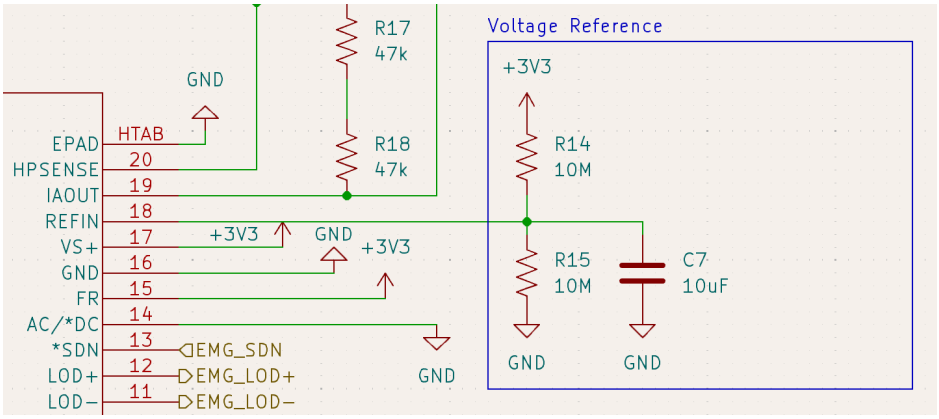

Figure 7. Input to Reference Buffer

To minimize noise, the AD8232 datasheet advises placing the resistors close to each other and the REFIN terminal on the PCB, in addition to adding a capacitor for additional filtering. The proximity of the voltage divider components to the AD8232 is shown in **Figure 8**. A higher capacitor value improves the noise filtering capabilities, but also increases the buffer's settling time after the chip is turned on. The settling time can be estimated using the formula below:

$$t_{settle} = 5 \times \left(\frac{R_{14}R_{15}C_7}{R_{14}+R_{15}}\right)$$

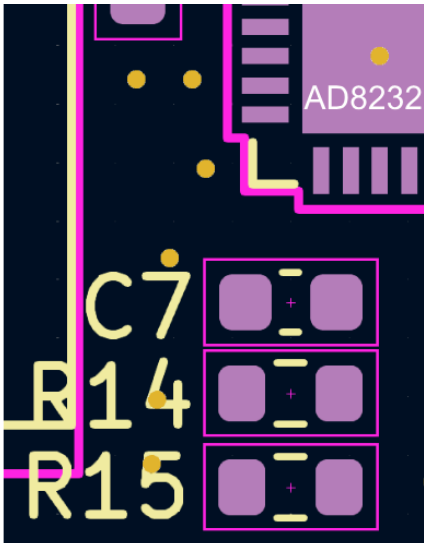

Figure 8. PCB Layout of Reference Buffer Components

e. Bandpass Filter

To filter baseband and higher frequency noise, the design applies a bandpass filter to the differential signal, with a passband range of 17-500 Hz. These frequencies were selected based on the usable frequency range of EMG signals, which is 20-500 Hz. The op-amps used in the bandpass filter are internal to the AD8232.

The cutoff of the highpass filter is set by the following equation:

$$f_{-3dB} = \frac{100}{2\pi RC} Hz$$

The passive components utilized in our design are pictured in **Figure 9**. Note that due to the internal circuitry of the AD8232 chip, the cutoff frequency is affected by the internal 100x gain of the instrumentation amplifier. The AD8232 also allows the implementation of higher order high-pass filters, which were not used in this design. The purpose of the high pass filter is primarily to filter out noise at baseband.

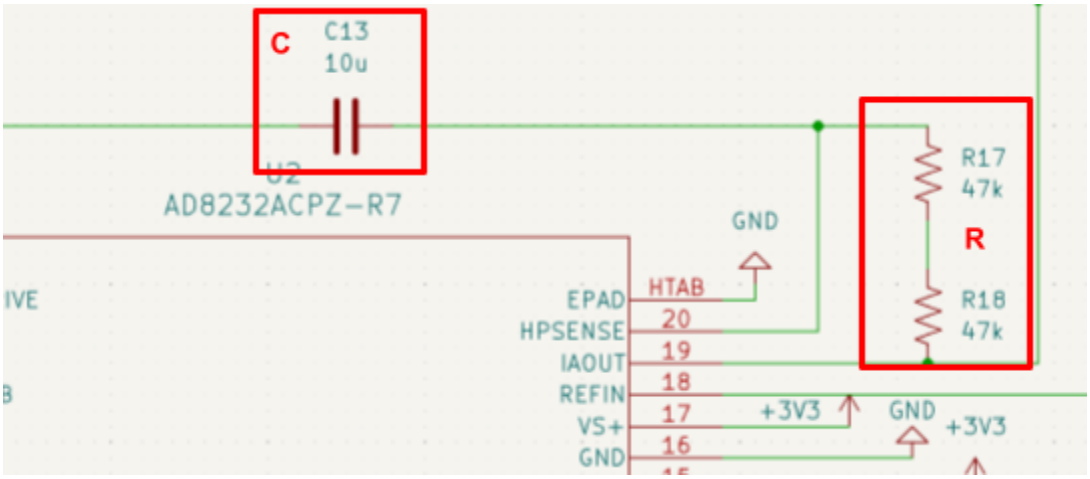

Figure 9. Passive Components of High-Pass Filter

The lowpass filter cutoff is set by the passive components pictured in **Figure 10**, based on the following equation:

$$f_{-3dB} = \frac{1}{2\pi RC} Hz$$

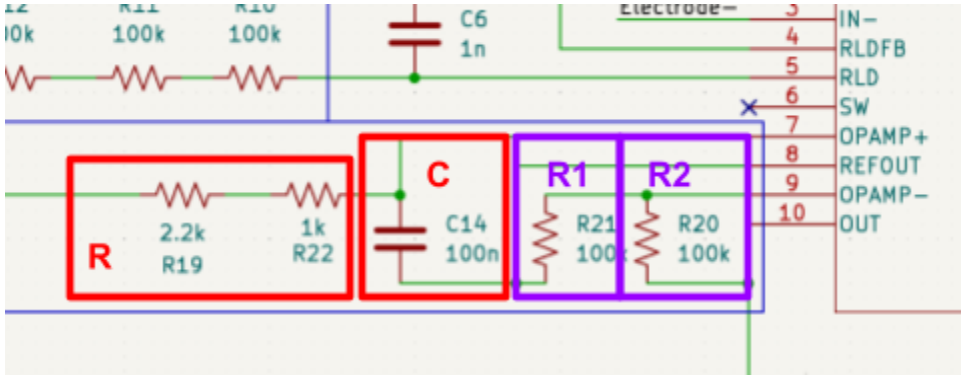

Figure 10. Passive Components in Low-Pass Filter

This filter can also be used to amplify the signal, based on the following equation:

$$G = (1 + \frac{R_2}{R_1})$$

For our design, the internal 100x gain of the instrumentation amplifier provided sufficient ADC resolution for reliable EMG detection. Note that the maximum input signal to the ESP32-S3 cannot exceed the specifications of the ADC pin, which is rated for a voltage range of 0-3V3.

If a higher order filter is deemed necessary for further product improvements, the AD8232 datasheet provides specifications for the design of a Sallen-Key filter, which would provide a sharper roll off. As previously stated, the majority of energy within the EMG band is centered around the lower frequency range, and the primary noise is at 60 Hz, so the high pass filter is primarily for extraneous

noise, and it is not necessary to have a sharp cutoff for high frequencies. **Figure 11** shows the signal at the output of the low-pass filter.

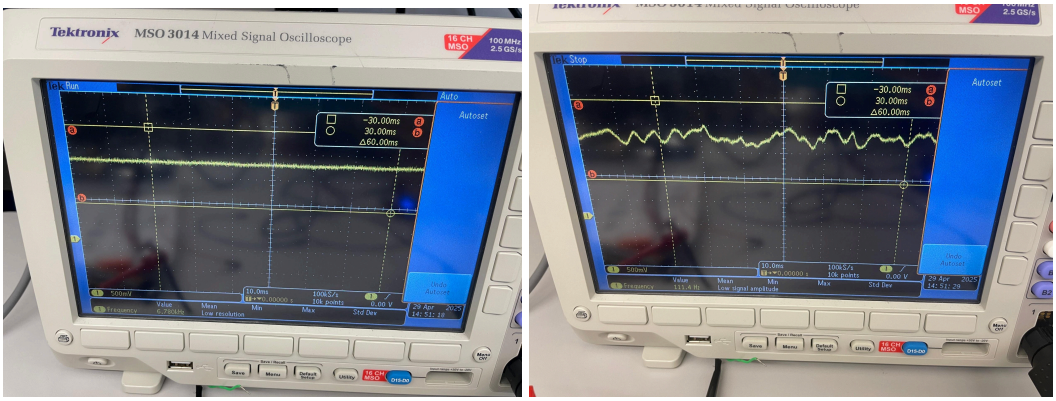

Figure 11. Relaxed vs. Flexed Signal at Output of Low-Pass Filter

f. Envelope Detector

The envelope detector implemented in the PCB design is not fully functioning, but this section describes the ideal behavior of the envelope detector. The envelope detector ideally smooths the output of the bandpass filter, reducing the required sampling rate at the microcontroller. The diode in the envelope detector rectifies the signal, filtering out any negative elements to ensure the signal stays within the range of the ESP32 ADC pin, which is 0-3V3. The time constant of the envelope detector is calculated using the following equation:

$$\tau = RC = (100\,nF)(4.7\,k\Omega) = 0.47\,ms$$

The greater the time constant, the smoother the output signal will be. However, increasing the time constant too much could cause the desired signal to be lost because the detector sensitivity is too low.

In the current design, pictured in **Figure 12**, the signal input to the envelope detector has a DC bias and is centered around 1.8V. Additionally, the time constant is not properly tuned to the desired response. The envelope detector in the PCB circuit is acting as a second low pass filter with a cutoff frequency calculated below:

$$f_c = \frac{1}{2\pi RC} = \frac{1}{2\pi(4.7\,k\Omega)(100\,nF)} = 338.63\,Hz$$

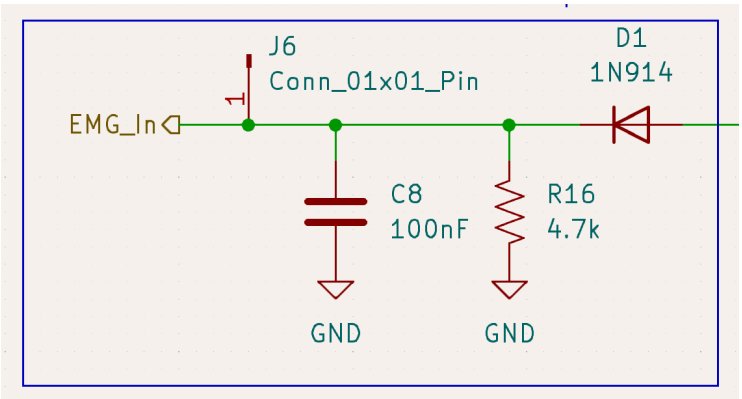

Figure 12. Envelope Detector

Due to the misconfigured envelope detector, some of the higher frequency components of the signal are lost. However, as previously mentioned, the energy of an EMG signal is concentrated to the lower frequencies, so the impact is minimal. The EMG processing software is tuned to function reliably with the existing envelope detector, so no changes were made to the final product. In future revisions, the envelope detector could effectively be removed to save space on the PCB. **Figure 13** shows the signal at the output of the envelope detector, which is the input to the ADC pin of the MCU.

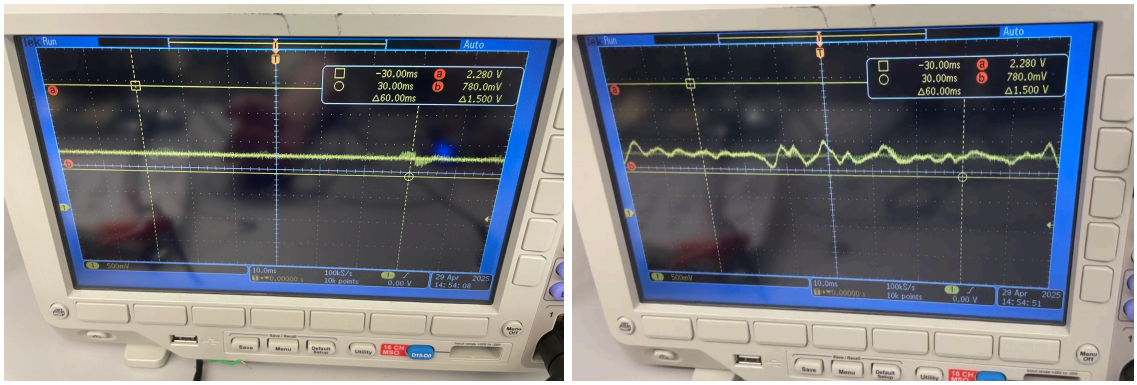

Figure 13. Relaxed vs. Flexed Signal at Output of Envelope Detector

Software

Each sample from the ADC pin of the microcontroller is passed through two notch filters, one at 60 Hz, and the second at the harmonic, 120 Hz. We utilized the Arduino Filters library because it has more extensive documentation and examples, and it is more user-friendly than the ESP-DSP library. The filtered input signal is analyzed in time blocks of 200 ms. The software was initially designed to average the signal within each time increment, but due to the DC bias on the input signal, the “negative” peaks are not filtered by the envelope detector. As a result, when muscle is contracted, the input signal has higher maximums and lower minimums, which average out to the same as a non-contracted signal input. Instead of averaging the time increment, the software analyzes the maximum signal amplitudes within each 200 ms block, and compares it to a threshold set by the User Interface software. The flow chart in **Figure 14** describes the EMG Sensing software.

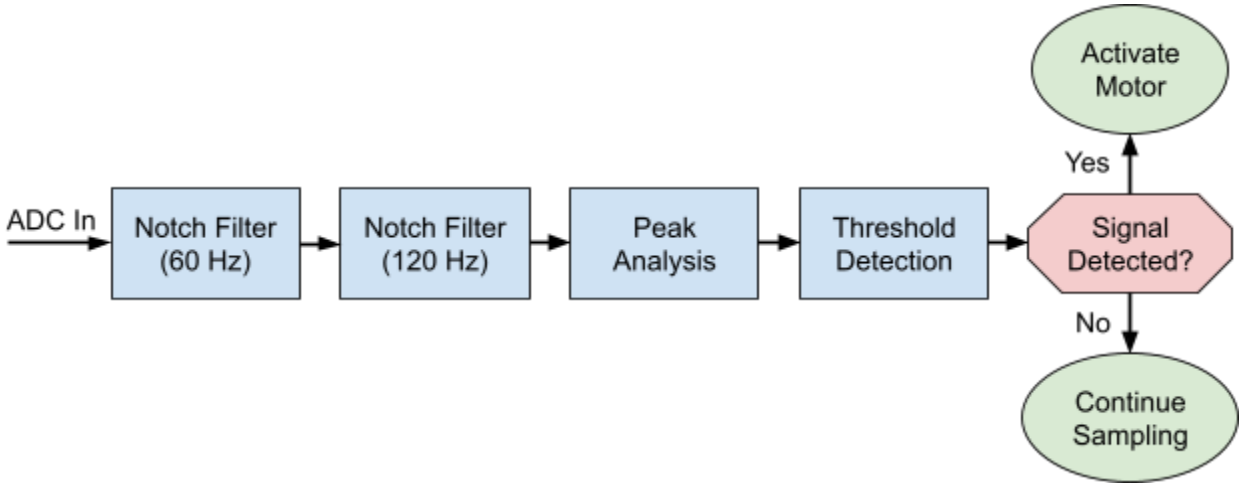

Figure 14. Software Flowchart for EMG Sensing Subsystem

The software was tested by printing the digitized value of the ADC signal, and observing how the value changed under different conditions. When implementing the filter, both the pre and post filter values were printed, and we observed a greater difference between the flexed and relaxed signals in the filtered values. The overall software analysis was tested in a variety of rooms to confirm reliability under different noise levels, in addition to testing on multiple different muscles and users. The calibration software, discussed in the User Interface subsection, was also tested concurrently with the EMG software to ensure the signal detection was flexible enough to accommodate different users, muscles, and noise levels.

The testing demonstrated that peak detection was a reliable determination of whether the muscle is contracted. The difference between the maximum of contracted and non-contracted signals is consistent, and large enough that no false positives or false negatives were observed during software testing.

3.4 Detailed Operation of Motor Subsystem

Subsystem Summary

For the end-to-end functioning of our team's prosthetic device, something needs to be the bridge between the EMG sensing and processing and the actual opening/closing of the 3-D printed hand. That "bridge" is the motor and its related components. When activated by a command signal from the microcontroller after the user flexes their muscle, a motor shaft will turn and change the tension in strings that are attached to the finger joints. Originally, the design called for use of a servo motor due to their commonality and ease of use. However, due to the need for continuous rotation and and increased torque, a DC geared motor was chosen instead. Additionally, a motor driver compatible with our power system and with the ability to move the motor at varying speeds, both clockwise and counterclockwise. The tension in the string either reels it in around a 3-D printed spool, or it releases it allowing it to reel out of the spool. The primary string, fishing line, is connected to a network of strings within the fingers. Applying tension, especially at the right speed, at the right time, and in the right direction, is crucial for the operation of the device.

Requirements

1. The closed-position grip of the hand must be able to hold objects without slippage.
2. Hand and socket must be reasonably lightweight for realistic, daily use, so the motor and related hardware cannot be too heavy or imposing.
3. The motor must consume an amount of voltage that is easily provided by battery power.
4. Bi-directional movement and variable speed of the motor must be possible so that the hand can both open and close and do so at speeds optimized to the physical design of the hand and socket.
5. The motor must provide enough torque so that the string can overcome the stiffness of the joints and actually move the hand.

Hardware

a. **Metal DC Geared Motor with Encoder:**
   Sourced from DFRobot, this motor was selected after extensive searching and comparison with other robots in the market. While a multitude of options exist, finding options that put out larger amounts of torque without being too large or so heavy that they would weigh the entire device down was a challenge. Additionally, finding options that didn't require an excessive amount of voltage while still putting out enough torque providing increased complexity. Rated at 6V,

this motor could easily be powered by two batteries and would still spin and provide some torque at voltages as low as 1V. The reducer reduction ratio is 1:75, a prime consideration as this gear reduction provides much more torque in comparison to a servo motor. Additionally, the stall torque of 6.5kg-cm, while it could be higher, is respectable given the size and voltage requirements. **Figure 15** below shows related force calculations and **Figure 16** shows the dimensions of the motor.

6.5 kg-cm = 0.63743 N-m

F=T/r = 0.63743 N-m/0.004m (radius of spool is 4mm)

F = 159.41 N

Figure 15. Calculation of force that the motor is capable of pulling the string with.

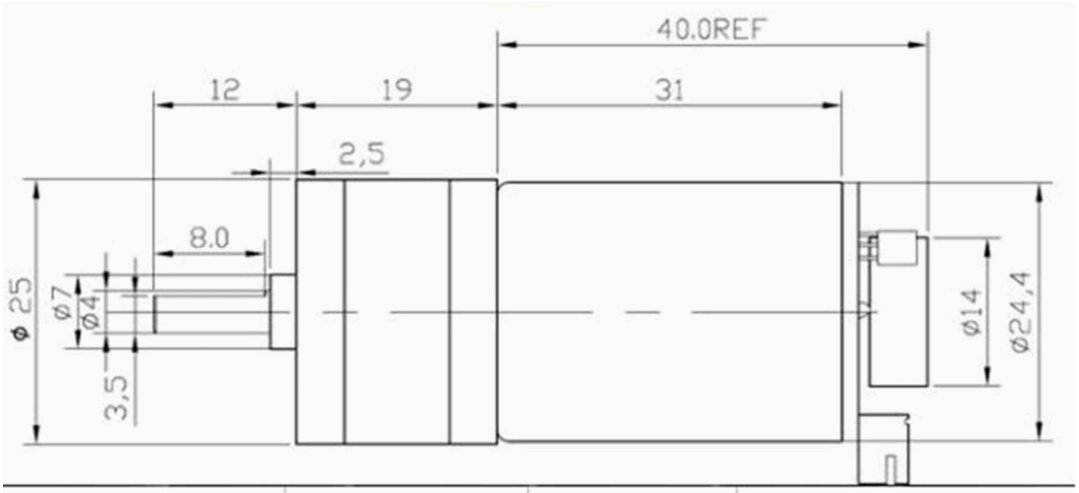

Figure 16. Dimensions of the motor.

Furthermore, while we determined that it wasn't necessary to meet the requirements of our design, the motor has an encoder, which allows for the possibility of feedback should future design improvements require it. Finally, and most importantly, the motor weighs 96 grams, which is about the weight of a tennis ball.

b. Motor Driver

The Toshiba TB6612FNG was selected as it can control the DC motor at a constant current of 1.2 A and in four modes. It utilizes a PWM input signal to

control the speed of the motor. On demonstration day, we utilized a motor driver purchased with male header pins already soldered on. This was done with the intention of having an easier time debugging and troubleshooting issues. However, for the newest board revision that did not arrive on time, the same motor driver is used but integrated right into the PCB. The VM pin is given around 7.4 V (though it was closer to 3.7 V on demo day due to only being able to utilize one battery) and the VCC pin, used for logic control, is simply pulled HIGH at 3.3 V.

Below, **Figure 17** shows the connections made between the ESP32 microcontroller, the motor driver, and the motor on both revisions of our final PCB design.

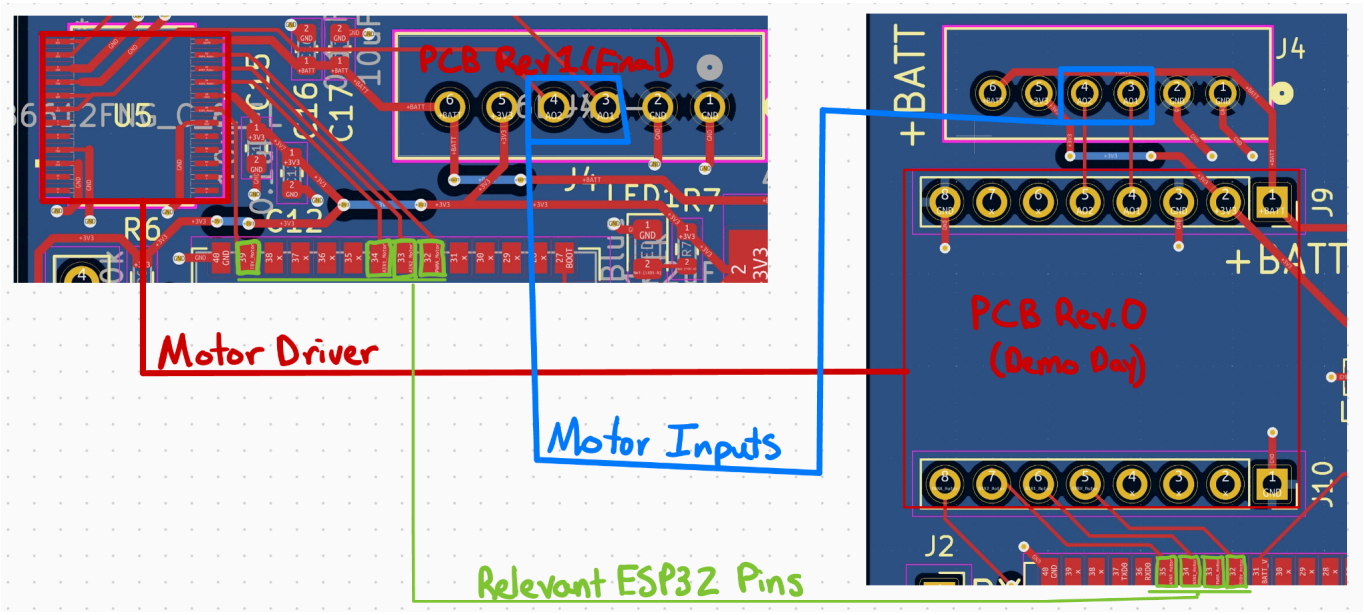

Figure 17. PCB layouts showing relevant motor components and electrical connections between the ESP32 microcontroller, the motor driver, and the motor.

A Note on Demo Day:

On Demo Day, we used PCB Rev. 0 due to supply chain issues. On the ESP32 microcontroller on this board, Pin 35 AIN2_Motor (GPIO 42) was blown out. This was determined after extensive troubleshooting, including analysis of the effect of PSRAM on the different GPIO pins. However, it was determined to be an issue affecting only that one piece of hardware, likely caused by excessive voltage being applied straight to the pin at some point in the testing and development process. Thus, we utilized GPIO 2 for

AIN2 and soldered this connection using a jumper between that pin and the AIN2_Motor pin on the motor driver.

Software

The motor-related software is the key link between the EMG sensing and the actuation of the device. First, four GPIO pins, one for motor standby, one for PWM speed control, and two for motor direction control, are defined as output pins. The standby pin is pulled HIGH to enable motor usage. Two primary functions are utilized in the code. One, entitled runMotor, keeps track of the state of the hand as a boolean variable (either open or closed). It calls the moveMotor function, which takes in two variables: an integer representing speed and a boolean value representing direction. If the hand state is currently closed, it calls the move motor and sends a speed and the direction as false. Inside the function, pin AIN1 is digitally written to the LOW, and pin AIN2 is written HIGH. This causes the motor to spin clockwise and the spool to unravel. The speed is analog written as a PWM signal between 0 and 255. For closing, the AIN1 is written HIGH and AIN2 is written LOW, causing the motor to spin counterclockwise and the spool to reel in. The motor spins in different directions when the voltage difference between the Motor + and Motor - on the motor (connected to AO1 and AO2 on the motor driver, respectively) is either 3.3 V or -3.3V.

Testing with our physical hand and socket was used to determine the best speeds and duration of turning for both opening and closing. With our design, a speed of about 120/255 for 4.0 seconds was best for closing. The slower speed allows for the spool to stay intact and prevents the user from dropping whatever they are holding. However, for opening the hand, a speed of about 180/255 for 5 seconds was optimal. A faster speed allowed for more torque, but wasn't so rapid that the string would snap. The software was designed this way on purpose so that e-NABLE can easily change the speed and opening/closing times to adapt to the exact hand and socket designs that they are using.

Subsystem Testing

The motor was first tested utilizing a kit board and motor testing code prior to being integrated with the rest of the code and with the designed PCB. Ensuring proper wiring was a consistent priority as 11 header pin connections were needed on the motor driver alone for proper operation. Furthermore, throughout the testing and development process, use of the encoder on the motor was considered. However, at no point during the engineering design process was it determined that the encoder would provide an enhancement to capabilities, so it was decided that it would not be used.

One of the largest testing challenges was figuring out why the motor would not turn in both directions, an issue discovered after a Design Review meeting. Around 14 hours was spent using a voltmeter and ammeter at each input/output of the motor, motor driver, and ESP32 microcontroller. Additionally, research was done on the effect of PSRAM, which is used to expand memory, and JTAG (Joint Test Action Group) debugger on different GPIO pins on the ESP32-S3-WROOM-1U. It was determined that these were not negatively affecting GPIO pins, specifically GPIO 42 (Pin 35), and that the motor driver and motor were functioning properly. The conclusion that GPIO 42, which corresponded to AIN2 on the motor driver, was shot, likely from excessive voltage being applied earlier in testing, was reached. However, a jumper wire was soldered to GPIO2 and the code was adjusted in the interim.

The last major part of testing was integration of the motor in the socket of the hand, as well as determining the proper speed and duration for opening and closing the hand. This was extensive and took a lot of trial and error, but ultimately, we were successful in attaching it and the spool so that proper operation of the hand could be achieved. The light weight of the motor was especially important here, as it was able to be anchored down successfully without slippage. On demonstration day, we were able to successfully show Requirement #1, the ability to hold objects without slippage, because of this.

3.5 Detailed Operation of Power Subsystem

Requirements

The requirements for the power subsystem are:

1. Provide necessary power to operate the motor
2. Provide necessary power to operate the signal processing hardware and microcontroller
3. Simultaneously fulfill the previous two requirements, which require different operating points
4. Offer a removable battery for minimal user downtime in the event of full battery discharge
5. Offer a rechargeable battery for user convenience
6. Communicate low battery level to the user
7. Offer protection from overcharging, over-discharging, and short circuits

Hardware

The power system uses two protected 18650 lithium ion batteries in series. The system uses Epoch 18650 2600mAh 8A batteries with built in protection. They are rechargeable via a standard 18650 lithium ion charger and removable. The batteries are 3.7 V, nominally, with a maximum charge voltage of 4.2 V and a minimum safe discharge voltage of 2.5 V. In series, the output is 7.4 V, with a maximum of 8.2 V and discharge cutoff of 5 V. To fulfill the requirement to power the motor, the batteries directly provide power to the motor driver. To power the signal processing hardware and the microcontroller, the voltage is regulated by the stocked AZ1117 voltage regulator. It has a maximum input voltage of 18 V and maximum current of 1 A. There is a large margin between the maximum safe input voltage and the output of the batteries. The maximum current draw that the ESP32-S3 will require is 355 mA while actively using the highest gain RF mode. We only utilize modes with lower power, so there is also a good margin between the maximum safe current of the regulator and the microcontroller. Finally, to monitor the battery charge level, we use a MCP602-E/P operational amplifier as a voltage follower. First, the battery voltage is scaled by a factor of 22 kΩ/69 kΩ via a simple voltage divider. Then, the op-amp follows the reduced voltage, protecting the input of the ESP32 from any spikes in current from the battery. Since the lithium batteries tend to retain their nominal voltage for most of their charge cycle, we illuminate a low power LED when the output voltage is less than or equal to 3.5V. A schematic of the system is below in **Figure 18**.

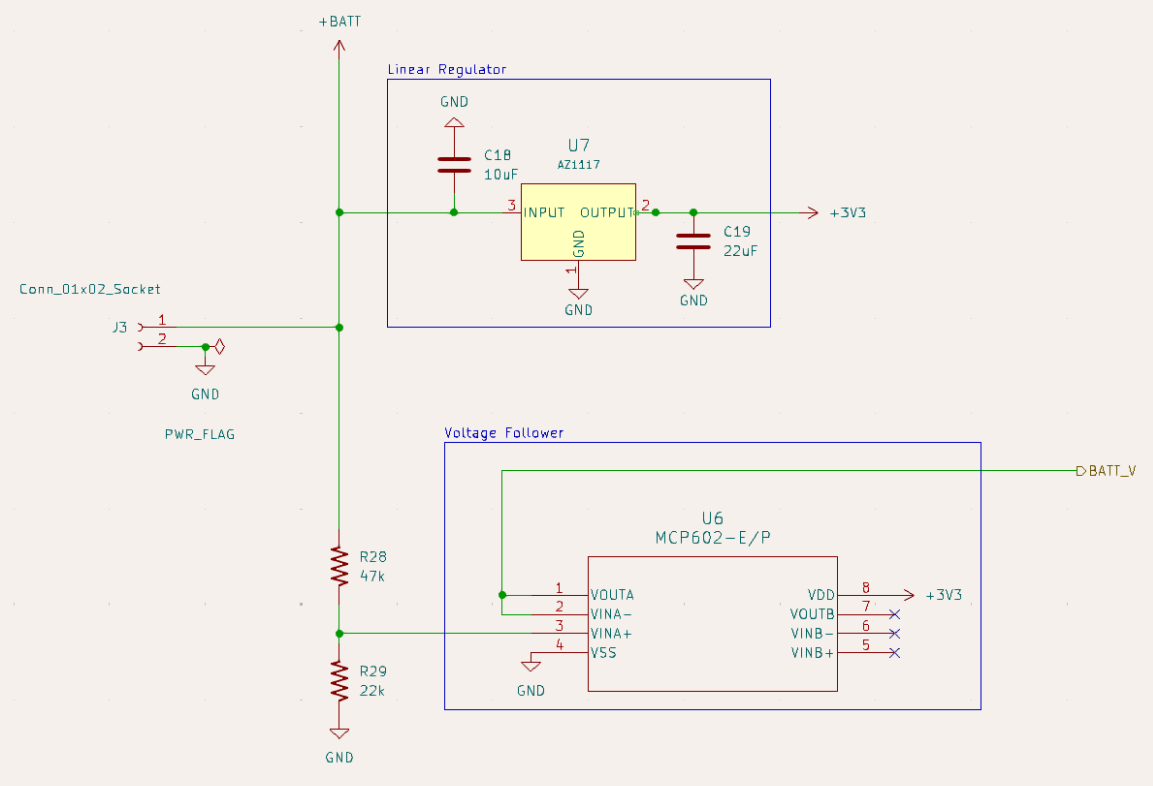

Figure 18. Power board schematic

An additional consideration for the power hardware is the mechanical battery holder. The user needs to be able to access the batteries and the ease of removing them should be reasonable to perform with one hand. Due to the built in protection circuit boards in each battery, they are slightly longer than standard 18650 batteries. To accommodate this, the power system uses a custom made battery holder board that better fits the dimensions of the batteries. The board also features a slide switch to connect and disconnect the batteries from the rest of the system. To connect the external power board to the main board, we use a JST connector. A CAD image of the power board is below in **Figure 19**.

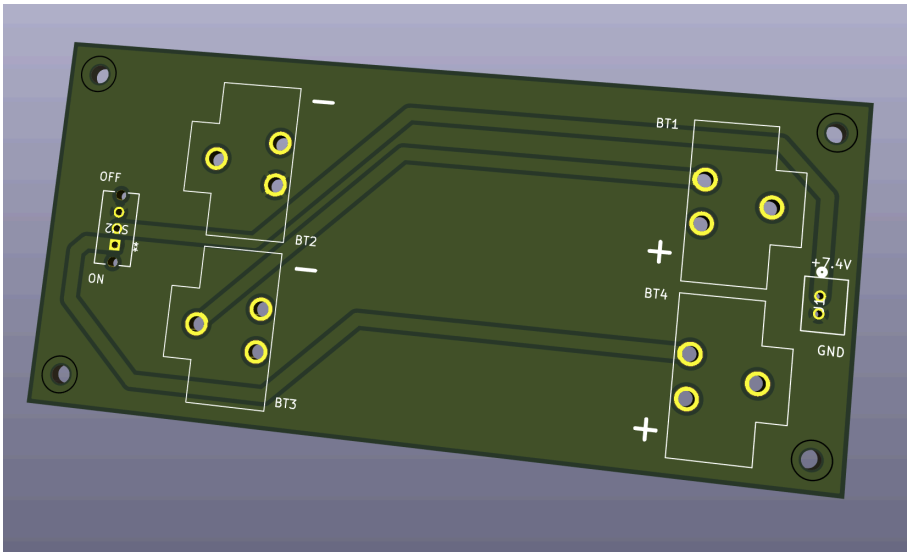

Figure 19. Power Board 3D render

Software

The software component of the power subsystem's main requirement is to monitor the battery voltage. The software needs to notify the user when the charge is low and then shut down the entire arm if the batteries reach their discharge cutoff voltage. To do this, we configure a hardware timer interrupt to set a flag every minute that will trigger a service routine. The service routine will be performed directly after gathering EMG data so that it does not interfere with the reading. During the routine, calling analogRead maps the output of the voltage follower (0-3.3 V) to an integer, x ∈ 0-4095. From this value, we can calculate the battery voltage as follows: $V_{battery} = \frac{22+47}{22} \times 3.3 \times \frac{x}{4095}$ If the battery voltage is below 7 V, a red LED turns on, indicating that charging is needed. Once the battery charge reaches 5 V, the built in protection circuitry in the batteries will disconnect them from the arm.

Subsystem Testing

There are three main tests that need to be performed to ensure that the power subsystem is working. First, we need to ensure that the linear regulator is outputting 3.3V. Second, we need to ensure that the voltage follower tracks its input voltage. Finally, we need to test that the hardware can correctly recognize a low battery voltage event and issue a response to the system.

Since the linear regulator only consists of one part, it was not tested independently and first tested during integration. Our first iteration of the board utilized the AP2112K stock regulator, which could not handle the full charge voltage of the batteries. So, for the first integration test, we relied only on the input from one battery instead of two in series to power the main board. To make up for the higher voltage required for the motor, we utilized an external power source. Moving to the final iteration of the board, we are using

the AZ1117. The two regulators have a nearly identical footprint, so the changes made as a result of the integration test were minimal.

To test the voltage follower, we wired the battery to input across a potentiometer. By sweeping the potentiometer and measuring the input voltage to the voltage follower, we found the following output curve shown in **Figure 20**.

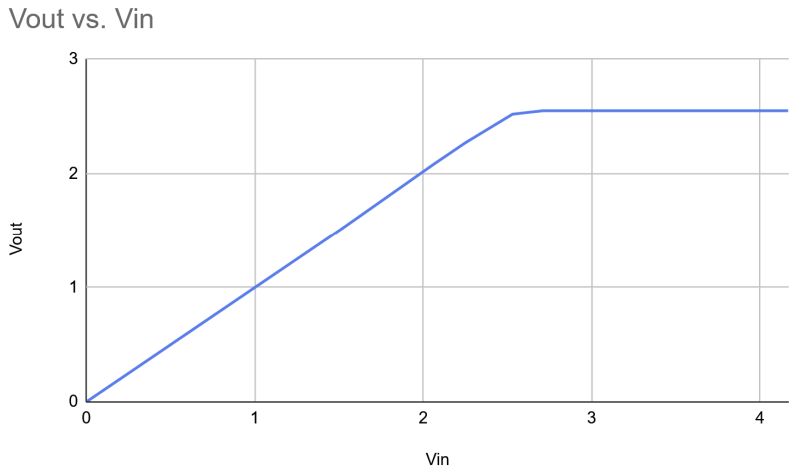

Figure 20. MCP602 op amp curve configured as a voltage follower

The output closely follows the input up to a saturation threshold at 2.55 volts. This test informed the selection of the resistors for the voltage divider. The divider needs to map 8.4 V (maximum battery voltage) to 2.55 V on the voltage follower input. Using two stock SMT resistors, the closest ratio achievable is 22 kΩ to 47 kΩ for a dividing factor of $\frac{22}{22+47} = .319$. This maps the maximum voltage to 2.68 V. This is slightly above the saturation threshold for the voltage follower, but this was an acceptable tradeoff for using only two SMT resistors. The nominal, low battery, and shutoff voltages are all within the linear region of the voltage follower.

The final test is the code. To test that the system can recognize important battery thresholds (7 V and 5 V), we connected the DC power supply across the battery holder terminals and varied the voltage. The system successfully recognized low voltages and turned on the LED.

A Note on Demo Day

Due to delayed shipping of the final board from OSH Park, the demo day product used an older variation of the board that featured the AP2112K linear regulator, only capable of handling a 6 V input. Because of this, we only used one battery. The board also used a green low battery LED as opposed to the red LED intended for the final design.

3.6 Detailed Operation of Hand & Socket Design

Subsystem Summary:

The physical design of the prosthesis should allow for an able-bodied person to demonstrate full functionality on EE Senior Design Demo Day. Additionally, the design should allow for easy adaptability for real-user applications in the future. The main goal of this subsystem is to build upon existing eNABLE designs to provide housing for all necessary electronic hardware and wiring, in addition to means for string-based actuation. User experience should be kept in mind throughout the design process. The final physical design is shown in **Figure 21**.

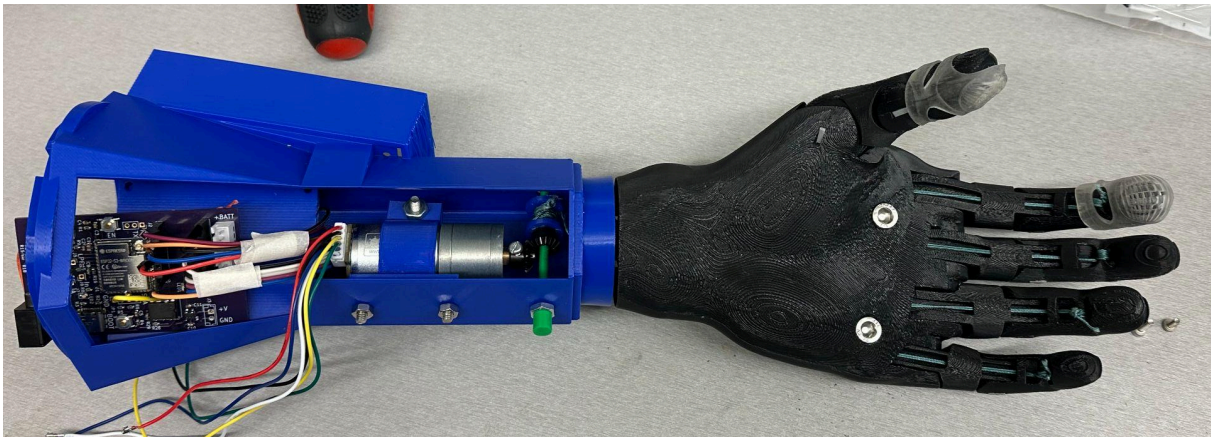

Figure 21. Final Physical Design

Requirements:

The requirements for the physical hand and socket design are:

1. Hand must be designed to include joints which allow for easy opening and closing of all five fingers. Fingers must include hollow tunnels for stringing.
2. Arm socket must include housing for motor(s), batteries, microcontroller and any additional PCB dimensions, as well as any other additional hardware required.
3. Battery location must be accessible by the user for changing or charging batteries.
4. The closed-position grip of the hand must be able to hold objects without slippage.
5. Any buttons, switches, or sensors for locking or calibration purposes must be in a position accessible to the user.
6. Hand and socket must be reasonably lightweight for realistic daily use.

Theory of Design:

The basic theory of operation for most existing eNABLE hand designs utilizes a stringing mechanism within a 3D-printed hand to facilitate the closing of the hand. As shown in Figure 22 below, string is threaded through hollow tracks in the printed fingers, the ends of which are then tied to a whippletree, so that actuation may be performed using a single string.

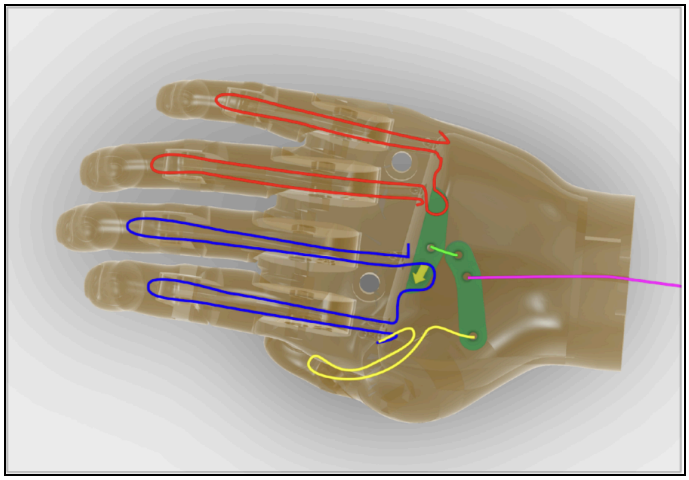

Figure 22. Hand/Stringing Diagram

Since the goal of the project is to leverage electrical engineering expertise to create a design which eNABLE ND can build upon further, we elected to build our hand and arm design off of an existing open-source eNABLE design: the Kwawu 3.0 Socket-Version prosthesis, shown in Figure 23 below.

Figure 23. Image of Overall Hand/Socket Basis Design

This design was selected mainly due to the full-circumference socket, which allows optimal space to add means of housing the electronic parts. eNABLE-ND has used the Kwawu Arm design in recent years, with a team successfully printing and assembling the Kwawu 3.0 within the last year. Thus, due to the familiarity and space for housing/mounting which the design provides, the Kwawu 3.0 was selected as the basis for our physical design.

It should be noted that other basis designs from the eNABLE open-source catalog were considered. The assessment of these designs is summarized in the trade study chart in **Table 1** below. Ultimately, our decision relied primarily on documentation of a precedent of former

device success if used previously by eNABLE ND, arm volume for potential electronics housing, and some aesthetic preference.

Table 1. Basis Design Trade Study

Existing Device:	Surface Area + Modification Potential	Precedent	Joint Flexibility	Aesthetics
(1) Unlimbited	~½ circumference socket still allows some room for additions but more limited than Kwawu	Used by eNABLE ND but not in recent 2-3 years	TBD	More robot-looking
(2) Kwawu	Full-circumference sockets allows max room for added housing / interface features	Used by eNABLE ND frequently and recently	Some issues with joint flexibility when used by enableND	Very human-hand looking
(3) Reborn	~½ circumference socket still allows some room for additions but more limited than Kwawu	No known precedent	TBD	More robot-looking
(4) Po	Available images indicate full-circumference socket?	No known precedent	TBD	More robot-looking
(5) Kwawu 3.0	Full-circumference sockets allows max room for added housing / interface features	Used by eNABLE ND recently with good results	Good joint flexibility/performance (report from enableND)	Very human-hand looking

The Kwawu 3.0 files are designed in the OpenSCAD software for ease of scaling to user fit. We were able to leverage this scaling feature to create a hand of a reasonable adult hand size, which still allows for proper housing of all parts. The OpenSCAD software files allow for input of features such as the width of the hand, desired length of the forearm, etc. The scaling parameters which were chosen are shown in **Figure 24** below.

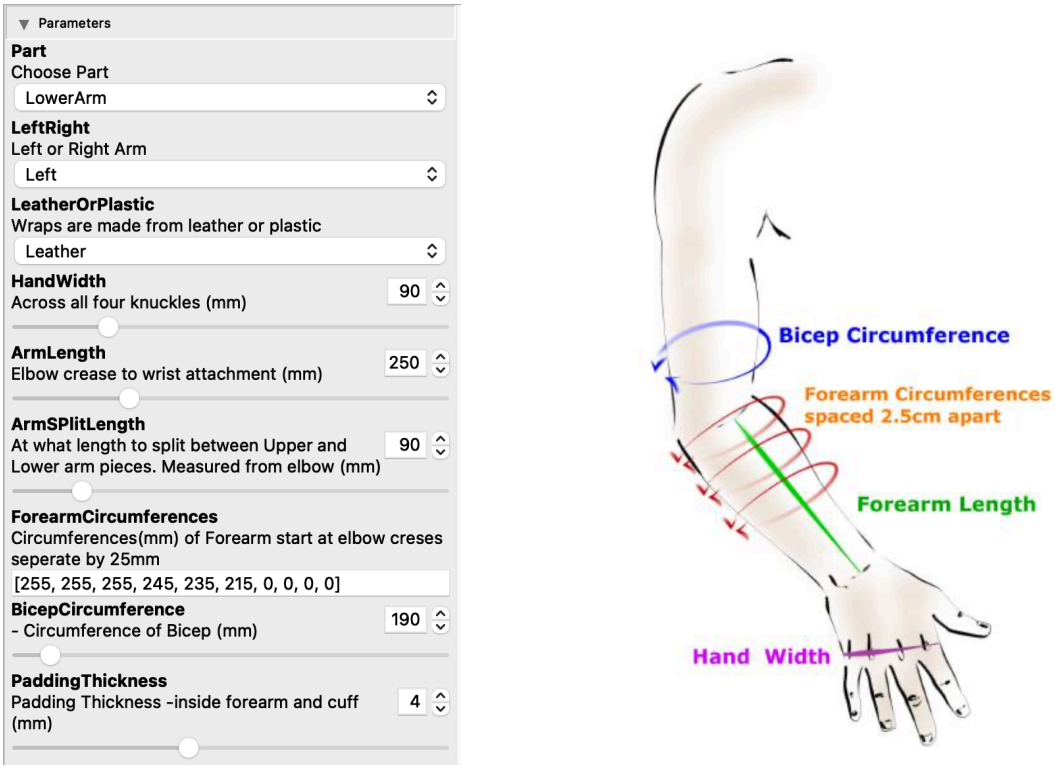

Figure 24. OpenSCAD Scaling Parameters

Making adjustments in the right-hand parameters column will scale all parts of the arm, but the files need to be individually rendered and then exported as .stl files. The LowerArm part is the part subject to further modifications in SOLIDWORKS, as this part is providing housing for the electronics. The remaining parts can be sent straight to print. The exception to this may be the Hinges file, which may need to be slightly scaled up to fit tightly in the finger joint holes of the hand. Scaling can either be done in SOLIDWORKS, or by increasing the HandWidth parameter and then exporting the Hinge file again.

CAD Design - Modifications:

Having scaled the ARM design to a reasonable size in OpenSCAD to accommodate housing and mounting mechanisms for the electronic components, the design of the lower arm needed to be modified in CAD to provide actual housing components. The initial upload of the LowerArm file in SOLIDWORKS, is shown in **Figure 25**.

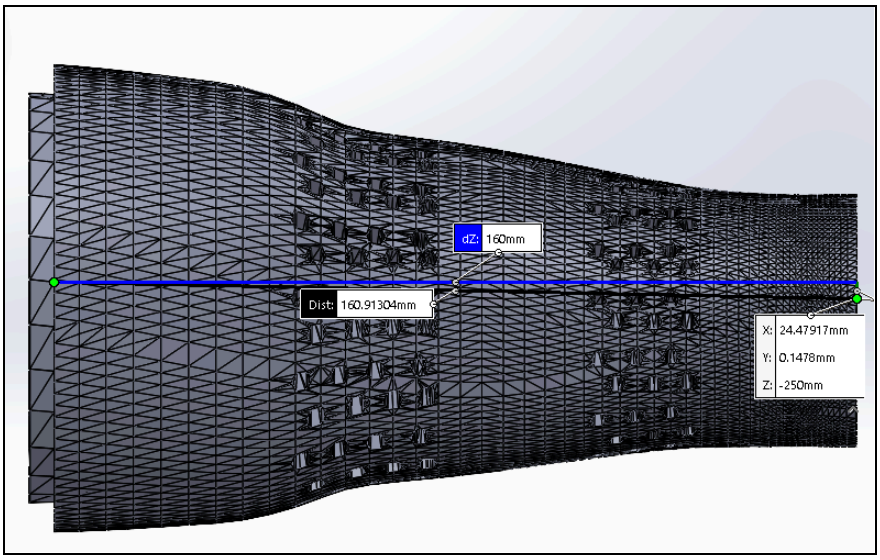

Figure 25. Lower Arm in SOLIDWORKS

While the original lower arm design is meant to mimic the shape of a human forearm, this portion of the arm was ultimately replaced with a hexagonal, geometric-style lower arm. This decision was made in order to improve ease of making modifications, mounting the boards and motor by minimizing curved surfaces within the arm. To do this, the section of the arm between the connection ends was removed. It is essential to preserve at least ~10mm on either end of the original solid body in order to preserve the means of connecting the LowerArm to the UpperArm and Palm. These ends can then be saved as separate solid bodies, and the distance between them can then be adjusted to increase the length available for parts housing. Ultimately, an additional 16mm was added between the ends to make room for the motor and gear mechanism.

In order to create the replacement geometric forearm, we opted to implement a hexagonal shape to abstractly retain the overall round shape of the LowerArm, while also being able to work with substantially tall flat surfaces for mounting. To do this, a hexagon of the same approximate radius of the wrist connection was extruded from the wrist to the LowerArm center. Another extrude feature was then used to add a drafted hexagon from the end of the former section to the socket connection end. The length of the upper section must be long enough and have a diameter wide enough to accommodate the main PCB dimensions. The lower section (closer to the wrist) must be long enough to house the motor, in addition to any string-spooling mechanisms. The general hexagonal shape is shown in **Figure 26**.

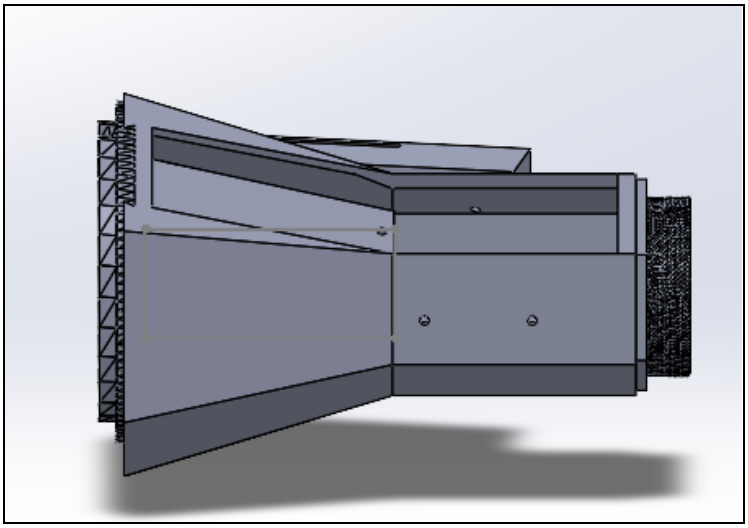

Figure 26. Hexagonal LowerArm

Part Integration:

Figure 27 below shows the overall locations of the electronics within the final LowerArm design. The subsequent sections discuss the design decisions relevant to the integration of each subsystem in the final ARM design.

Figure 27. Electronics Locations in Final Design

a. PCB Mounting

The PCB also needs to be mounted within the arm, in a position where it is secure but in which the OLED on the board can be seen by the user and any relevant buttons included on the board for user interfacing can be pressed.

For purposes of demonstrating the board design on Demo Day, we opted to leave the lower arm with an open section, so that the board layout and motor location is able to be viewed during demonstrations. A diagonal panel was extruded through the upper section of the lower arm, with mounting holes for securing the PCB in place. In the future, a rotating panel may be implemented for closure and safeguarding the electronics from external elements. **Figure 28** offers a clear visualization of this panel location.

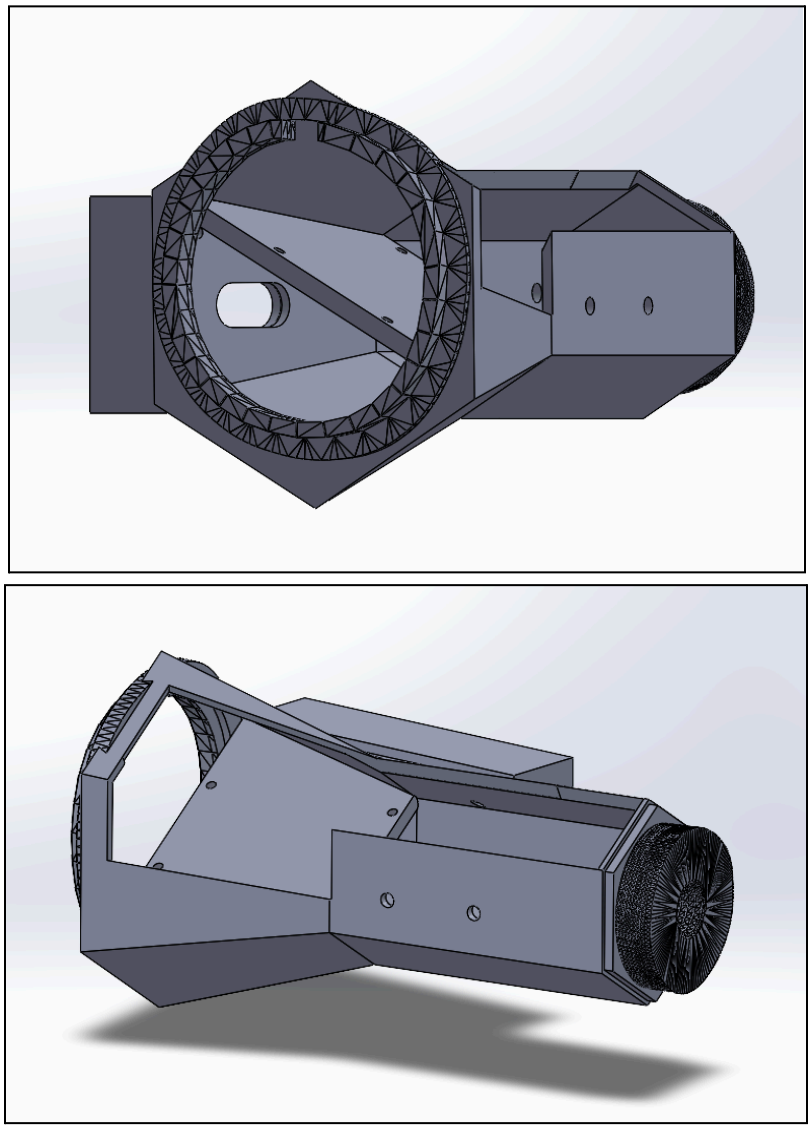

Figure 28. PCB Mounting Panel

b. Battery Housing

The power subsystem also requires means to be housed within the arm. For this purpose, an external box is included on the outside of the lower arm, inside which the power board can be mounted. In SOLIDWORKS, a hole was cut through the base of the power box to the main hollow section of the arm, so that the power cables could be threaded through said hole, and connected to the main PCB without any wiring externally observable. The external box is shown in **Figure 29** and **Figure 30** below.

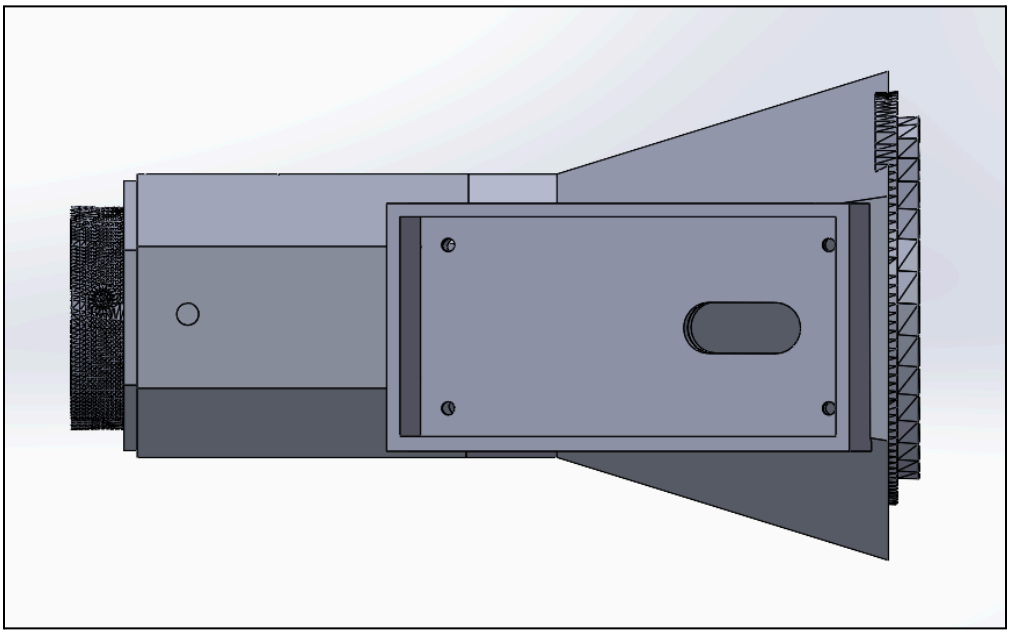

Figure 29. External Box for Power Board

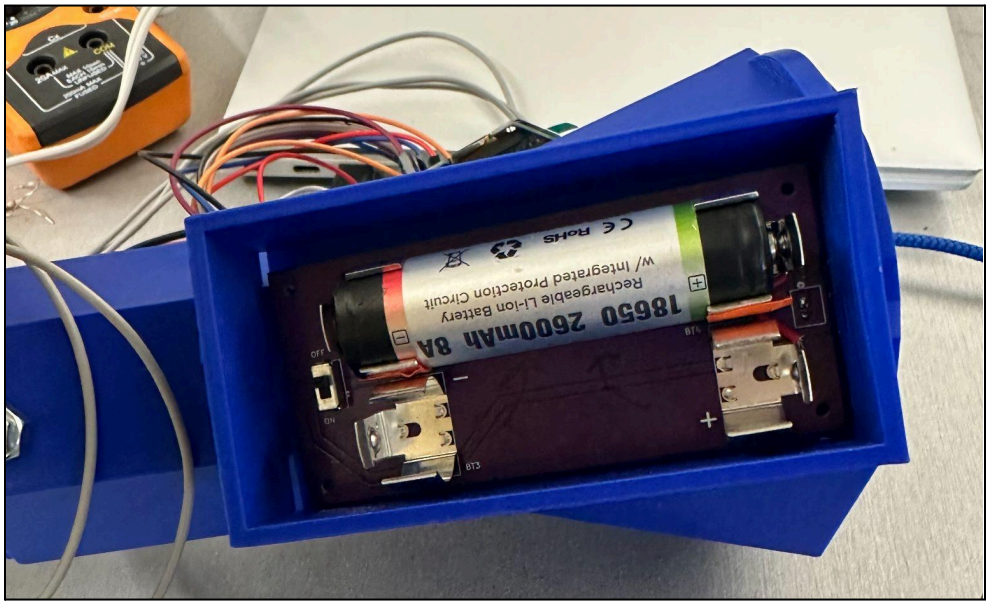

Figure 30. Power Board in External Box

c. Motor Mounting

One challenge of the design was finding a mechanism to mount the motor which would allow the motor to not only fit in the arm, but to also allow for correct tension to be applied to the string from the hand. Ultimately, we opted for a circular clamp which would hold the motor in place and could be mounted with screws on an inner-edge of the hexagonal lower arm piece. This clamp design is shown in **Figures 31 and 32** below.

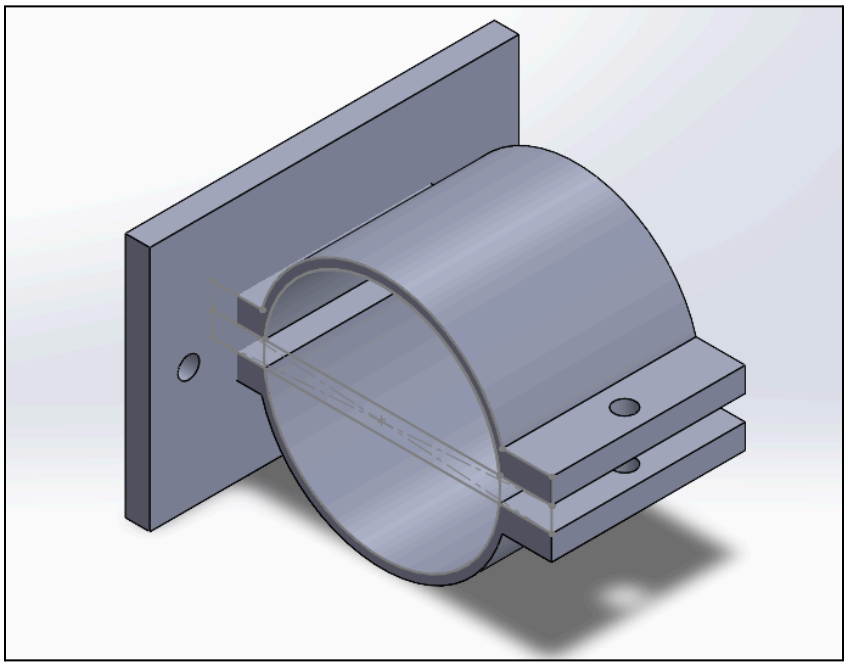

Figure 31. Motor Clamp

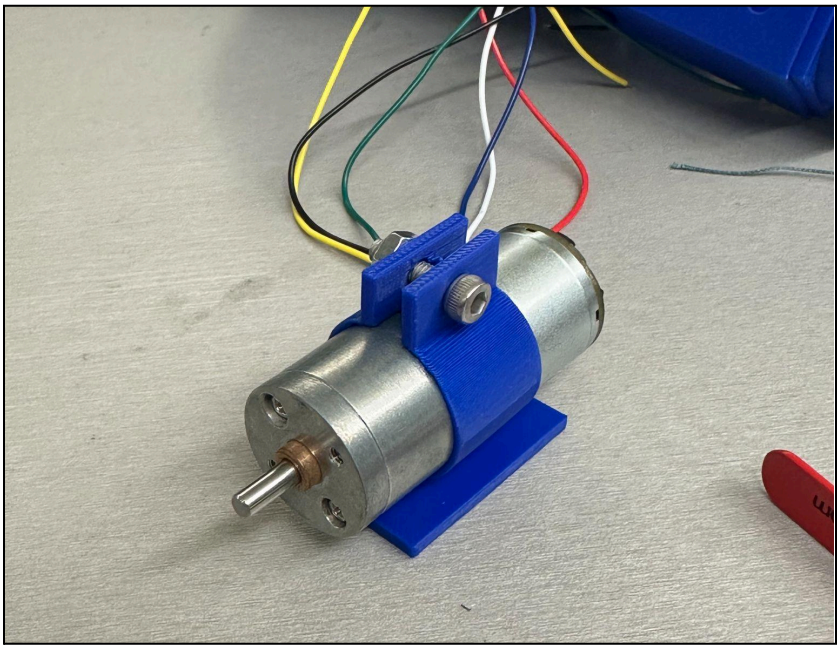

Figure 32. DC Motor in Motor Clamp

d. Stringing Mechanism Integration with Motor

The most challenging part of the physical design was integrating the stringing mechanism of the hand with the motor shaft. This process required iterative designs, as some failed or proved not to be as robust as originally anticipated. Several variables influence the success of the hand-closing mechanism, including the robustness of the knots in the hands, the torque provided by the motor, the flexibility of the material used in the hinges of the fingers, and the technique of translating the rotational motion (roll to pitch).

The initial concept, shown in **Figure 33**, used a bevel gear mechanism to redirect rotation. While functional (see **Figure 34**), the 3D-printed spool lacked mechanical durability under load. The design was revised to incorporate an eye bolt and the repurposed spool (**Figure 35**), allowing reliable string tensioning directly from the motor shaft. This final configuration was mechanically stable and effective in operation

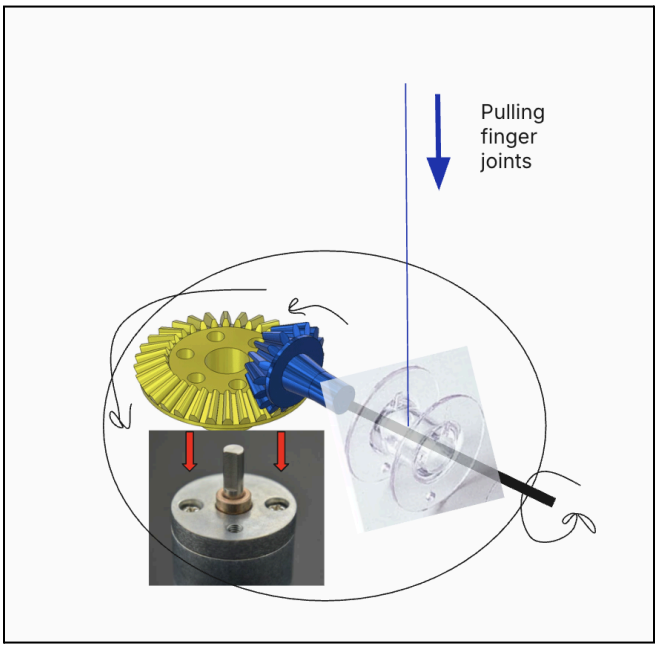

Figure 33. Initial Conceptualization of Spooling Mechanism

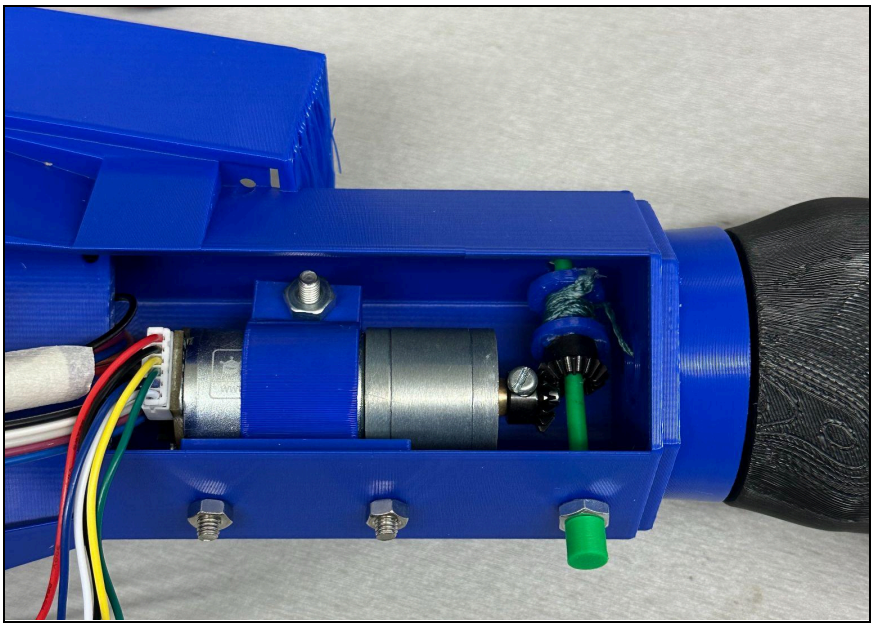

Figure 34. Set-up of Failed Spooling Mechanism

Figure 35. Final Spooling Mechanism

3.7 Detailed operation of User Interface Subsystem

Requirements

The requirements of the User Interface Subsystem are:

- There must be a calibration process so the device can be tuned to the user.
- There must be user manuals for both the end-user and eNABLE members.
    - For the user, this will include information on electrode placement and the calibration process.
    - For eNABLE, this will include a description of the code and information about the board and CAD files.

The calibration process was designed to use a screen to prompt the user on what to do and a button for the user to start calibration/ move to the next step. In the final board version, a larger green button was selected to make it more obvious which button should be pressed for calibration. The boot and enable buttons that are needed for the eNABLE team members are much smaller than the user button.

A screen also needed to be selected. The screen did not need to have multiple colors, and it needed to be compact to fit in the arm. Additionally, many screens have ribbon cable PCB connectors. These seemed quite tedious to work with and difficult to connect successfully, so I decided to select a screen already on a board. To permanently affix the screen board to the main board, the screen board's header pins can be soldered on the main board. Since the functions in Arduino's SSD1306 library are quite intuitively named, I also wanted to use a screen that would be compatible with the library. This led me to select the Hosyond OLED 5 pk, as it fits these requirements, is affordable, and readily available.

Details of the code for the user interface subsystem are detailed in section 3.8: Interfaces: Software Interface and section 5: User Manual/ Installation Manual.

3.8 Interfaces

Software Interface

The majority of code operations are mutually exclusive functions, which are time sensitive, run for a finite duration, and should not be interrupted. When a muscle flexion is sensed, EMG sensing is temporarily suspended to avoid multiple motor activations from the same muscle contraction. Furthermore, the prosthetic hand only has two discrete states: open or closed. The motor should not be interrupted while changing the hand between states because the software does not track partial states. Additionally, while the hand is calibrating, the normal operation (EMG sensing and motor activation) should be suspended because the software is in the process of updating the threshold.

Considering the desired operation of the prosthesis, the software is configured as a superloop with discrete states. The software states are:

- EMG Sensing
- Motor Activation
- Calibration
- Power Tracking

The software is primarily sampling the EMG signal until an event occurs. When an event occurs, the software temporarily pauses EMG processing to handle a different operation. The potential events and their corresponding operations are:

- Threshold detected → Activate motor
- User button pressed → Enter calibration
- Timer flag → Check battery power
    - Battery is low → Activate warning LED

Figure 36 depicts the entire decision tree of the project software.

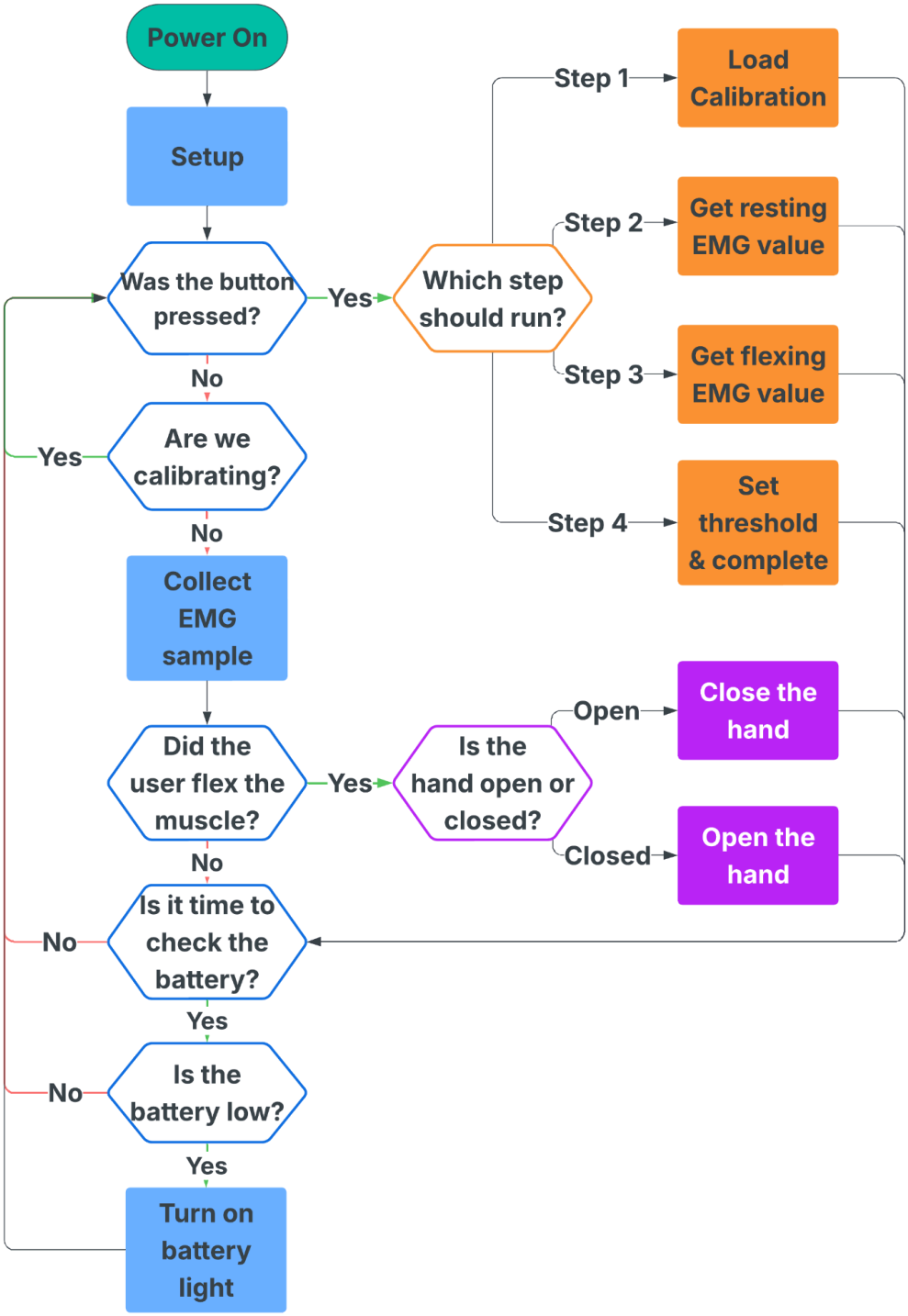

Figure 36. ARM ProsthEEsis Software Operation

PCB Layout

The full PCB layout is depicted in **Figure 37**. The primary design considerations for the PCB layout were:

1. **EMG and Motor Separation:** The motor components were placed on the opposite side of the board from the EMG sensing components to minimize noise in the EMG signal due to the motor sharing a ground with the sensing components.
2. **Motor and Battery Proximity:** Because the motor is the only component powered by the full 8V battery supply, the battery input was placed next to the motor components.
3. **Minimizing Size:** The PCB must fit within the 3D printed forearm, which informed the following decisions:
    a. **Board Dimensions:** The PCB is rectangular to ensure the width is narrow enough to fit within the forearm.
    b. **Audio Jack Orientation:** Rather than facing the outside of the board, the audio jack input is oriented towards the center of the board, allowing the EMG electrodes to be connected without the cable hanging over the side of the board.
    c. **OLED Orientation:** The OLED is oriented to be above the MCU on the board, ensuring it doesn't extrude beyond the board dimensions.

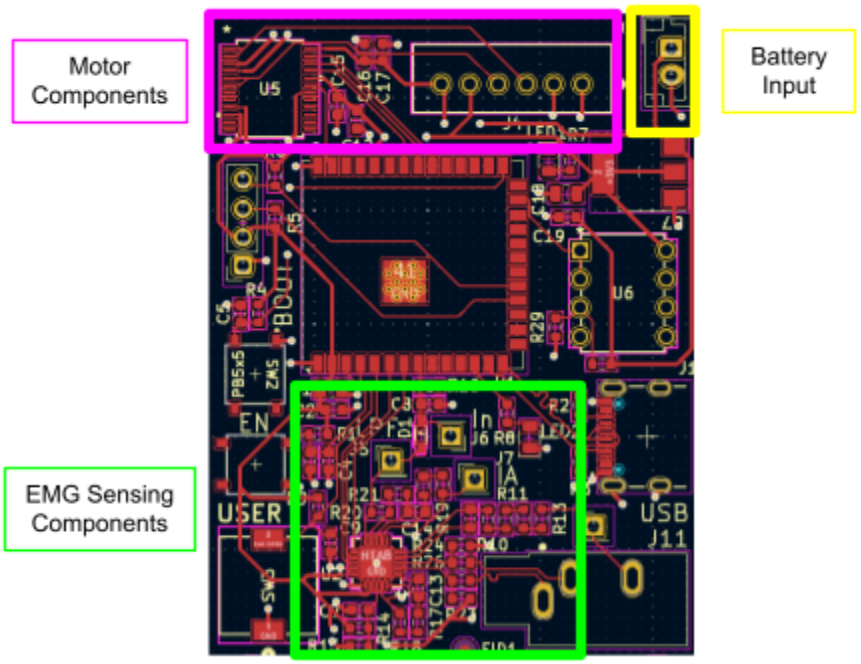

Figure 37. PCB Layout

MCU Selection

We have selected the ESP32-S3 for our project because it has Successive Approximation Register (SAR) ADC pins, which can operate more efficiently with lower

power consumption than the standard ADC pins on the ESP32. In addition to having optimized ADC pins, the Espressif ESP-DSP library is also optimized for the S3. In earlier design revisions, we anticipated utilizing this library for software processing, motivating our MCU choice. The ESP32-S3 meets our other listed requirements, including minimum required input/output pins.

The ESP32-S3 variation selected was the WROOM-1U because the chip comes without the antenna module, saving space on the PCB. The antenna module is unnecessary for this project, as the software does not utilize Bluetooth or WiFi.

Any of the ESP32-S3-WROOM-1U memory variations are sufficient as the software only utilizes 1 MB of flash, 2 MB of RAM, and no PSRAM. Note that the Platformio file in the final software version is configured for the memory partition of the ESP32-S3-WROOM-1U-N16R8, which has 16 MB of flash, and 8 MB of RAM.

4 System Integration Testing

4.1 Describe how the integrated set of subsystems was tested.

The overall set of subsystems was tested in a logical, iterative manner. The first step was transitioning from each using breadboards and kit boards to using the PCB that we designed. First, we confirmed that EMG sensing was accurate, and the OLED display was operational. The next step was iterating the calibration code by using the serial monitor output to view EMG values as the calibration process was configured. Throughout this, power was being tested with the voltmeter to ensure that the right components were being powered correctly. Once all this was confirmed, the motor and its corresponding code were added to the mix to ensure that it could spin in the right direction, at the right speed, and at the right time without too much latency. The final step was integrating and securing each of the systems into the 3-D printed hand and socket. This was done last, as at this point, we had ensured proper code operation and confirmed the integrity of all electrical and power connections and components.

The most difficult part of this integration process was securing the motor and connecting the spool of the fishing line so that it could effectively apply tension in the line and actuate the hand. The original plan called for the use of connecting gears and a plastic 3-D printed rod to connect a spool of line with the top of the motor head. However, this turned out to be too complicated, and the plastic 3-D printed rod snapped because it was not strong enough to withstand the applied torque. However, the team came together and tested a new option, which used a 3-D printed spool glued to the head of the motor and an eye bolt to direct the spool and maintain tension when applied. This worked, and, along with some changes to the joints in the hand to soften them, allowed the project to come together.

4.2 Show how the testing demonstrates that the overall system meets the design requirements

In summary, the overall system requirements called for a device that provides “intuitive, reliable, and comfortable control of a prosthetic hand utilizing electromyographic (EMG) signals” and is safe and easy to use. Additionally, the device needed to be light enough for efficient use and able to be documented for e-NABLE ND to use in the future and apply to real users.

On demonstration day, each of these criteria were demonstrated successfully. The device easily turned on when the power switch was flipped, and the user was easily and seamlessly able to go through the calibration steps and follow the directions provided on the OLED display. When that was complete, the user was able to open and close the hand by simply flexing their muscle. The hand opened and closed several times, and a Dr. Pepper can was picked up. Most importantly, though, all of the design decisions, component decisions and sourcing information, code, PCB files, and CAD files have been compiled together so that e-NABLE ND can easily and efficiently replicate our design for real users.

5 Users Manual/Installation Manual

5.1 User Manual

1.1 How to install your product

The first step for using the prosthesis is placing the electrodes. Two electrodes should be placed near the muscle belly, in line with the muscle fibers. The third electrode acts as a ground, and it should be placed near a bone or electrically unrelated tissue. An example of electrode placement is shown in **Figure 38**.

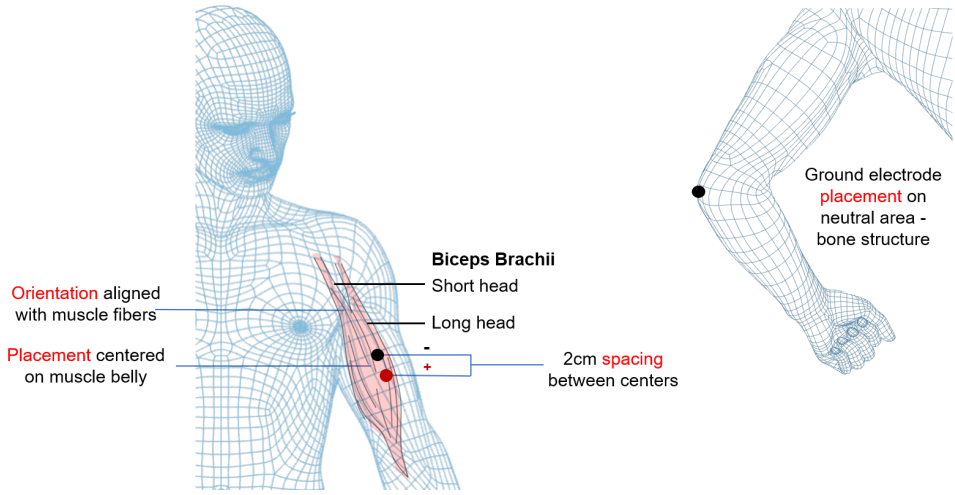

Figure 38. Electrode Placement Guide (source)

Next, the cables need to be snapped onto the electrodes. The green connector should snap onto the ground electrode. The yellow and red connectors should snap onto the muscle belly electrodes.

Then, the arm & socket should be fitted onto the residual limb. Now, the device is ready for setup and subsequent use.

1.2 How to setup your product

Before setting up the device, two 18650 lithium ion batteries with built-in protection should be placed between the snaps in the battery box in the orientation corresponding to the markings.

To turn on the device, move the switch on the battery board to the “ON” position.

Once the power is on, a blue LED should light up on the main board within the arm. If you see this light, you can proceed with calibration.

To begin the calibration process, press the “USER” button on the circuit board. The OLED display will then provide instructions for the calibration process:

- First, the calibration process will load. After loading, you will be prompted to press the “USER” button.
- Then, you will be prompted to relax the target muscle. The device will take measurements during this time. After, you will be prompted to press the “USER” button.
- Next, you will be prompted to flex the target muscle as the screen flashes. There will be a countdown before the flashing starts. When the screen is fully white, flex the target muscle as you will to change the hand position.
    - Note: The screen will flash three times, for varying amounts of time. Continue to flex the muscle until the screen goes dark again.
    - After, you will be prompted to press the “USER” button.
- Two things may happen in the next stage:
    - If the calibration was successful, the display will let you know. Then, it will calculate and set the threshold value. You are now finished with calibration – hold the button to start the calibration process again.
    - If the calibration was not successful, you will get an error message. If this happens, hold the button to start the calibration process again. You will be prompted to do so.

After successful calibration, setup is complete.

1.3 How the user can tell if the product is working

You can tell the device is operating properly when:

- The blue power LED is on.
- The OLED display provides a prompt after holding the “USER” button.
- The calibration process ends in success.

- The movement of the hand corresponds with the input from the target muscle.

1.4 How the user can troubleshoot the product

- What if the blue power LED does not come after I flip the power switch?
    - This is likely due to an issue with the batteries. First, charge them and try again, ensuring proper orientation.
    - If this does not work, try another set of batteries.
- What if the OLED display does not provide a prompt after holding the “USER” button.
    - First, ensure you are pressing the “USER” button: there are two other buttons present on the board.
    - Next, make sure you are holding the button when the hand is not moving. The button will not work during other processes, so if the hand is opening or closing, hold the button for at least 5 seconds. The prompt should appear when the motion stops.
- What if the calibration process has an error?
    - First, try the calibration process one more time. There may have been temporary signal interruptions that interfered with the signals detected in calibration.
    - If this process fails again, check the metal cage on the inside of the device. This cage helps minimize the noise due to nearby electronic devices. If it is broken, wireless signals will variably interfere with signal processing: it needs to be repaired.
- What if the hand does not move when I flex my muscle (or vice versa)?
    - This is likely due to a calibration issue. The calibration process may need to be repeated throughout the day if the muscle weakens quickly with use. Try repeating the calibration process.
    - Next, check the metal cage on the inside of the device. This cage helps minimize the noise due to nearby electronic devices. If it is broken, wireless signals will interfere with signal processing: it needs to be repaired.
- If there is another issue or these solutions do not work, please contact eNABLE. They can further troubleshoot the hardware and software to resolve the problem.

5.2 e-NABLE Manual

First, the relevant code, board, and CAD files are uploaded to the website. Here, you will also find the bill of materials (BOM) and copies of both user manuals.

You will first want to purchase the necessary materials, which are listed in the BOM. Ensure the components are in stock before you order to avoid delays, and order a few weeks before you plan to assemble the device.

Code Files

To adapt the device for each application, you will likely need to tinker with the code. Please refer back to Figure 36 to understand the overall code flow before diving in.

First, download Visual Studio Code and install the PlatformIO package. This website may be helpful for guiding installation.

To open the file, go to the PlatformIO home page in VS Code. Click “Open Folder”, then open the entire ARM_VX.X folder. The key code files are in the “src” folder. All the code is thoroughly commented, so with the following descriptions, it should be easier to understand and change as desired..

Some overarching comments:

- You **must connect the board to batteries in order to upload & test code**! The USB port was not configured to also provide power, so the batteries are needed to troubleshoot.
- .h files are for function declarations (which are required). These allow the multiple .cpp files to be used in main.cpp, making the code easier to read, understand, and manage. .h files are *header files*, and they are called at the beginning of a .cpp file. Header files allow the code to access libraries, which are essentially what the other .cpp files are.
- delay() is often called after display functions: this allows the user time to read the message on the OLED before the program continues.
- Everything that prints to the serial monitor is left in the code to ease the troubleshooting process. These are not vital to the function of the end-user device, and can be commented out before the device is given to the user.

The **calib.cpp** file contains the function definitions for the calibration process.

- **calibBegin** tells the user that the process is loading and prompts them to press the button to proceed.
    - It only consists of display functions from *display.cpp.* The delay gives time for the user to read the screen before changing the display.
- **relax** will collect EMG signals when the muscle is relaxed to determine the average peak value in the resting state.
    - After initializing necessary variables, it prompts the user to relax the target muscle. Then, the for loop will collect values from the sample_200 function in *sample.cpp* and sum them together. Upon exit, it will divide this value by the number of samples. Then, the user will be prompted to press the button to proceed, and the function will return the calculated average.
- **flex** will flash the screen and collect EMG signals when the muscle is flexed to determine the average peak value in the active state.
    - After initializing necessary variables, it prompts the user to flex the target muscle. A countdown will be displayed before the screen clears then flashes white. Then, the for loop will collect values from the sample_200 function and sum them together. Upon exit, it will divide this value by the number of samples to get the average. This process will repeat two more

times, with the new average being added to the existing average. Finally, this sum is divided by 3 to get the flexed average that is returned at the end of the function. Then, the user will be prompted to press the button to proceed.

- **calibEnd** will determine if the calibration process needs to be repeated. If it was successful, a threshold value is set and the calibration process ends. If there was an error, the user is notified to restart calibration, and a random threshold is returned.
    - The return variable is initialized. Then, the if statement corresponds to calibration success: the flexed value should be larger than the relaxed value. The user is notified of the success, then the threshold is calculated as an average of the relaxed value and the flexed value. Then, the user is notified how to recalibrate, the display is cleared, and the threshold is returned.
    - The else statement corresponds to a calibration failure. The user is notified of an error on the display and prompted to restart calibration by holding down the button. A random threshold value is returned.
- **senseEMG** determines if an EMG sample exceeds the threshold.
    - First, the sample is collected. If the sample > threshold, the muscle is flexed, and the function returns true. If sample ≤ threshold, the function returns false.

The **display.cpp** files contain the definitions for the functions that display text on the OLED.

For all our sanity, I will not go through each display function, as they all follow the same architecture (except for the initialization function).

- **init_display** checks to see if the OLED is connected properly. This is a pretty standard initialization function you will also find on code online. You shouldn't need to change it: it's not unique to our board.
- **All other display functions** follow the same general set of functions within them, and they are all intuitively named.
    - First, the display will be cleared. If you do not do this, the new text will show up on top of the old text. If this changes or it's the first display function used, formatting functions, such as for font and text size, will be called. Then, the cursor is set to the intended location on the OLED (ours is 64 x 128) and the text is set to display. The OLED won't actually change until you call display.display(), so this is at the end of all the display functions.
    - All of these functions were created just to make the other functions (re: calibration functions) easier for you to read and understand.

main.cpp is the file with the setup and the superloop: this is the main file (aptly named) that calls functions from the other files and follows the code flow diagram.

- First, we are calling all the necessary header files and initializing the variables we need later in the code. Nothing too crazy here, and comments further explain what you're looking at.

- At the end of the timer variables, we are also declaring a timer initialization function and creating a timer interrupt.
- **setup** contains many functions that are needed to establish connections to allow the following code to run and interface with other board components & outlets. This function will set up the OLED, user button, EMG pins, motor pins, and timer pins and interrupt. You shouldn't have to really change this unless you add another component to the board that you want to interact with.
- **loop** is our main superloop!
    - First, it checks if the button was pressed.
    - If yes (i.e. low), we will enter the switch statement, which will place us at the correct calibration step, call the corresponding function, and increment our calibration step tracker variable. Then, there is a delay to minimize mechanical issues with the button.
    - If the button was not pressed, we go ahead and check to see if the target muscle was flexed.
        - If yes, we open or close the hand and reset the hand state variable.
    - If it's time to check the battery voltage, we read the battery voltage. If the voltage is low, we turn the LED on. Then, we reset the timer flag.
- **onTimer** is our timer interrupt handler: when the timer is up, we set a flag. This flag tells us to check the battery voltage.

motor.cpp is the file that contains our two motor functions.

- **runMotor** opens/ closes the hand for a set amount of time, then resets the hand state variable.
    - if the hand is closed, we turn the motor clockwise to open the hand. We delay so the motor can turn as much as needed before stopping the motor. The position variable is changed. Then we delay for the hand to adjust before returning the new position variable.
    - if the hand is open, we turn the motor counterclockwise to close the hand. We delay so the motor can turn as much as needed before stopping the motor. The position variable is changed. Then we delay for the hand to adjust before returning the new position variable.
- **moveMotor** tells the motor pins which direction to go and how fast to move, actually sending the signal to run the motor. This is called within runMotor.

sample.cpp is the file that contains the values for the digital filters for the EMG and the code to sample the EMG input.

- After calling the necessary libraries, we set our frequencies and create simple FIR notch filters. We want to filter out the 60 Hz power line noise, so we do notch filters for 60 Hz and its first harmonic, 120 Hz.
- **sample_200** collects 200 EMG samples and outputs the sample max. In the for loop, we read in the EMG data, then we run the signal through both filters (to eliminate 60 Hz and 120 Hz noise). Next, we compare this sample to the ones already collected and keep the maximum value. We set a short delay to keep the sampling frequency we want. We repeat through this loop for 200 times, and the function will output the sample's peak value.

That covers all the key code files. With these descriptions, you should be able to understand the code. This will allow you to make tweaks to accommodate design changes, such as different hand size. If you want to upload changes, click the → at the bottom of the screen to upload to the board.

Board Files

First, note that there are two boards: one board is for the batteries, and the other contains the processor and connections to other components. Both of these are needed in the final design. The power board components are easily hand soldered, but the components on the main board should be placed using both the automatic and the manual pick-and-place machines in the EIH. The proper file for the pick-and-place machine is posted on the website: be sure to upload this to an SD card before going to the EIH to put the boards together.

In the earliest uses of the board, it will likely not need to be changed. However, the files need to be accessed for fabrication. To order boards, you will need to generate a gerber file, which this website provides instructions for. Feel free to use any company: we used OSH Park because they are based in the US and have faster shipping with less direct shipping cost - each board house will have slightly different specifications for the gerber file, which can be found on their website.

CAD Files

As you know, you will want to change the scale for each user. You'll want to adjust your parameters in OpenSCAD just as you would for purely mechanical devices and export them as .stl files. You can send most of these to print as-is, with the exception of the hinges and the LowerArm. For a better fit, scaling the hinges a bit larger is likely helpful. The lower arm file, however, needs much more work.

You will want to change the connection pieces of the corresponding file we have uploaded on the website. We want to keep the body of the arm with the proper housing, but we want the connection pieces to fit the new hand and socket. For a bit more information on how we created our file, refer to section 3.6: Detailed Operation of Hand & Socket Design in our final report.

Next, you will likely want to change the length of the arm to be closer to the needs of the user. When you do this, it is to ensure that the mounting holes, and component housings are still compatible with the parts. The motor, main board, and battery board still need to fit on the arm. This may require repositioning of the battery holder and the main board: the motor has a bit less leeway.

6 To-Market Design Changes

Each of the subsystems in the project can be improved in a few ways if the arm was to be taken to market. The first change made to the EMG system would be using a more

effective Faraday cage to eliminate electromagnetic interference. The arm currently uses aluminum foil, which reduces some of the noise, but is prone to mechanical gaps which limit its effectiveness. In practice, the user will need to access the inside of the hand to calibrate and change the batteries. This necessitates some type of opening to free space which EM waves can propagate through. Optimally, the Faraday cage would have no gaps in it during normal operation. With just aluminum foil, this is difficult to do. Although other metals, like copper are more conductive than aluminum, aluminum is very light, so we'd likely stick to aluminum as the material going to market. A better option to ensure continuity of the Faraday cage during normal operation would be using some form of sheet metal. Molding a layer of aluminum to the shape of the arm would be impractical, especially if we are trying to keep costs low. A happy medium could look like a solid aluminum lining around the opening in the arm that mates to another aluminum lining on the hatch. This solid aluminum would be fixed to a foil that lines the inside of the arm. To further improve the noise isolation, if needed, we could use a thicker foil or add more layers of foil. This would mainly benefit the operation in very noisy electromagnetic environments (near a generator etc.).

Another additional feature that could be added to the EMG sensing system is an error detection code that recognizes if an electrode falls off of the user. All of the hardware for detecting an electrode disconnection event is already set up to support this system, so the only necessary modifications would be adding a new conditional to the EMG sensing loop. When the electrodes are attached to a person, the EMG hardware gives predictable readings, averaged over a sampling cycle. Adding a conditional to detect significant deviation from these readings would add the error detection functionality.

For the power system design, a very convenient feature to offer to the user is more resolved battery charge tracking. The current system only can alert the user if the battery is "low." This is because the system only monitors the battery's voltage, not the amount of power it has used. There are battery monitoring ICs that can be used for this, such as TI's Impedance Track technology or Coulomb counter ICs. The best way to communicate the charge to the user would be a series of LEDs, a multicolor LED, or a small screen visible from the outside of the arm. The main constraint of adding in more hardware quickly becomes space. During our design process, one of the main changes that we made from our proto-board to our final board was adjusting the layout to better accommodate the limited space available inside the arm. Especially if the end user of the arm is a small child, to have a realistically sized arm, space is extremely limited. Given the current design, the best solution to create more room for added peripherals would be creating a two-sided PCB and utilizing the area around the mounting screws. Due to the time constraint, drastically changing from the protoboard design to a new PCB with more layers would not have been possible. In addition, boards with more layers would drive up costs. Another place for the power system to improve is the electrical connections from the battery board to the main board. Our final design uses JST connectors. The connectors work fine, but aren't very durable and wouldn't be reliable. The user is inserting and removing batteries directly next to the JST connector and in general the arm will be subject to daily wear and tear, which could be significant, especially for a child. Opting for a more permanent solution that the user couldn't

accidentally remove while servicing the batteries or during daily life would be best. An easy to market change here would be using Molex connectors instead.

For the hand and socket design, the two main limitations were 3D print quality and joint hinge and threading design. First, the goal of improving print quality would be to make the arm look more like a human arm. The current design has a rough, stepped texture, which approximates the shape of a hand, but doesn't look or feel much like a real biological hand. Using a 3D printer with a higher resolution between layers of material would reduce the stepped texture and casting the hand with a mold would completely eliminate the issue. Also, using a plastic that more closely matches human skin tone would make the hand and socket look more like a human hand. Due to the constraints of the EIH and our budget, these options were not possible. Another limitation of our 3D print was the texture of the inside of the hand. This texture is very important for how the hand grips objects. The smooth texture of the 3D print is not very sticky, and therefore not very good at picking things up. Using some sort of molded rubber or spray on rubber would improve this. Increased friction between the hand and objects also reduces the force output front he hand needed to successfully hold objects, which makes the system more energy efficient. Finally, having access to higher quality prints, in combination with better mechanical designs, would improve the joint resiliency. This would result in more fluid grasping and would require less torque from the motor to grasp and release the hand.

Second, we experienced major difficulties with the hinges and the threading. For demonstration, the prototype that we present does not spring into an open position when it is not grasped. In other words, the original goal was for the hand to have two states: firmly closed and firmly open. However the demo day design's states were firmly closed and slack. In the slack state the fingers can freely open and close. This shortcoming is due to a few things. First due to the delayed shipping for the final board, we could only use half of the voltage that we intended to provide the battery. This significantly reduced the maximum torque that the motor was capable of outputting. Second, even with full voltage, the motor likely wouldn't be strong enough to grasp the fingers. This leads to the final cause: too much force required to close the hand. The cause of this is that the joints were too firm. The joint flexion requires deforming soft plastic pieces within each finger. The plastic pieces used in the final design were soft, but not soft enough that the motor could pull against them. To demonstrate the hand, we opted for very soft improvised joints constructed out of wires and duct tape. They allow flexion, but are not elastic enough to return the fingers to a firmly open state. The ideal joints would be somewhere between the original elasticity and the improvised joint elasticity.

7 Conclusions

The ARM ProsthEEsis team was ultimately successful in the creation and demonstration of a working prototype of an EMG-controlled prosthetic hand. By integrating the EMG sensing, user interface, motors and actuation, hand and socket design, and power subsystems, a functioning end-to-end design was brought to life and successfully demonstrated. In line with the overall goal of this project, the lessons

learned and all engineering materials and documentation will be passed on to e-NABLE ND so that our work can be used to improve the functionality of low-cost prostheses for real users.

8 Appendices

Hardware Schematics

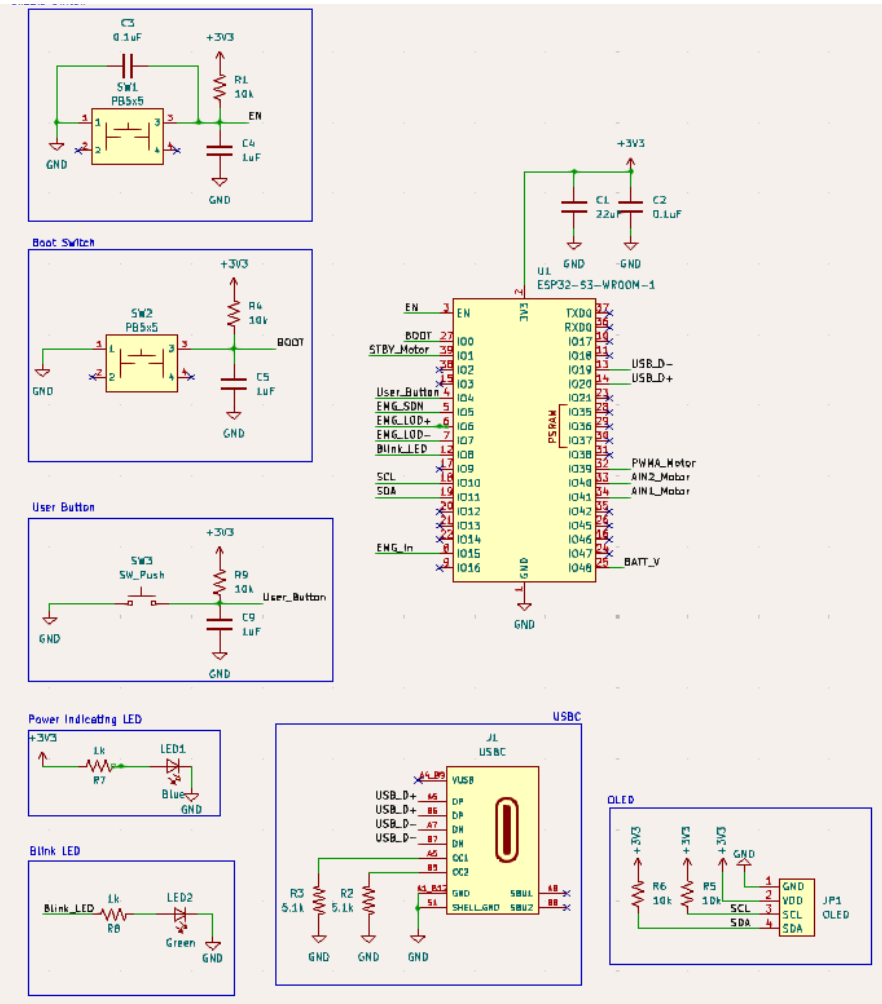

Figure A1. Schematic of MCU and Peripherals

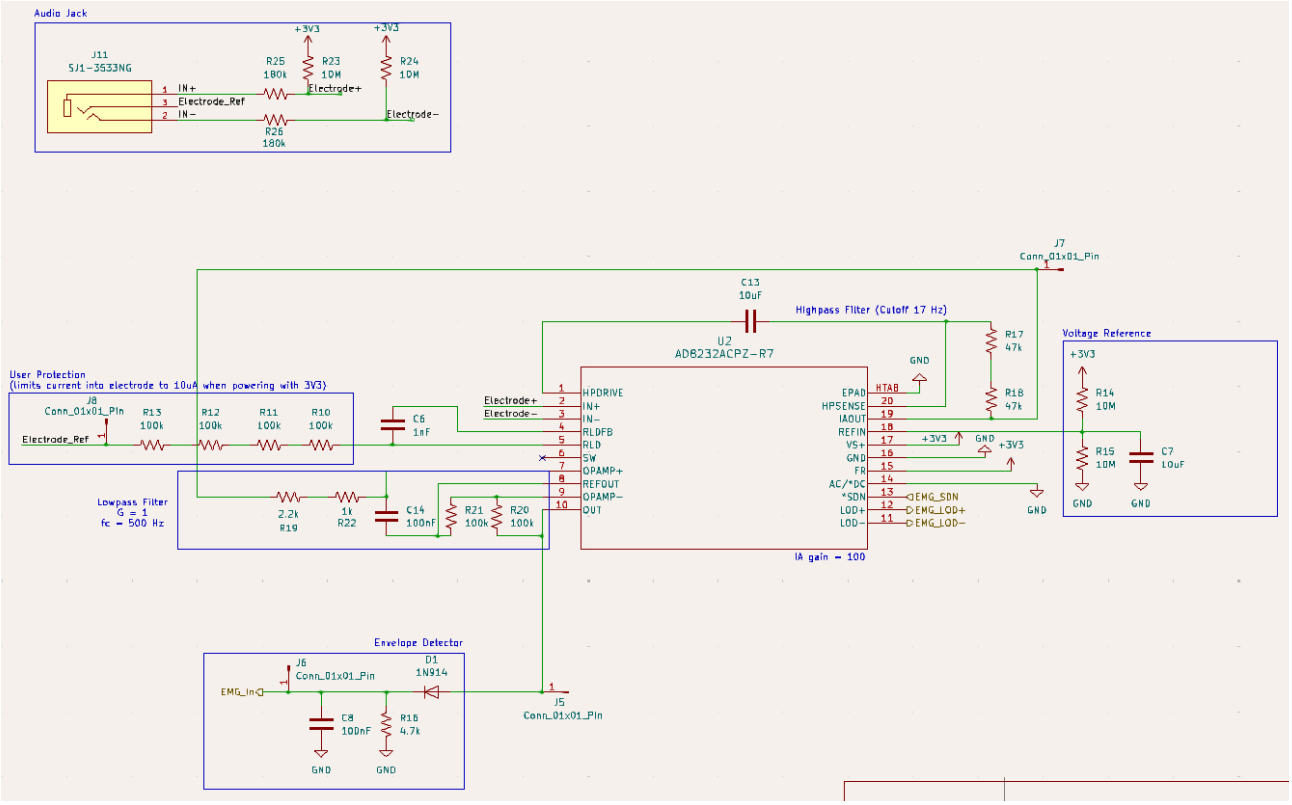

Figure A2. Schematic of EMG Sensing Subsystem

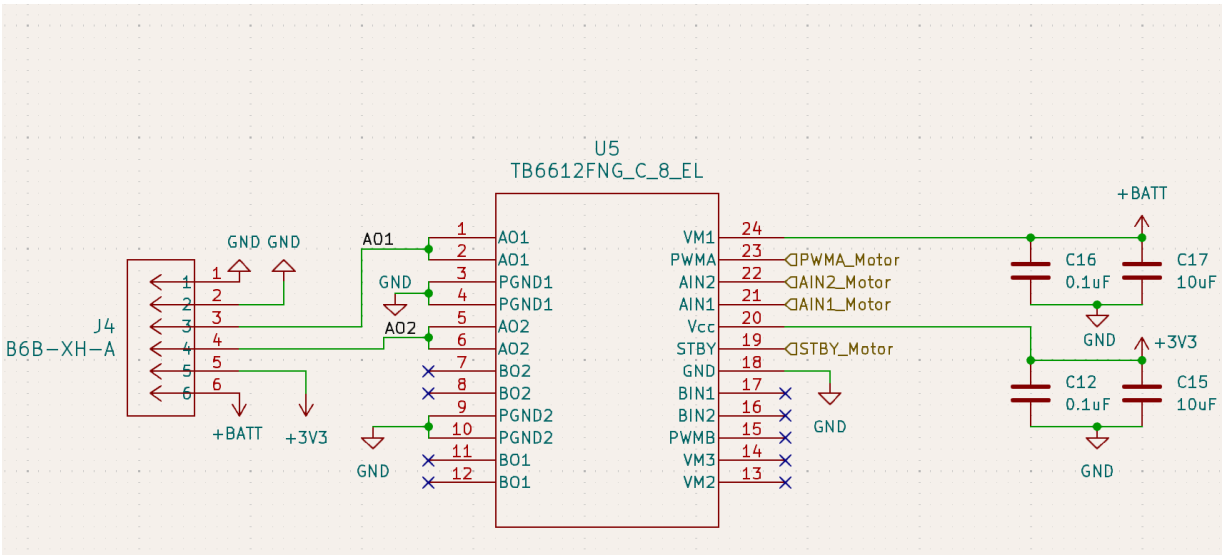

Figure A3. Schematic of Motor Subsystem

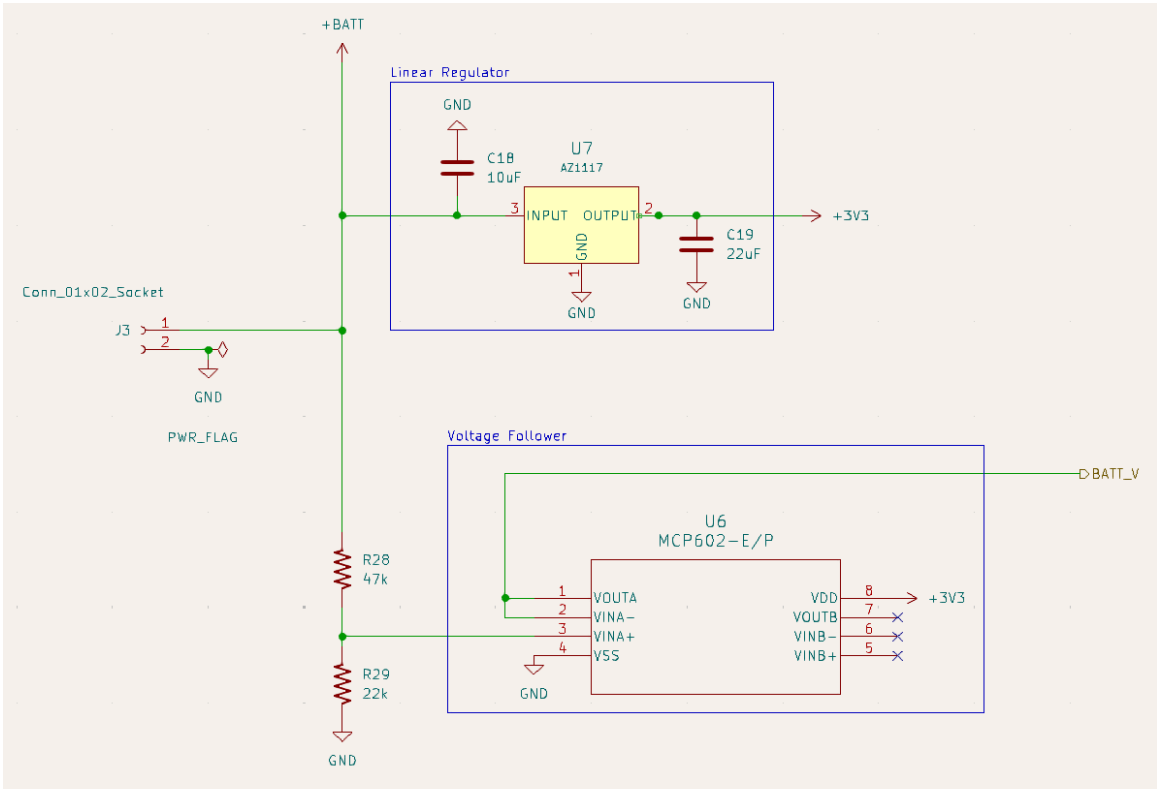

Figure A4. Schematic of Power Subsystem

Software Listing

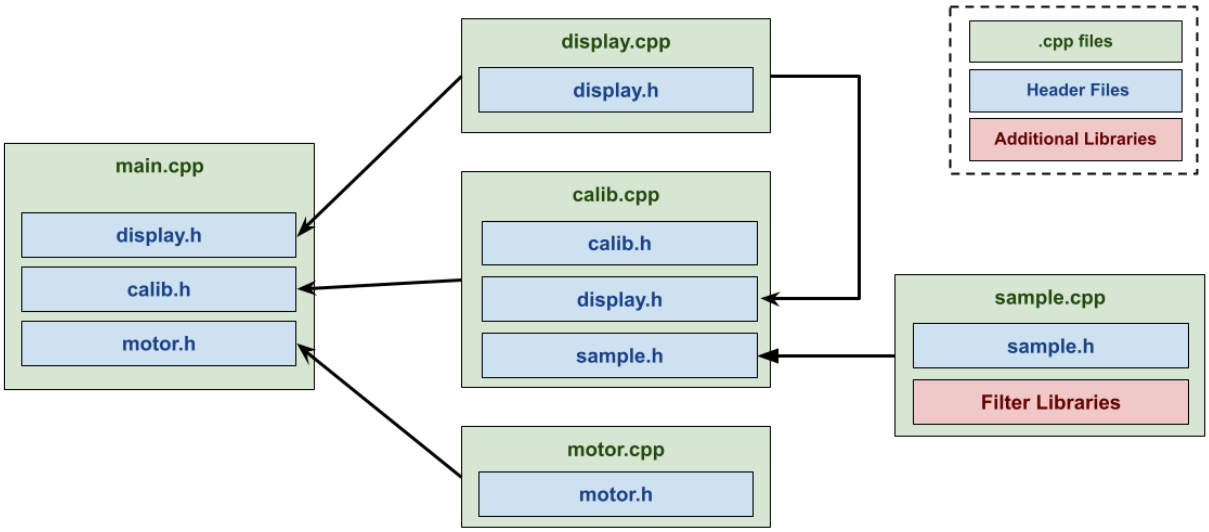

Figure B1. Organization of Software Files

Platformio File

```
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html

[env:esp32dev]
platform = espressif32
platform_packages = platformio/framework-arduinoespressif32 @ https://github.com/espressif/arduino-esp32.git#3.0.5
    platformio/framework-arduinoespressif32-libs @ https://github.com/espressif/esp32-arduino-libs.git#idf-release/v5.1
board = esp32s3-1_16_8
build_flags =
    -DARDUINO_USB_CDC_ON_BOOT=1
    -DARDUINO_USB_MODE=1
    -DCORE_DEBUG_LEVEL=4
board_build.variants_dir = variants
monitor_speed = 115200
framework = arduino
lib_deps =
    tttapa/Arduino Filters@^1.0.0
    adafruit/Adafruit GFX Library@^1.12.0
    adafruit/Adafruit SSD1306@^2.5.13
```

main.cpp

```
#include <Arduino.h>

#include <SPI.h>
#include <Wire.h>

// Custom Includes
#include "display.h"
#include "calib.h"
#include "motor.h"

// #defines
  // EMG variables
#define EMG_SDN 5       // shutdown pin for EMG processing chip (low is shutoff)
#define EMG_LOD_pos 6   // leads off detection for IN+
#define EMG_LOD_neg 7   // leads off detection for IN-

  // power pin
#define LED 8   // used to notify user of low power

// additonal variables
  // variables for chunks of samples
bool muscle = 0;    // return value of senseEMG --> do we run the motor?
bool handState = 0; // return value of runMotor --> is the hand open or closed?

  // calibration variables
int calibStep = 0;    // where are we in calibration?
int buttonState;      // did you press the button?
float avgRelax;       // the average max EMG value in relaxed state
float avgFlex;        // the average max EMG value in flexed state
float threshold;      // the threshold EMG value for turning on the motor

  //timer variables
# define BLINK_LED 8
# define V_BATT 12
# define VOLTAGE_DIVIDER_RATIO (22.0/(22.0 + 47.0))
bool timer_flag = false;    //flag that is triggered when timer interrupt expires
hw_timer_t *timer = NULL;   //timer structure
```

```
38  void Timer_Init(void);
39  void ARDUINO_ISR_ATTR onTimer();
40
41  ////////////////////////////////////////////////////////////////////////////////////////
42
43  // setup
44  void setup() {
45    // OLED setup
46    Serial.begin(115200);
47    init_display();
48    // handle display buffer
49    display_clear();
50    delay(500);
51    // button setup
52    pinMode(BUTTON_PIN, INPUT_PULLDOWN);  // make GPIO4 an input & enable pulldown pin
53
54    // EMG setup
55    pinMode(EMG_SDN, OUTPUT);
56    pinMode(EMG_LOD_pos, INPUT); // high when IN+ electrode is disconnected
57    pinMode(EMG_LOD_neg, INPUT); // high when IN- electrode is disconnected
58    pinMode(LED, OUTPUT);
59    delay(1000);
60    digitalWrite(EMG_SDN, HIGH); // enable the EMG chip
61
62    // Motor Stuff
63    pinMode(STBY, OUTPUT);
64    pinMode(PWMA, OUTPUT);
65    pinMode(AIN1, OUTPUT);
66    pinMode(AIN2, OUTPUT);
67
68    digitalWrite(STBY, HIGH);  // Enable motor drive
69
70    //timer setup for battery monitor
71    timer = timerBegin(1000000);    // Set timer frequency
72    timerAttachInterrupt(timer, &onTimer);    // Attach onTimer function to our timer.
73      // Set alarm to call onTimer function every x (value in microseconds).
74      // Repeat the alarm (third parameter) with unlimited count = 0 (fourth parameter).
```

```
  timerAlarm(timer,2000000*30, true, 0);
  pinMode(BLINK_LED, OUTPUT);
  pinMode(V_BATT, INPUT);

}

////////////////////////////////////////////////////////////////////////////////////////////////

// loop
void loop() {
  // check the button
  buttonState = digitalRead(BUTTON_PIN);
  // if you press the button
  if(buttonState == LOW){

    switch(calibStep){
      case 0: // the case 0 prevents the EMG from starting before calibrating upon startup
        Serial.println("button pressed, step 1");
        calibBegin();
        calibStep = 2;
        break;
      case 1:
        Serial.println("button pressed, step 1");
        calibBegin();
        calibStep++;
        break;
      case 2:
        Serial.println("button pressed, step 2");
        avgRelax = relax();
        calibStep++;
        break;
      case 3:
        Serial.println("button pressed, step 3");
        avgFlex = flex();
        calibStep++;
```

```
110         break;
111       case 4:
112         Serial.println("button pressed, step 4");
113         threshold = calibEnd(avgRelax, avgFlex);
114         calibStep = 1;
115         break;
116 
117     }
118     // wait some time so the button can be unpressed
119     delay(250);
120   }
121   // if you are not calibrating
122   if(calibStep == 1){
123     // get EMG data
124     muscle = senseEMG(threshold);
125     // if the muscle is flexed, run the motor
126     if(muscle == 1){
127       // run motor code
128       handState = runMotor(handState);
129     }
130 
131     // during normal, non-calibrating operation, check the battery level when the timer goes off
132     if(timer_flag == true)
133         {
134           //read battery voltage
135           float battery_voltage = (analogRead((V_BATT))/4095.0)*3.3/VOLTAGE_DIVIDER_RATIO;
136           // print battery voltage
137           Serial.print(battery_voltage);
138           Serial.print("V \n");
139 
140 
141           // low battery
142           if(battery_voltage <= 7.4){
```

```
143                 Serial.print("battery low\n");
144                 digitalWrite(BLINK_LED, HIGH);
145               }
146
147               // battery dead... if this happens, the battery protection circuitry
148               // in the batteries will trip and they will shut off
149               if(battery_voltage <= 5){
150                 Serial.print("Shutting down\n");
151               }
152
153               //reset timer flag
154               timer_flag = false;
155             }
156
157     }
158   }
159
160   ////////////////////////////////////////////////////////////////////////////////////////////////////
161
162   /* Timer Interrupt Handler */
163   // for battery monitor service routine, this function sets the timer flag when the timer expires
164   void ARDUINO_ISR_ATTR onTimer() {
165
166     if(timer_flag == false)
167     {
168       timer_flag = true;
169     }
170
171   }
```

display.h

```
1   // OLED libraries
2   #include <Adafruit_GFX.h>
3   #include <Adafruit_SSD1306.h>
4   #include<Fonts/FreeMono9pt7b.h>   // add external font
5
6   // OLED variables
7   #define SCREEN_WIDTH 128 // OLED display width, in pixels
8   #define SCREEN_HEIGHT 64 // OLED display height, in pixels
9
10  #define OLED_RESET -1 // Reset pin # (or -1 if sharing Arduino reset pin)
11
12  // Function Definitions
13  void init_display(void);
14  void display_clear(void);
15  void display_intro(void);
16  void display_button_press(void);
17  void display_relax(void);
18  void display_flex(void);
19  void display_countdown(void);
20  void display_white(void);
21  void display_complete(void);
22  void display_recalib_option(void);
23  void display_goodbye(void);
```

display.cpp

```cpp
#include "display.h"

#ifndef DISPLAY_EMG_
    #define DISPLAY_EMG_
    // Declaration for an SSD1306 display connected to I2C (SDA, SCL pins)
    Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);
#endif

////////////////////////////////////////////////////////////////////////////////

void init_display(){
  // SSD1306_SWITCHCAPVCC = generate display voltage from 3.3V internally
  if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
    Serial.println(F("SSD1306 allocation failed"));
    for(;;); // Don't proceed, loop forever
  }
}

////////////////////////////////////////////////////////////////////////////////

void display_clear(){
  display.clearDisplay();   // clear the buffer
  display.display();        // display the buffer
}

////////////////////////////////////////////////////////////////////////////////

void display_intro(void){
  display.clearDisplay();
  display.setFont(&FreeMono9pt7b); // set font
  display.setTextSize(1); // 1:1  pixel scale
  display.setTextColor(WHITE); // Draw white text
  display.setCursor(2, 15); // set cursor
  display.println("Loading");
  display.setCursor(2, 35); // set cursor
  display.println("calibration");
  display.setCursor(2, 55); // set cursor
```

```
38    display.println("process...");
39    // display intro
40    display.display();
41  }
42
43  ////////////////////////////////////////////////////
44
45  void display_button_press(){
46    // clear display
47    display.clearDisplay();
48    // create instruction display
49    display.setTextColor(WHITE); // Draw white text
50    display.setCursor(2, 15); // set cursor
51    display.println("Press the");
52    display.setCursor(2, 35); // set cursor
53    display.println("button to");
54    display.setCursor(2, 55); // set cursor
55    display.println("continue.");
56    //display instructions
57    display.display();
58  }
59
60  ////////////////////////////////////////////////////
61
62  void display_relax(void) {
63    // clear display
64    display.clearDisplay();
65    // create instruction display
66    display.setCursor(2, 15); // set cursor
67    display.println("Relax your");
68    display.setCursor(2, 35); // set cursor
69    display.println("arm.");
70    // display instructions
71    display.display();
72  }
```

```
74  ////////////////////////////////////////////////
75
76  void display_flex(void){
77    // clear display
78    display.clearDisplay();
79
80    // create instruction display
81    display.setTextColor(WHITE); // Draw white text
82    display.setCursor(2, 10); // set cursor
83    display.println("Next, flex");
84    display.setCursor(2, 25); // set cursor
85    display.println("your arm as");
86    display.setCursor(2, 40); // set cursor
87    display.println("the screen");
88    display.setCursor(2, 55); // set cursor
89    display.println("blinks.");
90    //display instructions
91    display.display();
92  }
93
94  ////////////////////////////////////////////////
95
96  void display_countdown(void) {
97    // display 3 seconds
98    display.clearDisplay();  // clear
99    display.setCursor(55, 35); // set cursor
100   display.println("3");  // display text
101   display.display();   // display on screen
102   delay(1000); // wait
103
104    // display 2 seconds
105   display.clearDisplay();  // clear
106   display.setCursor(55, 35); // set cursor
107   display.println("2");  // display text
108   display.display();   // display on screen
109   delay(1000); // wait
110
```

```
  // display 1 second
  display.clearDisplay();  // clear
  display.setCursor(55, 35); // set cursor
  display.println("1");  // display text
  display.display();   // display on screen
  delay(1000); // wait
}

////////////////////////////////////////////////////

void display_white(){
  display.clearDisplay(); // clear
  display.fillScreen(WHITE);  // flash white
  display.display();  // display on screen
}

////////////////////////////////////////////////////

void display_complete(){
  // clear display
  display.clearDisplay();
  // create end part 1
  display.setCursor(2, 10); // set cursor
  display.println("Great! The");
  display.setCursor(2, 25); // set cursor
  display.println("calibration");
  display.setCursor(2, 40); // set cursor
  display.println("process is");
  display.setCursor(2, 55); // set cursor
  display.println("complete.");
  //display part 1
  display.display();
}

////////////////////////////////////////////////////
```

```
146
147  void display_recalib_option(){
148    // clear display
149    display.clearDisplay();
150    // create end part 2
151    display.setCursor(2, 10); // set cursor
152    display.println("Hold the");
153    display.setCursor(2, 25); // set cursor
154    display.println("button if");
155    display.setCursor(2, 40); // set cursor
156    display.println("you need to");
157    display.setCursor(2, 55); // set cursor
158    display.println("recalibrate.");
159    //display part 2
160    display.display();
161  }
162
163  ////////////////////////////////////////
164
165  void display_goodbye(){
166    display.clearDisplay();
167    // goodbye
168    display.setCursor(20, 35); // set cursor
169    display.println("Goodbye!");
170    display.display();
171  }
```

calib.h

```h
#define BUTTON_PIN 4 // GPIO4 is wired to the user button

// initialize functions
void calibBegin();  // calibration startup
float relax();      // get the relaxed muscle EMG value
float flex();       // get the flexed muscle EMG value
float calibEnd(float lowEMG, float highEMG);   // set EMG threshold & end calibration
bool senseEMG(float triggerVal);    // process EMG & determine muscle status
```

calib.cpp

```
#include "calib.h"
#include "display.h"
#include "sample.h"

////////////////////////////////////////////////////////////////////////////////

void calibBegin(void){
    // clear display
    display_intro();
    delay(3000);  // wait
    // create instruction display
    display_button_press();
}

////////////////////////////////////////////////////////////////////////////////

float relax(void){
    // initialize the variable to be returned
    float lowVal;
    int sample_max = 0;
    int sample_out = 0;
    display_relax();
    delay(2500);    // give the user time to read & relax the muscle

    // gather EMG data & keep sample maximums
    for(int i=1; i<=16; i++){
      sample_max = sample_200();
      sample_out = sample_out + sample_max;  // sum up the sample_max values
    }
    // get average maximum to return out of function
    lowVal = sample_out / 16;
    sample_out = 0; // reset sample_out

    // for debugging
    Serial.print("The relaxed maximum is: ");
    Serial.println(lowVal);

```

```
    display_button_press();

    // output
    return lowVal;
}

////////////////////////////////////////////////////////////////////////////////

float flex(void){
    // initialize the variable to be returned
    float highVal;
    int sample_max = 0;
    int sample_out = 0;

    display_flex();
    delay(5000);  // wait

    display_countdown();

    // display dark screen
    display_clear();
    delay(1000); // wait

    // data collection 1
    // flash screen for 1.5 seconds
    display_white();
    delay(500);
    // gather EMG data & keep sample maximums
    for(int i=1; i<=6; i++){
      sample_max = sample_200();
      sample_out = sample_out + sample_max;  // sum up the sample_max values
    }
    // keep the average sample_max
    highVal = sample_out / 6;
    sample_out = 0; // reset sample_out
```

```
73
74      // for debugging
75      Serial.println(highVal);
76
77      // clear display
78      display_clear();
79      delay(2000);   // wait
80
81      // data collection 2
82      // flash screen for 2 seconds
83      display_white();
84      delay(500);
85      // gather EMG data & keep sample maximums
86      for(int i=1; i<=8; i++){
87        sample_max = sample_200();
88        sample_out = sample_out + sample_max;  // sum up the sample_max values
89      }
90
91      // for debugging
92      Serial.println(sample_out/8);
93
94      // keep the average sample_max
95      highVal = highVal + (sample_out / 8); // sum of last average max & this average r
96      sample_out = 0; // reset sample_out
97
98      // clear display
99      display_clear();
100     delay(2000);   // wait
101
102     // data collection 3
103     // flash screen for 2.5 seconds
104     display_white();
105     delay(500);
106     // gather EMG data & keep sample maximums
107     for(int i=1; i<=10; i++){
108       sample_max = sample_200();
```

```
109         sample_out = sample_out + sample_max;  // sum up the sample_max values
110       }
111
112       // for debugging
113       Serial.println(sample_out/10);
114
115       // keep the average sample_max
116       highVal = highVal + (sample_out / 10); // sum of last average maxes & this av
117       sample_out = 0; // reset sample_out
118
119       // clear display
120       display_clear();
121
122       // get total EMG max average
123       highVal = highVal / 3;
124
125       // create instruction display
126       display_button_press();
127
128       // for debugging
129       Serial.print("The flexed maximum is: ");
130       Serial.println(highVal);
131
132       // output
133       return highVal;
134   }
135
136   ////////////////////////////////////////////////////////////////////////////////
137
138   float calibEnd(float lowEMG, float highEMG){
139       // initialize the variable to be returned
140       float returnVal;
141
142       display_complete();
143       delay(5000);
```

```
    // compute the threshold value here
    returnVal = (lowEMG + highEMG) / 2; // average of relax max & flex max

    display_recalib_option();
    delay(5000);

    display_goodbye();
    delay(2500);
    // clear display
    display_clear();

    // for debugging
    Serial.print("The new threshold is: ");
    Serial.println(returnVal);

    return returnVal;
  }

////////////////////////////////////////////////////////////////////////////////

  bool senseEMG(float triggerVal){
    // collect data
    int sample_max = sample_200();

    // for debugging
    Serial.println(sample_max);

    // determine return value
    if(sample_max > triggerVal){
      // if we flexed the muscle
      return 1; // true
    }
    else{
      return 0; // false
    }
}
```

motor.h

```h
#define STBY  39  // Standby pin
#define PWMA  40  // Motor PWM speed control
#define AIN1  41  // Motor direction control 1
#define AIN2  2  // Motor direction control 2

// Function Initialization for moving motor
void moveMotor(int speed, bool forward);
bool runMotor(bool position);   // move the hand to the other position

```

motor.cpp

```cpp
#include "motor.h"

#include <Arduino.h>
#include <SPI.h>


bool runMotor(bool position){
    if(position == 1){
      // hand is closed
      Serial.println("opening the hand");
      moveMotor(200,true);
      delay(1000);
      moveMotor(0,false);
      position = 0;
    }
    else{
      // hand is open
      Serial.println("closing the hand");
      moveMotor(200,false);
      delay(1000);
      moveMotor(0,false);
      position = 1;
    }
    delay(2500);    // wait while the hand moves
    return position;
  }

  void moveMotor(int speed, bool forward) {
    digitalWrite(AIN1, forward);
    digitalWrite(AIN2, !forward);
    analogWrite(PWMA, speed); // Set motor speed (0-255)
}

```

sample.h

```h
#define EMG_IN 15 // use analogRead(EMG_IN)

int sample_200(void);
```

sample.cpp

```cpp
#include "sample.h"

// signal filtering libraries
#include <Filters.h>
#include <AH/Timing/MillisMicrosTimer.hpp>
#include <Filters/Notch.hpp>

// filter variables
    // Sampling frequency
const double f_s = 800; // Hz (delay of 1.25 ms between samples)
    // Sampling Period
const double t_s = 1000/f_s;
    // Notch frequency (-∞ dB)
const double f_c = 60; // Hz
    // Normalized notch frequency
const double f_n = 2 * f_c / f_s;
    // Very simple Finite Impulse Response notch filter
auto filter1 = simpleNotchFIR(f_n);     // fundamental
auto filter2 = simpleNotchFIR(2 * f_n); // second harmonic

////////////////////////////////////////////////////////////////////////////////

int sample_200(void){
    int sample_max = 0;
    for(int tab = 1; tab <= 200; tab++){
        auto raw = analogRead(EMG_IN);
        auto filt = filter2(filter1(raw));
        if(filt > sample_max){sample_max = filt;} // set new max if needed
        delay(t_s);  // do approx 800 samples/ sec
      }
    return sample_max;
}
```

Additional Documentation

- [Arduino Filter Library Documentation]

Component Data Sheets

- Hosyond OLED
- AD8232 (EMG IC)
- ESP32-S3-WROOM-1U (MCU)
- AP2112 (Linear Regulator - Demo Day Board)
- AZ1117I (Linear Regulator - Final Board)
- MCP602-E/P (Op-Amp)
- DC Geared Motor
- TB6612FNG (Motor Driver - Demo Day)
- TB6612FNG (Motor Driver - Final Board)