ARM (Actuation via Real-Time Myoelectric Signals) ProsthEEsis

e-NABLE Manual

EE Senior Design 2024-25, University of Notre Dame

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

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.

1. 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.

2. 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.

3. 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.