Lab #5a – sEMG flashing LEDs on a PyBoard

Overview:

In this two-part lab, we'll

- run code on the PyBoard that detects muscle activation and lets you know via an LED or other method
- all calculations will be on the PyBoard, not the host
- This week, we'll read one muscle; next week we'll graduate to two or three muscles at the same time, giving us a richer choice of what we can control with them.

Legal reminder:

This lab involves capturing your sEMG. Legally, this signal does represent medical data. As such, federal law says that you have the right to keep it private, and we will of course respect that law. Please refer to our separate legal sheet (which you should have read and signed).

Background

Last week we looked at algorithms running on the host to take a raw sEMG signal as input and turn it into a "1" when a muscle is activated and "0" when not. This week we'll run those algorithms on the PyBoard.

We'll be using a slightly different version of the code this week, though. Instead of reading data from an input .csv file, it uses a timer to take a new ADC sample every 1ms.

This lab involves creative coding, thinking about the questions, and writing one report per group. Next week we'll graduate to reading two or three muscles at once, giving you the opportunity to be still more creative.

Lab setup – software

The setup for this lab involves first copying a Python program from the class shared folder to your own folder (where you can modify your copy).

The file locations are the same as usual; the class folder is Q → en1EIY → 2022f → public_html → labs → code. Use the standard Windows copy/paste to grab the file *5_emg_process_pyboard.py* and paste it into your own Z-drive folder.

Lab setup – hardware

The hardware setup is identical to what we did for lab #3 (the sEMG-on-a-scope lab). Please refer to the lab-3 instructions for a refresher.

Lab setup – electrodes

Just as with the hardware, the electrodes for this lab are identical to lab #3. Again, please refer to that lab for details.

Editing the software

Once you have *5_emg_process_pyboard.py* opened in Thonny, it's time to modify it. Most of *5_emg_process_pyboard.py* consists of code that

- digitizes the signal from the AD8232 board at 1K samples/second (similar to *adc_capture.py*)
- implements last week's analysis software in real time on the PyBoard.

The first thing is to pick your favorite parameters from last week. Enter them by modifying the code on roughly lines #19-22:

```
f1 = 16           # How long the first (smoothing) filter is
f2 = 64           # How long the second (baselining) filter is
f3 = 100          # How many samples per segment
```

and lines 26-27

```
ch0_Schmidt0=130  # Call the signal a 0 when it dips lower than this.
ch0_Schmidt1=220  #   "   "     "   " 1   "  "   "   higher "     ".
```

Near the top of the file, you will notice that *5_emg_process_pyboard.py* contains a function *`muscle_fun()`* that is essentially empty. Your first job will be to fill in this function!

As described in the previous lab, the software works in segments of **`f3`** milliseconds. At the end of each segment, *5_emg_process_pyboard.py* applies the Schmidt triggers and decides if the muscle state has changed. It then calls *`muscle_fun`*`(musc1_on, changed)`. The parameter **`musc1_on`** will either be **True** or **False,** thus telling you the new state of the muscle; **`changed`** will also be **True** or **False,** and tells you if the muscle state has changed relative to the previous time segment. Note again that *5_emg_process_pyboard.py* only calls `muscle_fun()` at the end of each time segment, rather than 1000 times/second.

Feel free to code *`muscle_fun`*`()` however you like. You have 4 LEDs to play with. You might simply turn on a particular LED on every muscle activation, or perhaps have one set of LED that indicate "muscle on" and another that indicate "muscle off." If you're feeling adventurous, the *Bonus versions* section below gives you plenty of extra ideas to try. Or you can disconnect your PyBoard from the host as detailed below, preparing it to be a portable medical device.

Once you have your code working, you should play with the value of **`f3`**. You may note some tradeoffs between noisyness and responsiveness that weren't obvious last week.

Debugging

Hopefully all of your code will work smoothly and correctly the first time you try it 😊. But what if it doesn't? How can you figure out what part is going wrong? Debugging can be an art as much as a science – and an art that you will get *lots* of practice at in the next few years!

Anyway, here are a few suggestions that may be useful:

- Sometimes the electrodes or the preamp are not working. The easiest way to check this out is to connect an oscilloscope to the AD8232 output, just as we did in lab #3. If you don't see a signal, then you know either the electrodes or the AD8232 are broken. If you do see a signal, then they're fine.
- You can try playing with the algorithm parameters to see if that helps; e.g., make the Schmidt-trigger thresholds higher or lower.
- You can always add `print()` statements to `muscle_fun()` to ensure that it is being correctly called. Note that printing *too* much information may slow down the program

enough that it is no longer able to sample at 1000 times/second and may eventually crash.
- You can use *adc_samples.py* to capture your signal and then examine the graph or use it to readjust the algorithm parameters (since your signals may be stronger or weaker than last week).

What to turn in:

A lab report that
- contains your code for `muscle_fun()` and describes what it does
- answers the questions below.

You need only turn in one report for each group. As usual, you should show off your creation(s) to one of us before you leave.

Questions:

- In this lab, you got to play around with `f3` (the length of the segment where we do our root-mean-square averaging). What are the pros and cons of making `f3` larger or smaller?
- Coming up with reasonable values of all these parameters is a nuisance! How might you either automate or ease this task (at least for the two Schmidt-trigger levels)?
- What do think would happen if your `muscle_fun()` function took a long time to run? This might happen because you had lots of really slow code in it, or perhaps because you put a `delay()` statement in it. Feel free to try this rather than guessing 😊.

Bonus versions

So far you've perhaps just flashed the same LED combination every time your muscle has activated. A next level of complexity could be to implement "sequences;" e.g., turn on a red LED on the first muscle activation, green on the second, etc.

You can do this by storing the current "state" – i.e., remembering which LED is currently on – in a variable. The Python trick to make this work, though, is that you have to set up the variable *outside* of the `muscle_fun()` function (thus making it what a programmer would call a "global" variable).

You can also experiment with other outputs instead of (or in addition to) the LED; for example, creating sound with headphones or a speaker as we did in lab #2. (If you want to try working with sound, the simplest thing to do is probably to copy `play_note()` and the DAC instantiation from *2_music_day.py* to today's code).

A nice fun bonus version could also become the beginning of your final project.

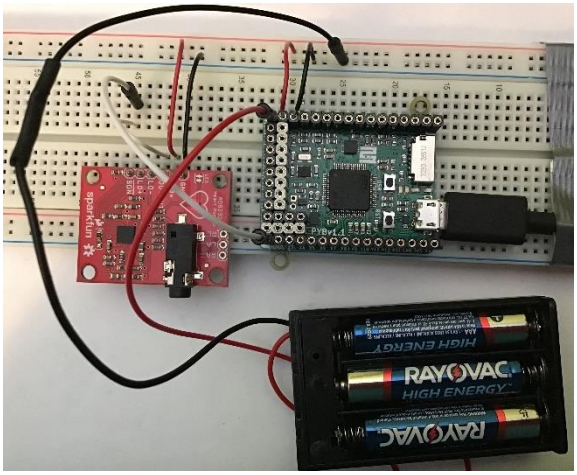
Running disconnecting from the host

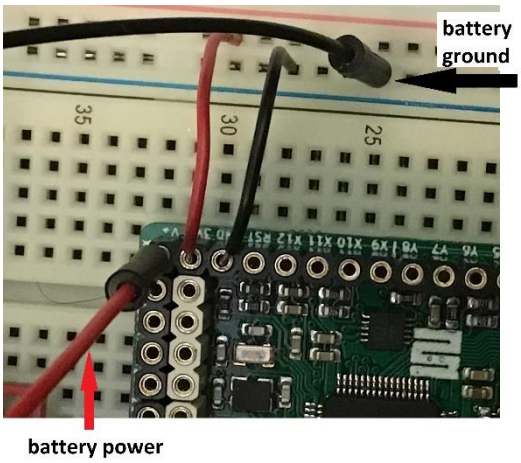
So far, we've used Thonny to run programs on the PyBoard. Thonny is convenient because it lets us use a keyboard and a display, allowing us to edit programs and easily see output. But Thonny requires a physical USB cable between the PyBoard and host, and it's often nice to run "untethered."

Running untethered – i.e., with the PyBoard completely disconnected from any host – is easy to do, but takes a few steps.

1. Make sure that *5_emg_process_pyboard.py* is completely finished and ready to go.

2. Using standard Windows tools, copy *5_emg_process_pyboard.py* to a new file *main.py*
3. In Thonny, click on View→Files to make sure that the Files display is present (usually on the left of the Thonny window). Then use Files to copy *main.py* directly to the PyBoard (just like how you have uploaded *sample.csv* from the PyBoard to the host, except in reverse).
4. Unplug the USB cable from the host PC. Leave it connected to the PyBoard, since the PyBoard USB port is kind of delicate.
5. In addition to sending data back and forth, the USB connection was also providing the PyBoard with power. Thus, you must now give the PyBoard separate battery power. Grab a 4.5V battery pack (i.e., three AAA batteries) and wire it up to the PyBoard as shown below. The battery's red "+" wire must drive one of the PyBoard's V+ pins (*not* a 3V3 pin); The battery's black "-" wire must drive both one of the PyBoards GND pins and also the AD8232 GND pin; and finally one of the PyBoards 3V3 pins must drive the AD8232 3.3v pin.





The two pictures above show what we just discussed in item #5 – the red battery wire drives the corner **V+** PyBoard pin, *not* the neighboring **3V3** pin. Instead, the **3V3** pin is how the PyBoard drives the breadboard's topmost red row to 3.3 volts (and thus drives the AD8232 board to 3.3V as well).

The next time you press the reset button (RST) on the PyBoard, it will automatically execute *main.py* and your program will execute. Of course, you won't have use of a keyboard or screen – but you will still have the LEDs.