ECE 167/L: Sensing and Sensor Technologies
Lab 0 Report

Jose Santiago

Table of Contents

Lab Introduction and Overview

In this lab we were expected to familiarize ourselves with the hardware we will be using in this class, namely the UNO32. Prior to starting the lab we must make sure the proper software is installed on our computer to program the UNO32. This setup portion of the lab before actually programming the device is important and is where a couple of roadblocks I had stemmed from. The ECE 167 Student Installations PDF includes instructions for installing the necessary software for the course (MPLABX & XC32 Compiler), but the CSE 13E Tools Installation PDF includes additional information on how to install CoolTerm. Unfortunately this PDF did not mention downloading the drivers necessary for CoolTerm to establish a serial connection with the UNO32. This information was provided by a TA after entering a query on Piazza. After successfully installing the necessary hardware, the programming portion of the lab could commence. To begin a project on MPLABX, follow the instructions in the CSE 13E New Project Instructions PDF file.

The additional hardware necessary to complete this lab include, an audio amplifier PCB, a speaker, two resistors, one capacitor, a potentiometer, and several jumper cables and wires. The amp PCB, potentiometer, and speaker are required for parts 3 to 6. These are used in combination to create a tone from a PWM signal coming from the UNO32. The potentiometer is used specifically to control the volume coming from the speaker. The resistors and capacitor are used in Part 5 and 6 of the lab. They are used for the design and construction of an RC low-pass filter. The purpose of the filter is so the speaker can output a smoother sound by eliminating noise from the output of the audio amplifier PCB.

The objective of this lab is to learn how to utilize the UNO32's buttons, potentiometer, and PWM pin to control the tone of a sound using a circuit implemented with an amp PCB and speaker. In Part 1 a serial communication is established with the UNO32 and the message "Hello World!" is printed to the serial terminal. In Part 2 the potentiometer values are read from the UNO32 using the functions provided by the AD.h header file. The values are then printed to the OLED display with the functions provided by the Oled.h header file. In Part 3 a tone is generated by connecting the PWM pin on the UNO32 to the audio amp PCB and speaker on a breadboard. The ToneGeneration.h header file provides the functions for generating tone via the PWM pin. In Part 4 components from Part 2 and 3 are combined so a tone can be generated on the speaker. The tone is dependent on the potentiometer readings. Software filtering is implemented as well so noise is eliminated from the potentiometer readings. In Part 5 an RC low-pass filter is added to our circuit to filter the output of the audio amplifier PCB so noise can be eliminated in our generated tone. In Part 6 we add buttons from the UNO32 to our design. We program the buttons to output different tone frequencies. The potentiometer is also programmed so the frequencies of the buttons are altered based on the potentiometer readings.

Part 1

For part 1 of the lab we simply had to print "Hello World!" via serial communication from the microcontroller to a serial terminal. One issue I had at the beginning of this lab was that I didn't include the necessary files in my project to successfully build and program my device. I figured this out after receiving assistance in a lab section. I assumed that the only header and source file I needed to include in my project were the BOARD.h and BOARD.c files, but the

BOARD files used the serial.h and serial.c files, so I needed to include those for my code to compile. After that I only needed to make sure CoolTerm was connected to the UNO32, and print the message “Hello World!” I printed the message with printf() and the PutChar() function within the serial.h file. I wasn’t sure which was the correct method, but the manual only said to print “Hello World!” to the serial terminal, so either method should be acceptable. To enable programming to the board we also have to initialize the board with BOARD_Init() at the beginning of the main function. In every embedded program we need to make sure we do not exit the main as well. This means a while(1) command is added before the closing bracket of the main function to ensure we don’t exit main.

Part 2

For part 2 of the lab input is read from the potentiometer and the value is printed out to the OLED display. The AD.h, Ascii.h, BOARD.h, Oled.h, and OledDriver.h header files must be included to our project. The Oled.h file uses Ascii.h, OledDriver.h, and BOARD.h files. The Oled.h file is included in our program to allow us to write to the UNO32’s OLED screen. The BOARD.h file is included to allow us to initialize and program the UNO32. The AD.h file is included to allow potentiometer values to be read. These components are enabled by initialization within our code. The board, oled, and AD pins are initialized with BOARD_Init(), OledInit(), and AD_Init(). The AD_A0 pin (potentiometer) is added to enable reading from the potentiometer. This is done with the statement AD_AddPins(AD_A0). Then any additional variables we may need can be added. A couple variables used in my program were: p, to store the potentiometer value, and one char array variable, oled_str, to store the string to be printed onto the OLED.

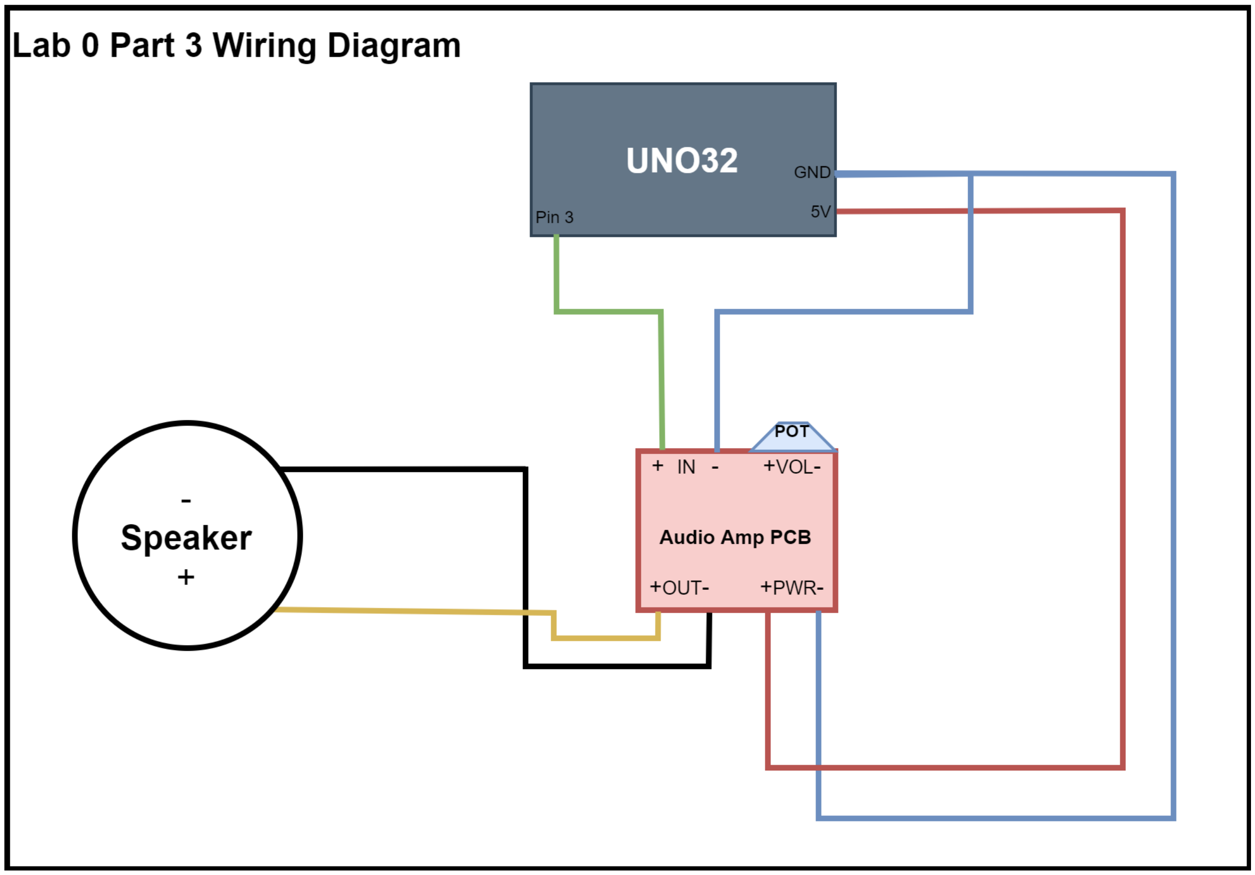
After initializing and declaring the variables an infinite loop, while(1), is added so potentiometer values can be read and written to the OLED display. Within the while loop, a check is done for new data to be read from the potentiometer with the AD.h function AD_IsNewDataReady(). This function returns TRUE when new data is ready to be read from an AD pin. An if statement is used in conjunction with this function to check if the data is ready. Within the if statement the current value of the potentiometer is stored into variable p, by using the AD.h function AD_ReadADPin(AD_A0). Nothing else is to be done within the if statement, so we close the if statement and proceed with the program.

Now that the value from the potentiometer has been stored, the OLED can be populated with the data. A char array is needed to store the data into, so the oled_str character array is used. The sprintf() function is used, coupled with our oled_str as an argument, to store the string we want to print on the OLED. The desired output string is then stored into oled_str with sprintf(oled_str, “Potentiometer value:\n%d”, p). The Oled.h functions are then used to populate the OLED. OledDrawString(oled_str) is used to write oled_str onto the OLED. OledUpdate() is called immediately after because otherwise the OLED would not update to display the string argument from printed with OledDrawString(). With this one may think the program was complete, but upon testing the readings, the potentiometer readings are different from what is expected. The potentiometer should read from 0 to 1023, but if you test the code as is, it will appear as a jumble of numbers after attempting to go from 1023 to 0. This can be confusing, but upon further inspection you may realize that if the OLED is not cleared after each print to the OLED, the most significant digits from the previous potentiometer values stay on the screen. So the OledClear(OLED_COLOR_BLACK) statement at the top of the while loop must be added, to clear the OLED display before reading and printing new values from the potentiometer. After compiling and programming the UNO32 again, the OLED should display the potentiometer values as expected from 0 to 1023.

Pseudocode:

```
while(1){

    if(NewPotentiometerValue()){
        p = PotentiometerValue();
    }
    oled_str = (“Potentiometer value: %d“, p);
    PrintOLED(oled_str);
}
```

Part 3



In Part 3 of the lab we learn to generate a tone using the ToneGeneration.h library. The BOARD.h library is also needed for this portion of the lab. There is minimal code written for this portion of the lab, but a circuit must connect the audio amplifier PCB, speaker, and UNO32 to generate a tone. We also add a potentiometer for volume control. To hook everything up I followed a tutorial online provided by SparkFun. The circuit diagram for this portion of the lab can be seen above.

For the programming portion of this lab, the board and tone generation are initialized before anything else inside the main() function. Then the frequency of the tone is set with the ToneGeneration_SetFrequency() function. There are predefined tones in the ToneGeneration.h library to use. Afterwards the tone can be turned on with ToneGeneration_ToneOn(). As mentioned previously, we don't want to exit main when we are writing embedded code, so we add a while(1) statement at the end of the main() function. After compiling this code and programming the UNO32, a tone should be heard coming from the speaker.

Part 4

Part 4 of Lab 0 consists of combining the potentiometer readings from Part 2 and the tone generation from Part 3. The potentiometer readings will be mapped to the frequency of the tone generation. Since the implementation of these components have been described in detail in Parts 2 and 3, this section will not go over them in great detail as before. This section will focus on the new code that is added for Part 4. The circuit is also unchanged from Part 3.

At the top of the main function we initialize the BOARD, OLED, the AD, and ToneGeneration components. Pin AD_A0 must be added to read from the potentiometer. Tone generation must be turned on as well. Following this are any variable declarations necessary. For this part of the lab a potentiometer reading variable (p), frequency variable (f), frequency buffer (f_buff), frequency buffer index (f_i), and average variable (avg) are recommended.

Within a while loop, the potentiometer values must be read and mapped to the frequency range of the tone generation library, 1 to 1000 Hz. First the potentiometer values are read as was done in Part 2 of the lab. A variable, p, can be used to store the readings from the potentiometer. Since the Potentiometer values read from 0 to 1023, they must be converted to go from 1 to 1000. The converted values can be stored in a new variable, f. Since the potentiometer reads 0 to 1023, we can first convert the potentiometer to read from 0 to 999 with a proportion ((p*999)/1023). Then the values can be shifted by one by adding a value of one to the conversion 0 to 999 to output values 1 to 1000. This conversion can be written in one statement: f = ((p*999)/1023) + 1. The values can be read from the potentiometer within an infinite loop like was done in Part 2. An if statement can be used again to check if new data is ready to be read from the potentiometer. Within this if statement, the potentiometer reading can be stored into variable p. Variable p can then be converted to the frequency range and stored in variable f. The final thing done within the if statement is setting frequency of the tone generation with variable f.

At this point the tone generated by the speaker is controlled by the potentiometer. The generated tone will sound scratchy as the potentiometer value changes. To fix this some software filtering must be implemented. There are several different methods of software filtering, but the method employed in this lab report will be a rolling average of the last 100 potentiometer readings. The software filtering is done immediately after the potentiometer value is read and converted to the frequency range.

The implementation of this software filtering method requires a frequency buffer (f_buff), a frequency buffer index (f_i), and an average (avg) variable. The frequency buffer is an int array of size 100. The frequency buffer index is initialized to zero. The frequency reading from the current iteration of the while(1) loop is stored into the frequency buffer. The current index of the frequency buffer is calculated by taking i_f modulus 100. The modulus is used because the frequency buffer needs to refill itself after it is full. The modulus ensures the index goes from 0 to 99. When the index reaches a multiple of 100, the index returns to zero and increments back up until the remainder reaches 99 again followed by a return to zero when the remainder reaches a multiple of 100. Immediately after storing the frequency, the average of the 100 frequency buffer elements is calculated. This can be done with a for loop to add the sum of the 100 elements, followed by the division of the sum by 100. This average is then the one that will be used to set the tone generation frequency. The index should then be incremented so the next index in the frequency buffer can be filled. This concludes the programming portion for Part 4. Something that may come up after testing this, is that the average will increase to above the frequency limit, 1000Hz. After finding this out experimentally it is suggested that a check is added after calculating the average. This can be done with an if statement immediately after calculating the average. If the average is greater than 1000, set the average to 1000.

Pseudocode:

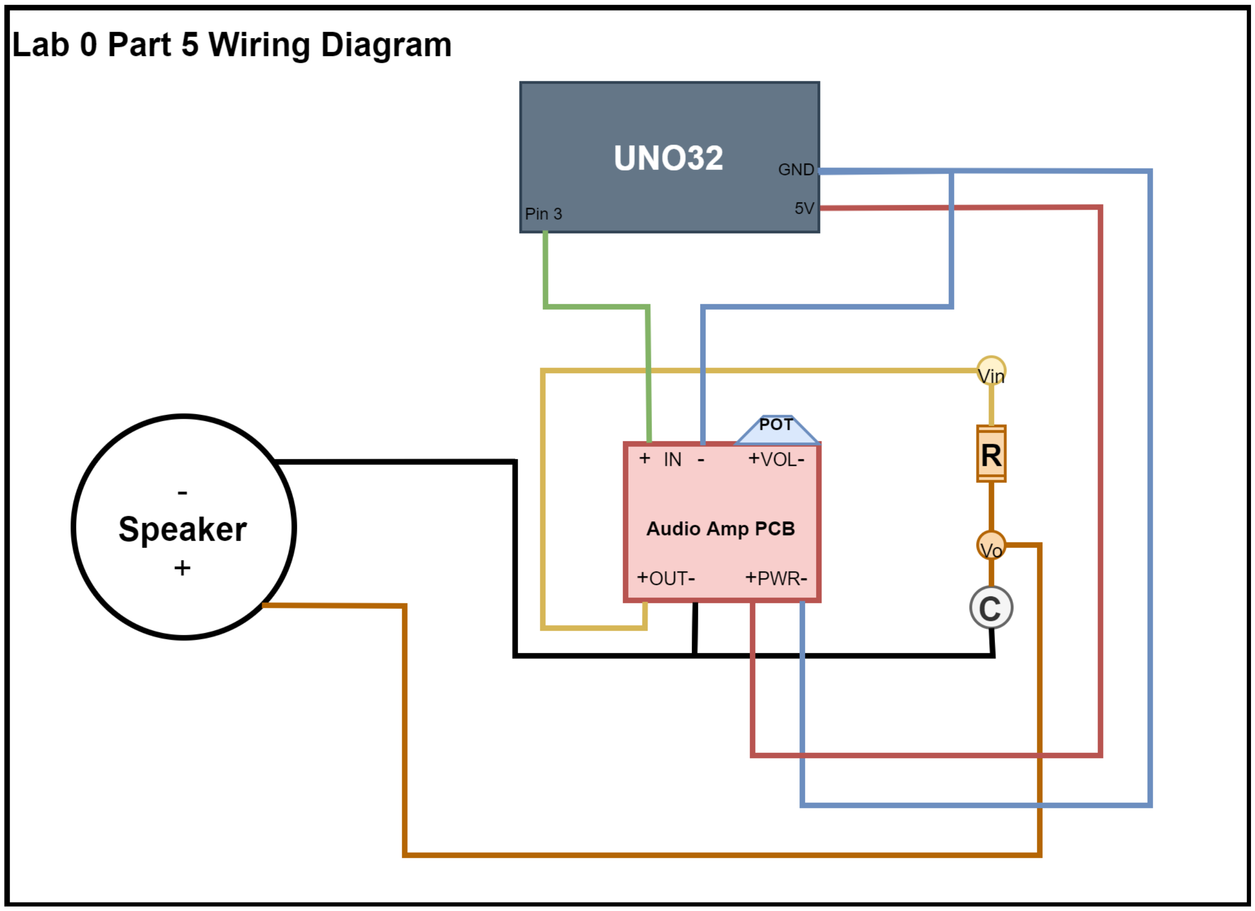
```
while(1){

    if(NewPotentiometerValue()){
        p = PotentiometerValue();
        f = ((p*999)/1023) + 1;
        SetFrequency(avg);
    }

    f_buff[i_f%100] = f;
    for(i = 0; i < 100; i++){
        avg = avg + f_buff[i];
    }
    avg = avg/100;
    if(avg > 1000){
        avg = 1000;
    }
    i_f++;

}
```

Part 5

In this part an RC low pass filter is implemented to eliminate noise from the output of the audio amp PCB. The code can be reused from Part 4. The PCB amp circuit will undergo some changes due to the addition of the RC low-pass filter. The output of the PCB amp will be fed into the Vin of the RC low-pass filter. The Vout of the RC low-pass filter is then fed to the positive terminal of the speaker. Wiring the RC low-pass filter can be confusing if you have not built the circuit in a while so it is recommended to review the circuitry of an RC filter prior to implementing it in this design.

The RC low-pass filter cuts off frequencies higher than the cutoff frequency. The cutoff frequency used in this design was 1000Hz. This value makes sense because the frequency of the audio amp PCB ranges from 1 to 1000Hz. Any frequency higher than that is unwanted noise. The cutoff frequency can be calculated with the equation $f_c = 1/2\pi RC$. The capacitor used in this design was 1 Microfarad. This means the resistance must be about 159.2357 Ohms to achieve a cutoff frequency of 1000Hz. Resistors of this magnitude are not provided in the resistor packet, but a resistance of 160 Ohms can be achieved by assembling a resistor of 150 Ohms with a resistor of 10 Ohms in series.

The 1 Microfarad resistor was found as the most effective for the RC low-pass filter by trial and error. The 1 Nanofarad resistor was used before the 1 Microfarad resistor, but the capacitance was too low. The volume of the audio amp PCB was reduced significantly with this configuration. The capacitance recommended from these findings are those in th 1 to 10 Microfarad range. Below is the circuit diagram detailing the RC low-pass filter configuration in combination with the audio amp PCB, speaker, and UNO32.



Part 6

In this part a digital music instrument is constructed using the software filtering from Part 4 and the RC low-pass filtering from Part 5. The buttons on the UNO32 will be assigned their own unique frequency. The potentiometer will be used to change the frequency the four buttons output while keeping their relative difference constant. Since the potentiometer and tone generation were discussed in previous parts of the lab, this part of the lab report will not cover them extensively.

A new hardware component that has to be implemented for this part are the UNO32 buttons. These can be accessed through the BOARD.h library. A press is signaled by the buttons with predefined macros from the BOARD.h file. The macros are BTN1, BTN2, BTN3, and BTN4. If these macros are equal to one, that means they are currently being pressed. So an if statement can be used to check if a button is pressed. The implementation in this design uses if-else-if statements to check the state of every button. A boolean, btn_pressed, is also used to signify whether a button is pressed or not. The boolean is important because if a button is currently pressed and another button is pressed after while the first button is still being pressed, the tone of the first button should continue to play. In other words, the first button to be pressed should continue to play, while any following button presses from different buttons are ignored, as long as the first button continues to be pressed. Only one button will generate a tone at the time.

If any of the if conditions are met in the if-else-if chain (if a button is pressed) and the button press boolean is equal to zero, the frequency associated with the button is set, the button pressed boolean is set to one, and the tone generation is turned on. There is an else condition at the end of the if-else-if chain that is checking for button presses. The else condition is considered the default state, when no buttons are being pressed. Within this else statement, the tone generation is turned off, and the button pressed boolean (btn_press) is set to zero.

The tones set for each button were derived from the default tones set in the ToneGeneration.h header file. They were shifted down so that the lowest frequency of the four was 1 Hz. This was so it would be easier to shift the frequencies around as the potentiometer values changed. The initial tone frequencies used were defined as macros as follows: TONE1 = 1, TONE2 = 98, TONE3 = 245, and TONE4 = 464. The lower bound is established by TONE1 since TONE1 is the smallest possible frequency, while the upper bound is established by TONE 4 since it is the largest frequency of the bunch. The upper limit of frequency as mentioned previously is 1000Hz, therefore TONE4 can only be changed by maximum 536 Hz. Therefore the potentiometer readings range from 0 to 536. To do this we establish another proportion conversion, as we did for Part 4. We store the frequency offset into a variable, f_offset. The proportion equation is used as follows: f_offset = ((p*536)/1023). The p variable is obtained the same way as in previous parts of the lab, by reading the AD_A0 pin (UNO32 potentiometer). The frequency offset variable is then sent through the software filtering, before an average is found to add to the default tones described above.

The software filtering is unchanged from the previous parts, but the average is set to 536 if the average exceeds the maximum offset value of 536. Pseudocode is provided below.

Pseudocode:

```
while(1){

    //software filtering
    f_buff[i_f%100] = f_offset;
    for(int i = 0; i < 100; i++){
        avg = avg + f_buff[i];
    }
    avg = avg/100;
    If(avg > 536){
        avg = 536;
    }
    i_f++;

    //Button pressing logic
    if(BTN1){
        if(btn_press == FALSE){
            SetFrequency(TONE1 + avg);
            btn_press = TRUE;
            ToneGenerationON();
        }
    }else if(BTN2){
        if(btn_press == FALSE){
            SetFrequency(TONE2 + avg);
            btn_press = TRUE;
            ToneGenerationON();
        }
    }else if(BTN3){
        if(btn_press == FALSE){
            SetFrequency(TONE3 + avg);
            btn_press = TRUE;
            ToneGenerationON();
        }
    }else if(BTN4){
        if(btn_press == FALSE){
            SetFrequency(TONE4 + avg);
            btn_press = TRUE;
            ToneGenerationON();
        }
    }else{
        ToneGenerationOFF();
        btn_press = FALSE;
    }

}
```

Conclusion

As stated in the introduction, this lab teaches one the basics for programming the UNO32. The use of AD pins, buttons, OLED screen, tone generation, and some circuitry are explored. This lab serves as a good refresher to embedded programming with the UNO32. Each part of the lab builds on the last, encouraging incremental design. The RC low-pass filter implementation also serves to refresh someone on basic circuitry.

The report mentions roadblocks that were induced by the setup portion of the lab, where one must install the appropriate software to program the UNO32. The first had to do with serial communication. Failure to download the proper drivers for serial communication will hinder one's ability to connect to the UNO32 via serial communication.

Another issue I encountered in this lab was the failure to include all files necessary for a program to compile. This was resolved during a lab section. It is advised for future labs to carefully check the header files necessary for a component of the UNO32. Some header files may not be obvious to include in the project, but if you look at the included files in a header file you know you need, then a string of header files necessary for that one header file to compile properly can be identified.

All in all this was a pleasant experience and a good lab to get back into the swing of embedded coding.