ECE 167 Sensing and Sensor Technologies

Lab #0

January 16, 2024

Check-off: Caden Grace Jacobs

Time: 01/16/2024, 12:30 pm

Commit ID: 1211bdc61c561845d9a0e263469ecdb06abef55b

Collaborators: Niki Mobtaker

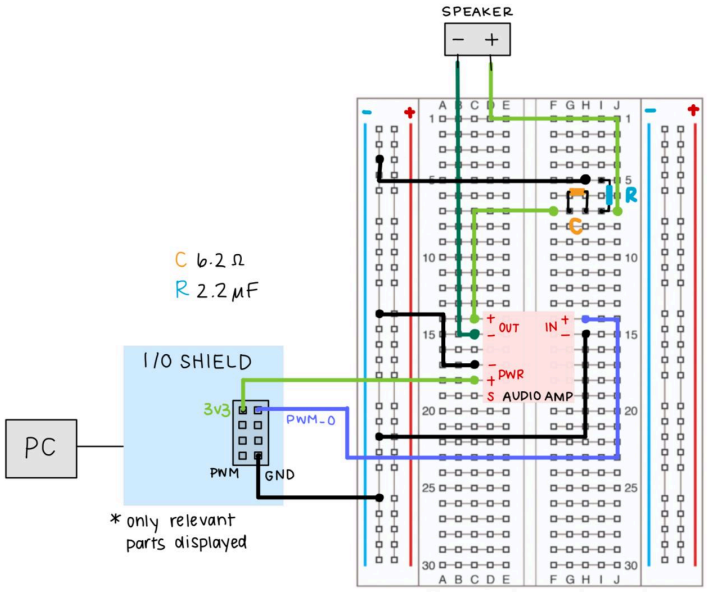
I. Overview

Lab 0 is an introduction to the development environment for the hardware and microcontroller by testing the hardware with A/D readings and generating tones with the speaker. This allowed us to apply the topics learned in class about low-pass filters, poles and signals.

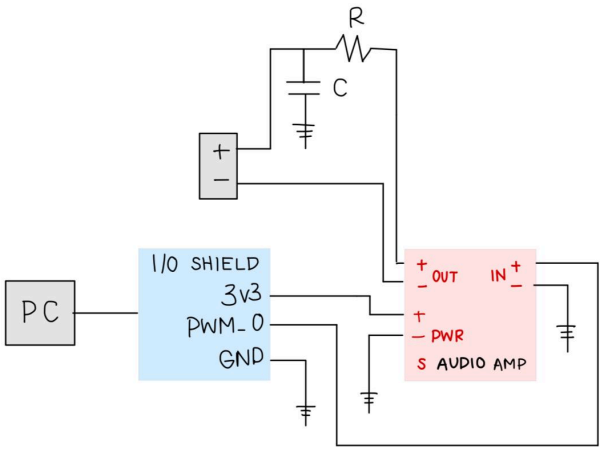
Parts Used:

- COM-11089
- TPA2005D1 Mono Audio Amp Breakout
- STM32
- 6.2 Ω Resistor
- 2.2 μF Capacitor

II. Wiring and Diagrams



(a)



(b)

Diagram 1. (a) Breadboard connections (b) wiring diagram

III. Project Design

The design of this project was divided into three parts with the first being setting up the environment. The next two parts involved the STM32 and the breadboard and only those two will be discussed.

Part 2: Hello World

Once I initialized the components of the lab I ran the printf command with “Hello World” within a while (TRUE) loop so it would continuously print. This result is in Diagram 7.

Part 3: Testing the Hardware

This part involved testing the hardware and is divided into five sections.

Section 1: Hello A/D (Reading an analog signal)

From the data sheets for the microcontroller and the speaker I could figure out what ports to connect as shown in Diagram 1. The value returned from `ADC_Read(POT)`, which returns a 12-bit ADC reading based on the selected ADC channel of POT (potentiometer), was printed along with a delay for ease of reading the results as shown in Diagram 8.

Section 2: Simple Tone

The tone is set using the PWM library with a frequency between 100 Hz and 100 kHz possible. I used the PMW_0 pin from PWM to pass the signal to the audio amplifier PCB. The frequency varied the tone being played so higher frequencies would play higher pitched notes and the same for lower frequencies. The duty cycle was affecting the volume of the sound so a

50% was set as the basis. A lower duty cycle would produce a lower intensity and to stop the note from playing the duty cycle would be set to 0% in the DUTY parameter of the function, `PWM_SetDutyCycle(PWM_0, DUTY);`

Section 3: Tone adjusted via potentiometer

There were various iterations of design when I attempted to define the relationship between the potentiometer reading and the frequency playing. Each iteration improved the quality of the sound.

The first iteration was simply passing the raw reading of the potentiometer as the input for setting the frequency using the command: `PWM_SetFrequency(ADC_Read(POT))` so that the possible frequencies played were between 0 Hz and 4095 Hz.

To improve the design, instead of setting the frequency of the speaker at the same rate that the potentiometer signal was read, I added a delay by passing in the tenth reading of the potentiometer to set the frequency as described below in the pseudocode.

```
while (TRUE) {
   for i = 1 to 10
       frequency = ADC_Read(POT);
   PWM_SetFrequency(frequency);
}
```

Diagram 2. Setting frequency from potentiometer every 10 samples

A third improvement was necessary using a modified Exponential Moving Average Filter modeled after the relationship in Eq. 1.

$$y[i] = \alpha \cdot x[i] + (1 - \alpha) \cdot x[i - 1] \qquad (1)$$

y[i] : filtered output

α : aggressivity factor

x[i] and x[i-1] : current and previous input

The modification was made to remove the recursive portion from the function by replacing the last term to take in the previous input rather than the previous output. This change prompted the aggressivity factor to have less of an effect on how the filter worked. The last iteration yielded the desired results since the frequency being set was derived from the average of the potentiometer outputs being taken in a sample time which made for a smoother tone.

Section 4: Basic filtering of output (single pole RC to smooth tone generation)

Based on the materials in class, I created a filter to smooth the signal coming from the MCU and going to the audio amplifier. The low-pass filter would prevent higher frequencies from playing on the speaker based on the cutoff frequency. I used a 6.2Ω resistor and a 2.2 μF capacitor to create a low pass filter with a cutoff frequency of 11.66 kHz. This was done based on the relationship in Eq. 2.

$$f_c = \frac{1}{2\pi RC} = \frac{1}{2\pi(6.2)(2.2\times10^{-6})} \approx 11.66\ kHz \qquad (2)$$

The circuit in Diagram 1(b) will show how I connected the filter to the audio signal using the resistor and capacitor. If a lower cutoff frequency was desired I could've used a larger capacitor, for example a 10 μF which would result in a 2.56 kHz cutoff frequency using the same resistor. A similar result of lower cutoff frequency could be obtained from maintaining the same capacitor, 2.2 μF, but with a larger resistor, for example a 47 Ω resistor would result in a 1.53

kHz cutoff frequency. Choosing the cutoff frequency to be around 10 kHz was a design choice so that the noise while testing would not hurt to listen to and because the average hearing limit for adults is 15 kHz anyway.

Section 5: Make some music

The frequencies chosen for each button were 900 Hz for Button 0, 700 Hz for Button 1, 500 Hz for Button 2, and 300 Hz for Button 3. Each button having a frequency difference of 200 Hz against its neighbors. The frequencies set for each button were played only when the button was pressed based on the pseudocode in diagram 3. The condition of no sound played was the last condition to be executed in case there were no buttons pressed.

```
if button 0 pressed
   PWM_SetFrequency(900);
else if button 1 pressed
   PWM_SetFrequency(700);
else if button 2 pressed
   PWM_SetFrequency(500);
else if button 3 pressed
   PWM_SetFrequency(300);
else
   No sound
```

Diagram 3. Pseudocode to play set frequencies only when specific button is pressed

To determine what button was pressed, the byte returned by `buttons_state()` was shifted right by a factor that would make the LSB the bit that reflected the state of the button being checked. The result of the right shift operation was then bitwise AND-ed with the hexadecimal literal 0x1 to isolate the bit. When the expression was equal to 0 the frequency corresponding to the button would play.

```
if button 3 is pressed: ((buttons_state() >> 3) & 0x1) == 0 is TRUE
```

Diagram 4. Logic to determine the state of button 3

The logic to determine a button press was as shown in the example for button 3 in Diagram 4. It was used to extract the only least significant bit which was representative of the state of that button.

To add an offset using the potentiometer, I used the same logic as before to determine what button was pressed but the values for the frequencies were determined differently. The offset would be 200 Hz less for each button. This would mean a range of 0-200 should be obtained from the potentiometer to use the value as an offset to the already determined frequencies of the buttons. The raw reading range of 0-4095 was mapped into the desired range of 0-200 using the input as a ratio multiplied by the desired range of 200 as shown in Diagram 4.

```
mappedRead = (ADC_Read(POT) / 4095.0) * 200;
```

Diagram 5. Mapped reading of potentiometer to range of [0, 200]

I applied the mapping to the current and previous input readings of the potentiometer to be used in the modified EMA filter whose relationship is described in Eq. (1). Once I had calculated the mapped values from the potentiometer, the output was fed into the EMA filter and the output of the filtering was set as the frequency offset by subtracting the value from the previously set frequency for the button as is illustrated in Diagram 6.

```
frequency = ((0.5) * currRead) + ((1 - 0.5) * prevRead);

PWM_SetFrequency(500 - frequency);
```

Diagram 6. Mapped frequency subtracted as offset from a set frequency

IV. Testing and Results

Part 2

```
Hello World
Hello World
Hello World
Hello World
Hell
```

Diagram 7. Results of Hello World

Part 3

Section 1

```
A/D reading from the potentiometer: 1
A/D reading from the potentiometer: 176
A/D reading from the potentiometer: 411
A/D reading from the potentiometer: 647
A/D reading from the potentiometer: 969
A/D reading from the potentiometer: 1336
A/D reading from the potentiometer: 1521
A/D reading from the potentiometer: 1685
A/D reading from the potentiometer: 2057
A/D reading from the potentiometer: 2436
A/D reading from the potentiometer: 2998
A/D reading from the potentiometer: 4029
A/D reading from the potentiometer: 4095
```

Diagram 8. Results of Hello A/D (Reading an analog signal)

Section 2

Testing the tone involved testing various values of frequencies. Based on the documentation for the PWM library, the function `PWM_SetFrequency(NewFrequency)` could take values from 100 Hz to 100 kHz so I tested values from 100-10 kHz. Beyond that frequency the tone was painful so I determined that the cutoff frequency for my low pass filter should be that frequency

While the frequency range processed is from 0.1-100 kHz the cutoff frequency to be tested would be 10 kHz, as higher frequencies cannot be heard or are painful to hear.

The duty cycle was also tested with values 0 to 100. What I noticed was that a duty cycle of 0 would produce no sound. As the cycle moved closer to 100 then the intensity of the sound increased.

Section 3

For the three iterations the goal was to improve the smoothness of the sound when varying the potentiometer. The first iteration was setting the frequency equal to the potentiometer reading at the same time the potentiometer was read. This resulted in a noise that varied too much as the potentiometer is very sensitive to motion. The second iteration involved setting the frequency less often compared to the frequency of sampling of the potentiometer. This resulted in less scratchy noise but there was still no smoothness. The third iteration involved a modified EMA filter which finally resulted in a smooth transition between frequencies and the potentiometer readings

Section 4

Adding a physical filter to the circuit involved using a resistor and capacitor to create a low pass filter with a single pole. The magnitude of the capacitor and resistor was based on the materials we had in class. Once the supply and the math matched a possible filter I added it to the board. This filter in conjunction with the EMA filter helped stop higher frequencies from playing. To test this, the frequency was set to 3 times the potentiometer reading as shown in Diagram 9. This would ensure that the values would range from 0 to 12,285 so that the filter would cut off values after 11,660 Hz.

```
PWM_SetFrequency(3 * frequency);
```

Diagram 9. Setting the frequency to 3x original

Section 5

The logic to determine and isolate the state of each button was from the button test code. Once each button was programmed with their corresponding frequencies then it began the testing of where to incorporate the software filter for the offset from the potentiometer. The EMA filter was combined with the tone mapping to generate values for the frequency offset. Piecing the mapping and filter together did not require much testing other than making sure the sampling frequencies of the hardware did not interfere with each other. This small issue was solved with adding delays to the code execution.

V. Conclusion

The lab took a couple days to complete with each day involving 3 to 4 hours of work. Overall, the setup and implementation of the development environment was straightforward. I used the SMT32 microcontroller and did not encounter any issues so far. The potentiometer is hard to move on the microcontroller so it was harder to determine what some results meant but once the filter was placed it went much smoother. The hardware testing provided valuable hands-on experience with the STM32 microcontroller and associated peripherals. The lab was a good introduction to how to use different resources like data sheets and schematics to build a circuit as well as give us a sense of future labs.