Final Report

Nathan Dodson & William Stadtlander

Introduction

In a time when Covid-19 is taking the lives of millions and personal well being is on our minds more than ever, people now have a greater understanding of the importance of tracking health statistics, particularly for the cardiovascular system. The advent of telemedicine is also raising the importance of average people having access to simple tools that can be used to track vital statistics and relay data to their physicians. However, getting something as basic as heart rate (HR) and arterial oxygen saturation (SaO_2) readings are difficult without the right tools. Smartphone apps abound, but don't handle the job completely. Pulse oximetry provides a safe, convenient, noninvasive, and inexpensive method to collect health information, while still providing data accurate enough for medical use.

Pulse oximetry is a noninvasive alternative for measuring arterial oxygen saturation (SaO_2). By placing a sensor on a portion of the body with high blood flow, such as the fingertip or earlobe, peripheral oxygen saturation (SpO_2) measurements are achievable within an accuracy of 2%. To accomplish this, two wavelengths of light are alternatingly passed through the high blood flow area and received by a photo sensor. The absorption of a 660nm red LED indicates oxygenated hemoglobin (HbO_2), while the absorption of a 940nm infrared (IR) LED indicates deoxygenated hemoglobin (Hb).

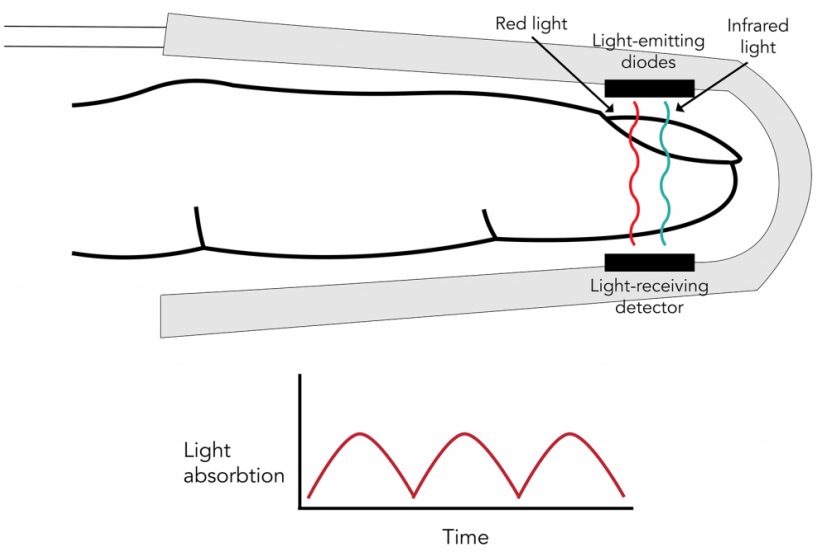

Figure 1: Light Absorption by Photo Sensor[1]

The absorbed light is then represented as a ratio of the alternating current (AC) and direct current (DC) components of the red LED and IR, respectively. Finally, the peripheral oxygen saturation (SpO_2) percent is calculated in a given equation and displayed.

The Virginia Tech Integrated Design Project for Pulse Oximetry requires the calculation of heart rate (HR) in beats per minute (BPM) and peripheral oxygen saturation (SpO_2) as a percentage. This data and the pulse signal must be visualized so the user can monitor their results. To accomplish this, a Nellcor Ds-100A sensor, a DE-9 F Terminal Block, a CD4053BE analog multiplexer, and an IIC OLED are given. Additionally, an Arduino Uno, Digilent Analog Discovery 2 (AD2) oscilloscope, and kit parts were utilized.

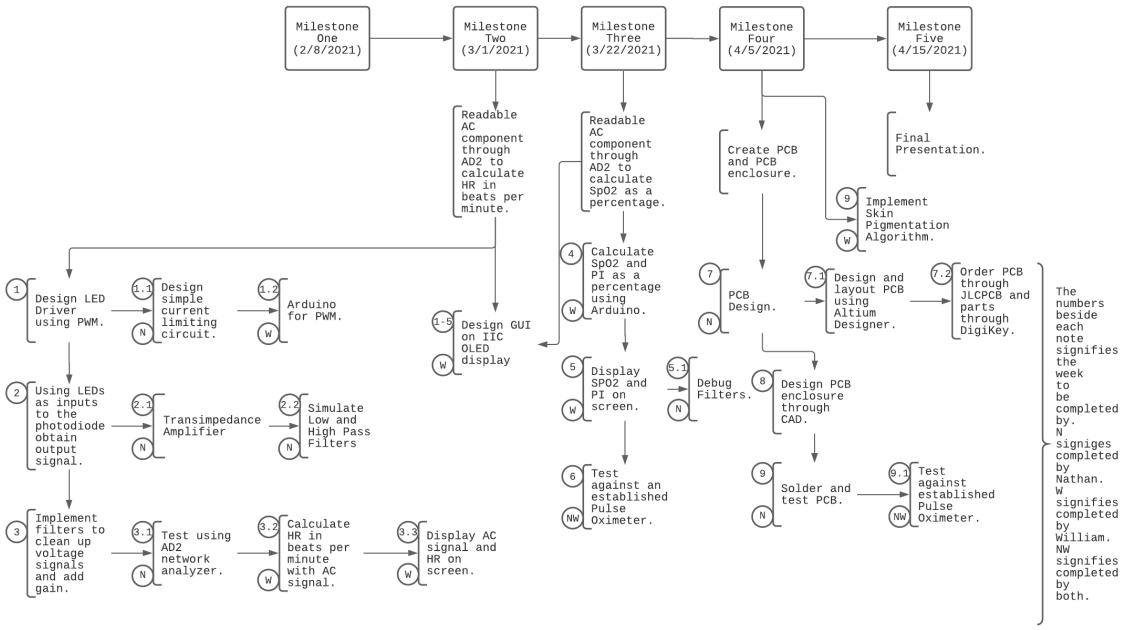

Figure 2: Initial Project Schedule

While not required, perfusion index (PI) measurement was implemented as an indicator of the relative strength of the pulsatile signal. PI is measured as a percentage typically within the range of 0.02% to 20%; it is calculated as the ratio of the given signal's AC portion to the DC component. This data must also be visualized so the user can monitor their results.

Communication between the application and user are critical. In the application's simplest form, the user must be able to identify critical measurements from the display. This could potentially create a large gap of knowledge for the user, so the user must be notified when a measurement is not in a healthy range. The dangerous levels can be indicated using the sound from a speaker, which will also allow seeing-impaired persons to use the sensor with little assistance from others.

High Level Design

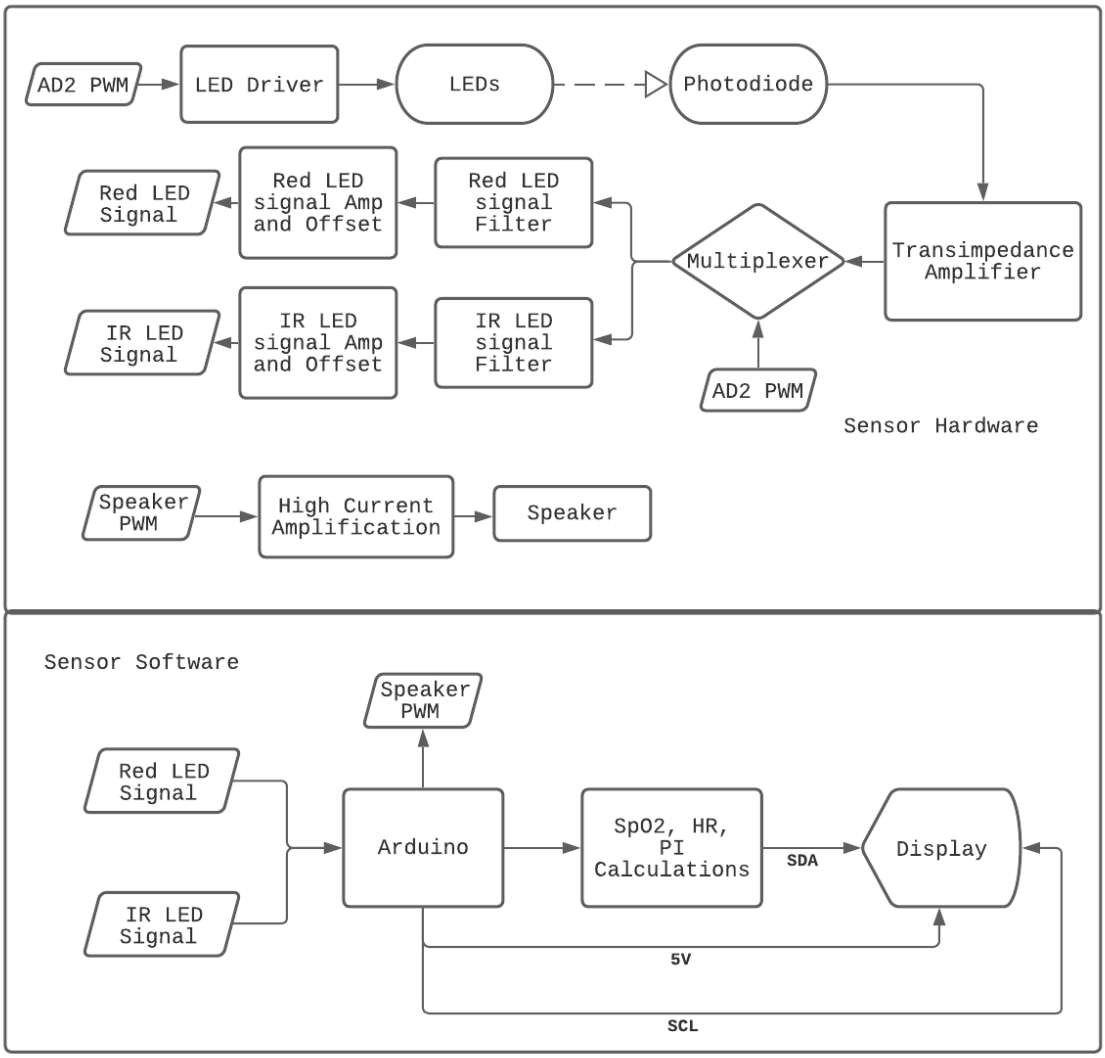

Figure 3: Block Diagram of Entire System

Sensor Hardware

The Nellcor Ds100-A SpO2 sensor comes equipped with anti-parallel LEDs and photodiode. The LEDs must be powered by a driving circuit, which alternates their power and limits current. The photodiode outputs a single signal, which is a combination of the red and IR LED absorption signals. This signal is extremely small, so it must be amplified using a transimpedance amplifier, which creates a workable voltage. The signals must be separated for the calculation, so they are demultiplexed using the PWM signal driving the LEDs. After demultiplexing, the signals have

even more noise, so the high frequency noise must be removed using a low pass filter. A second order low pass filter is able to remove the majority of the high frequency noise. After the low pass filtering, the signal must be passed through a high pass filter with an extremely low cutoff frequency to maintain the pulse signal, but remove the DC offset. This is because the DC offset is constantly changing. However, the Arduino is unable to read negative voltages, so a DC offset must be added. Additionally, the signal has few divisions for the Arduino to read, so it must be amplified. After all of these steps, the signal is ready to be read by the Arduino.

Sensor Software

Once a filtered signal is received from the red LED and IR LED, the Arduino can now properly calculate HR and SpO_2%. The Arduino, which is powered by a standard USB 3 port, utilizes three external libraries, seven I/O ports, and six functions to complete the required calculations and display the results.

The Adafruit BusIO, Adafruit GFX, and Adafruit SSD1306 libraries are the basis of the connections and graphical components between the IIC OLED. They allow for the display of all the calculated data as well as the HR signal. When deciding which libraries were necessary, there were no other possible options that would work well with the given components.

5V, which is supplied by the Arduino, is required to power the IIC OLED. Additionally, GND is provided by the Arduino. An interesting note is that the Arduino is also grounded by the AD2 to reduce noise on data between the two. In terms of data transmission, the Arduino utilizes Analog pins A0, A1, A4, and A5. A0 and A1 take in the red LED signal and IR LED signal, respectively. While, the A4 pin acts as the clock line (SCL) and the A5 pin acts as the data line (SDA). The voltage supply and GND are standardized choices, as well as the A4 and A5 pins. The data transmission pins, A0 and A1, however, were chosen arbitrarily.

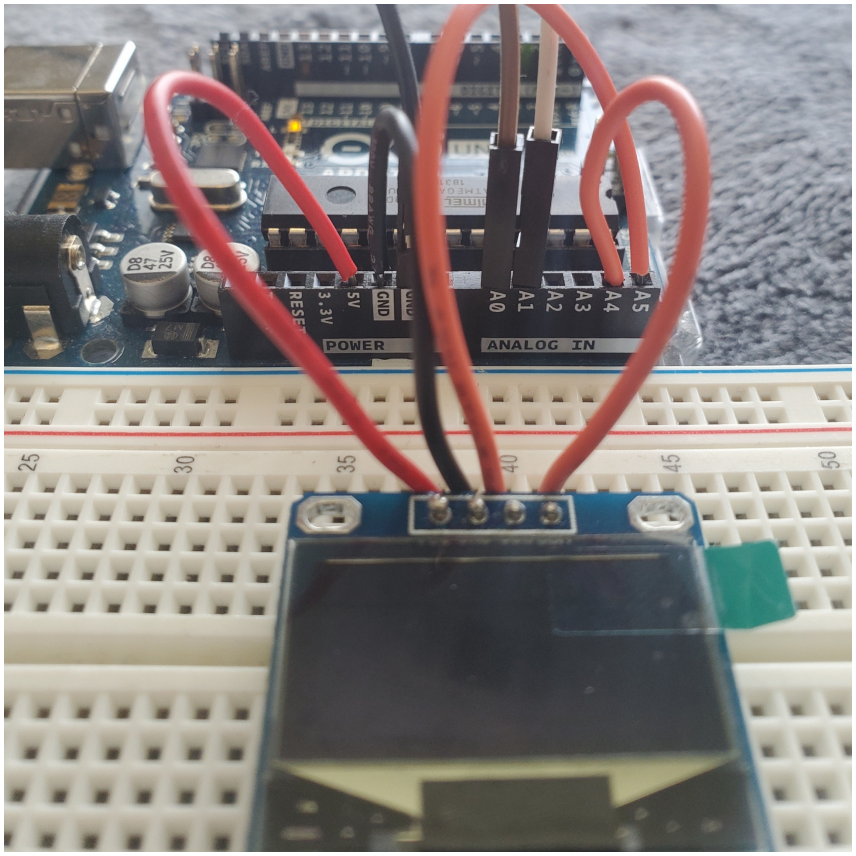

Figure 4: IIC Display Connected to Arduino

The functions used are as follows: setup, loop, calculateHR, calculateSpO2, calculatePI, and displaySetup. The setup function will begin serial transmission and initialize the display. The loop function will continuously call calculateHR, calculateSpO2, calculatePI, and displaySetup, until power to the Arduino is disabled. The remaining functions calculate the HR, SpO_2%, PI%, and display the results. Deciding on how to format the code was one of the most difficult portions of the project. An initial option discussed was a first-in first-out approach, where data would be constantly analyzed and calculated. However, ultimately, it was decided to have the code run on six second intervals, for the highest level of accuracy.

Detailed Design

Sensor Hardware

LED Driver

The Nellcor Ds-100A SpO2 sensor contains a red and IR LED, which need to be supplied with power. When greater than 15mA flows through an LED for an extended period of time, the LED has a chance of failing. Both LEDs are required for calculation of SpO_2, so either failing would render the sensor useless. The purpose of the LED driver is both to supply power and to restrict the current flowing to the diodes. Since the LEDs are anti-parallel, as shown in Figure 5, only a single current limiting resistor is needed. The LEDs can be powered using a general-purpose input/output (GPIO) swapping the pins between LOW and HIGH configurations. The circuit should draw the maximum amount of current possible to produce the maximum signal possible at the photodiode.

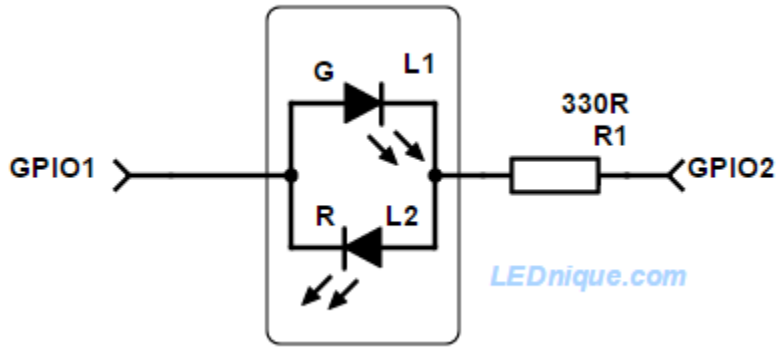

Figure 5: Schematic of LED Driver.[5]

Each LED color has different characteristics causing varying currents to be drawn depending on which LED is forward biased, as observed in Figure 6. Load line analysis was done for 270Ω and 330Ω resistances as shown in Figure 6. The results of the load line analysis can be seen in Table 1 for each resistor and LED. The analysis showed that both resistors would theoretically limit the current properly, but the 270Ω resistor would result in around 14.5mA, so both resistors were tested on the sensor. Integrating the resistors into the circuit shows 12.2mA is drawn when forward biased with the 270Ω resistor, which gives proper current limitation. The current through the red LED was also below the 15mA limit at 10.6mA. The current measurements are shown in Table 2. Since the 270Ω resistor limits the current below 15mA while maintaining a high current, it was used in the LED driver circuit. The 330Ω resistor limited the current properly, but did not maximum the current which, so it was not used.

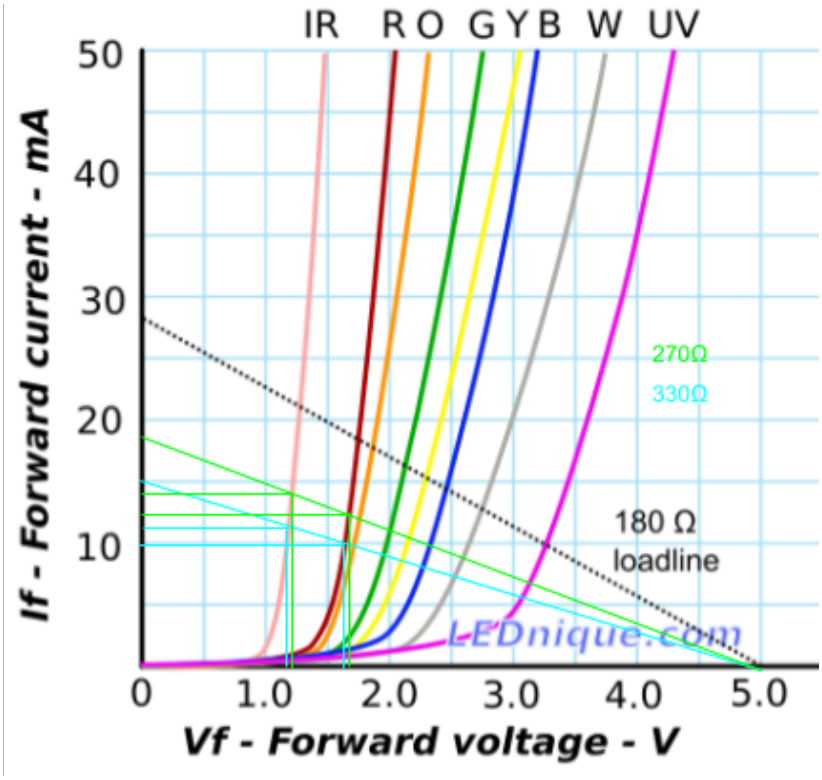

Figure 6: Load line analysis using a 270Ω and 330Ω resistor with a 5V source.[6]

Resistor	V_{IR}	I_{IR}	V_R	I_R
330Ω	1.15V	12mA	1.60V	10mA
270Ω	1.20V	14.5mA	1.65V	12.5mA

Table 1: Load-line analysis of IR and R LEDs using 330Ω and 270Ω resistors.

Resistor	V_{IR}	I_{IR}	V_R	I_R
330Ω	1.23V	10.3mA	1.73V	8.98mA
270Ω	1.24V	12.2mA	1.75V	10.6mA

Table 2: Implementation of 330Ω and 270Ω resistors on IR and R LEDs.

Transimpedance Amplifier

The output of the sensor comes from a photodiode, which converts light from the LEDs into current. However, the current is in the microamp range, so it must be amplified into a readable voltage signal through a transimpedance amplifier. The amplifier circuit and output voltage equation is shown in Figure 7. Since the output current will be in the microamp range a 680kΩ resistor should replace R_f in Figure 7 to bring the voltage to the 100mV range. The signal is not

ready to be read by the Arduino, however, since there is large amounts of noise hiding the pulse, shown in Figure 8, and the IR and Red signals need to be separated.

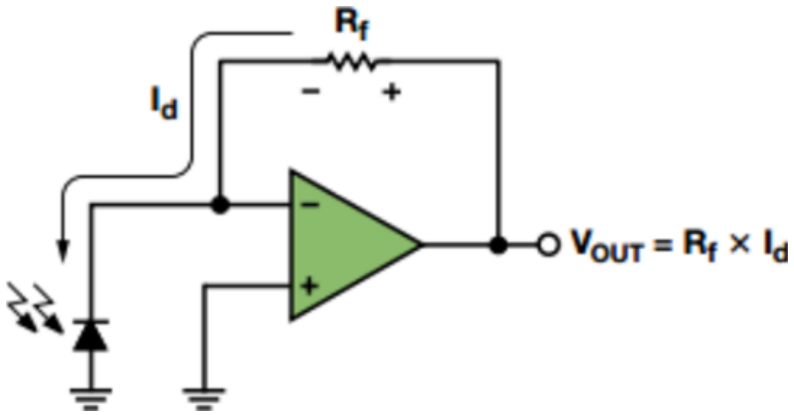

Figure 7: Transimpedance amplifier and output equation.[7]

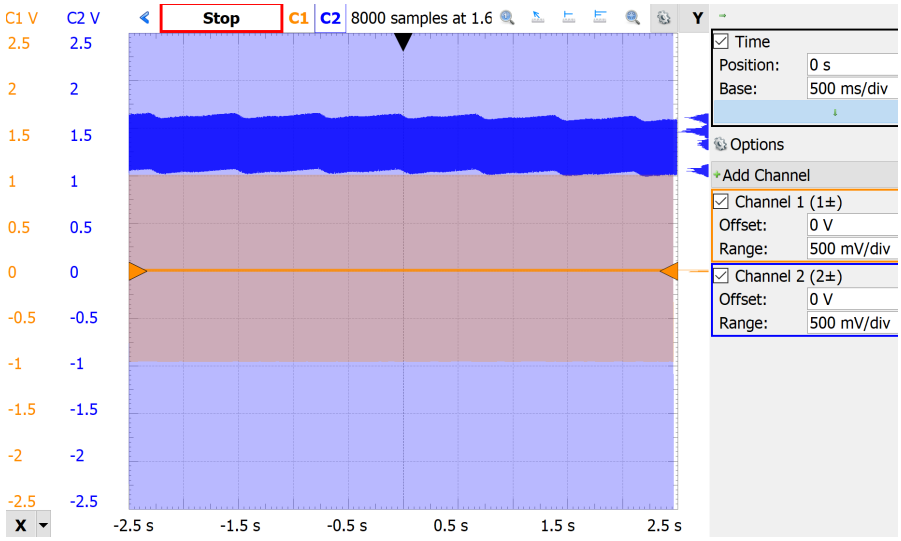

Figure 8: Output (blue) and input (orange) of transimpedance amplifier.

Multiplexer

The multiplexer is the key component to separate the red and IR LED signals. The filtering circuits contain several capacitors, which have a capacitive memory based on their charge. When turned on the infrared and red LEDs have two distinct signals, causing the capacitors to charge to different levels. If the output of the photodiode were fed through a single filtering network, then there would be times without any usable data due to the capacitors charging or discharging. This means that a switch must be used to swap networks at the same time the LEDs are switched. This can be done using a multiplexer to demux the signal from the transimpedance amplifier.

In the physical circuit, the transimpedance amplifier output was input into the mux, while the PWM signal was replaced with a 1000Hz PWM signal, which is also connected to the IR LED cathode. The 1000Hz signal was chosen because the signal becomes reduced after this amount

due to the switching. Additionally, a ground connection was added on the B channel, which is connected to the system when not powered. The output of the network is shown in Figure 10, with the infrared LED on channel 1 and the red LED on channel 2. This schematic for the multiplexer integration is shown in Figure 9.

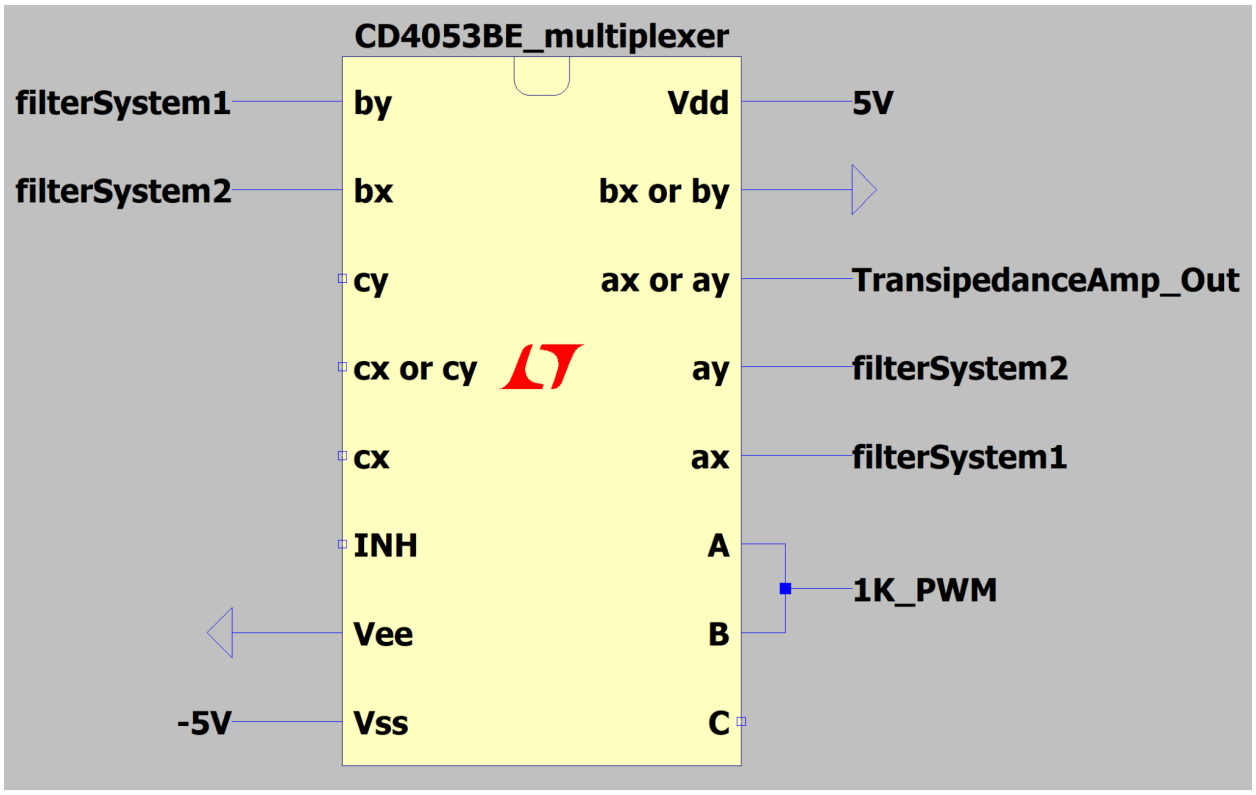

Figure 9: Schematic of the multiplexer circuit.

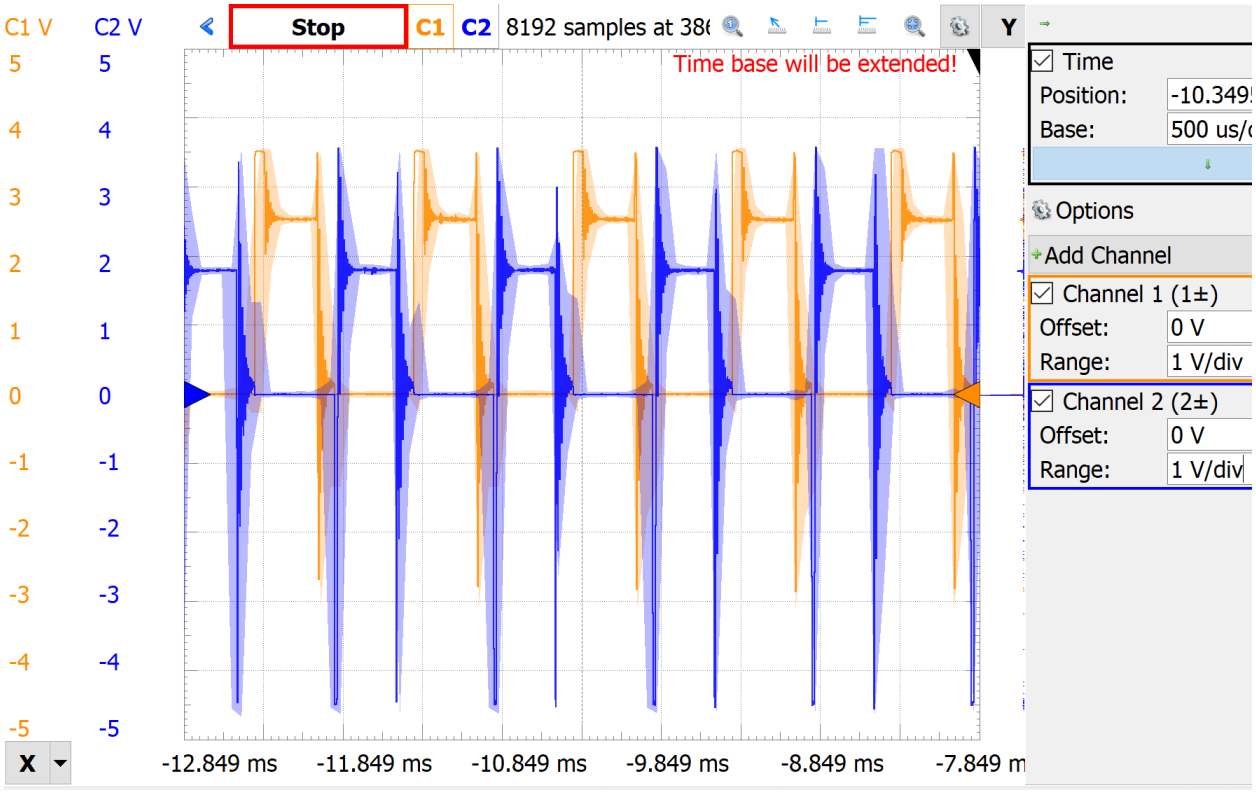

Figure 10: Output of multiplexer, infrared LED on channel 1 and the red LED on channel 2.

Low Pass Filtering

As shown in Figure 10, the signal has large amounts of noise hiding the AC pulse signal, so it requires processing. The primary source of noise is likely 60Hz noise from other electrical devices since it is used in the power grid. Another source of noise comes from switching at 100Hz. Additionally, the pulse is generally no more than 100bpm, which works out to around 1.67Hz. To ensure the pulse is maintained and the noise is filtered, a low-pass filter with -3dB at 6Hz and -40dB at 60Hz was created, shown in Figure 11. These parameters should be sufficient to remove the switching and environmental noise from the signals. The design of the circuit was created using a filter wizard, then optimized to readily available components. Prior to integrating the filter an AC sweep was conducted, which showed -2.7dB at 6Hz and -60.4dB at 60Hz, as seen in Figure 12. These results exceeded the standards set, so it was implemented in the circuit.

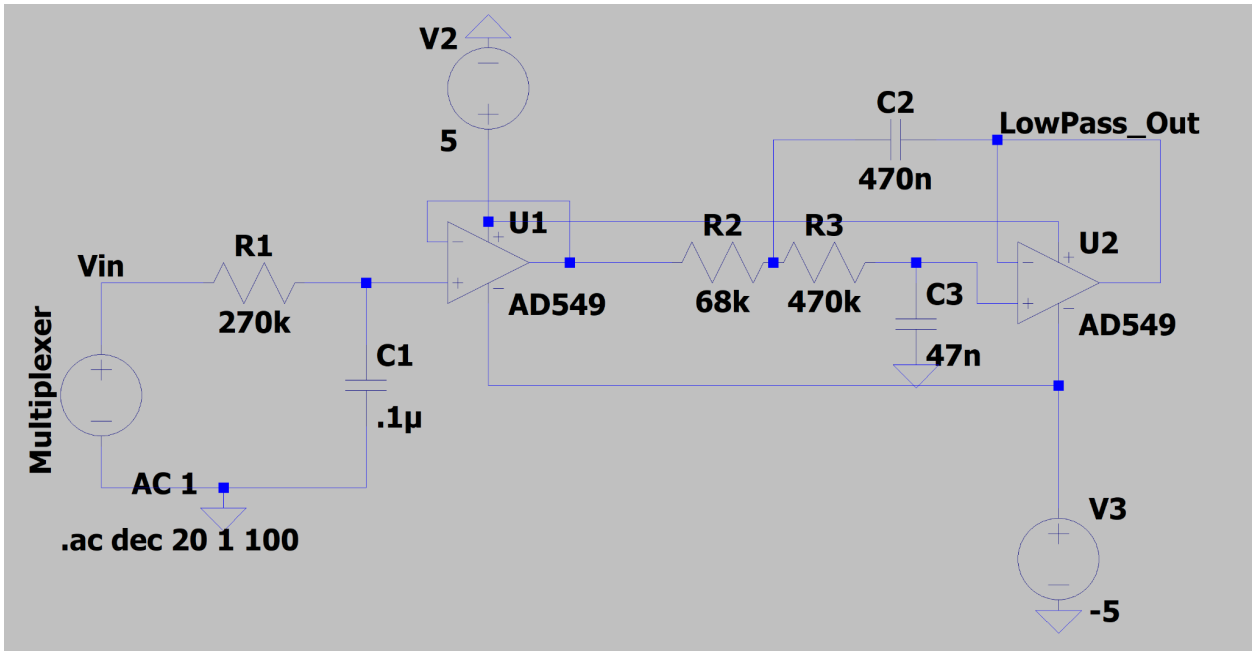

Figure 11: Schematic of low pass filter.

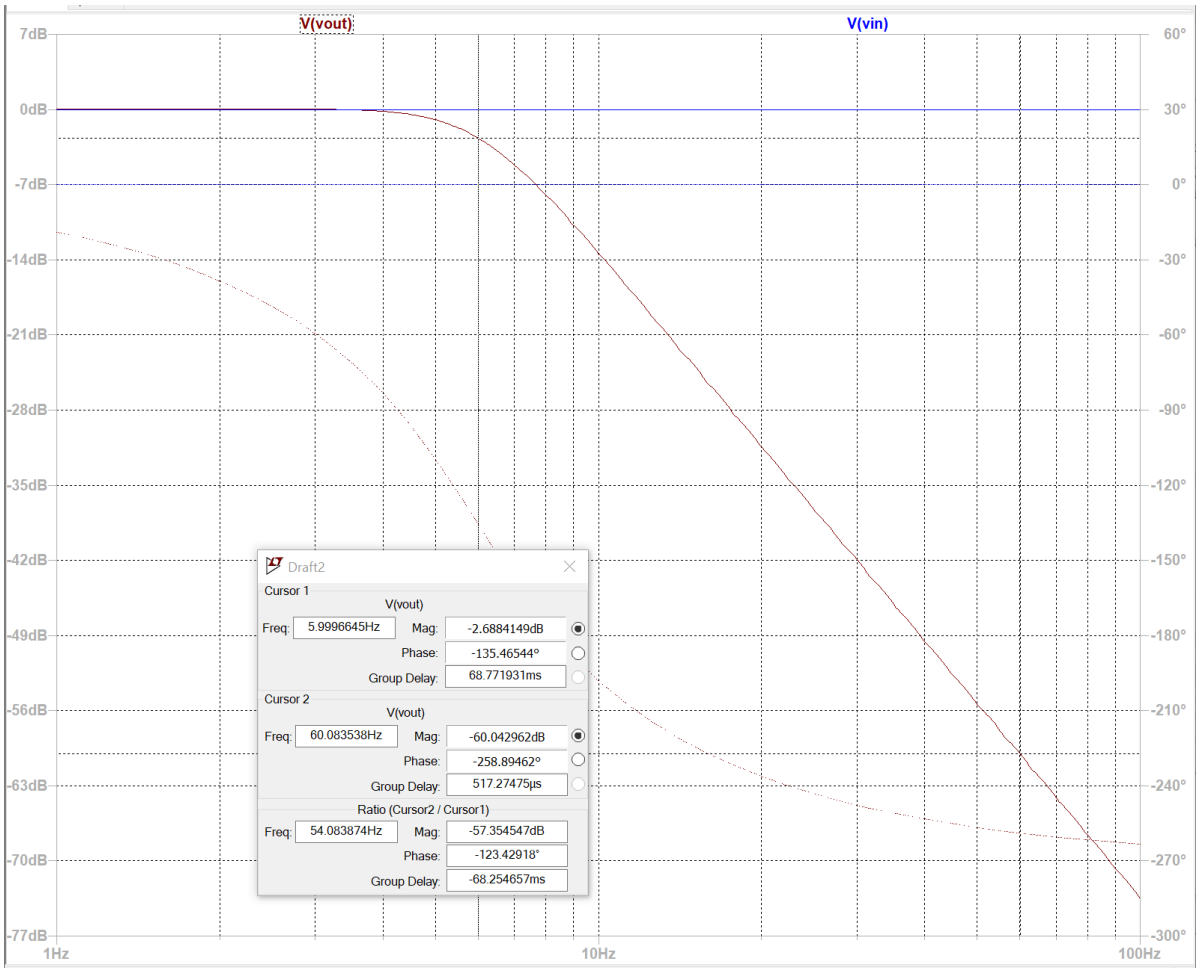

Figure 12: Simulated AC frequency response of filtering system.

When implemented in the physical circuit the filter showed -2.8dB at 6Hz and -59dB at 60Hz, shown in Figure 13. When connected to the multiplexer output, the filtered signal shows very little noise and a clear pulse, shown in Figure 14. The filtered signal is only slightly out of phase with the input signal, which is good for real-time readings. However, the pulse will still be unable to be read by the Arduino Uno since its analog inputs can only detect 4.9mV increments and the signal is around 10mV crest to trough. This means that the AC signal must be amplified.

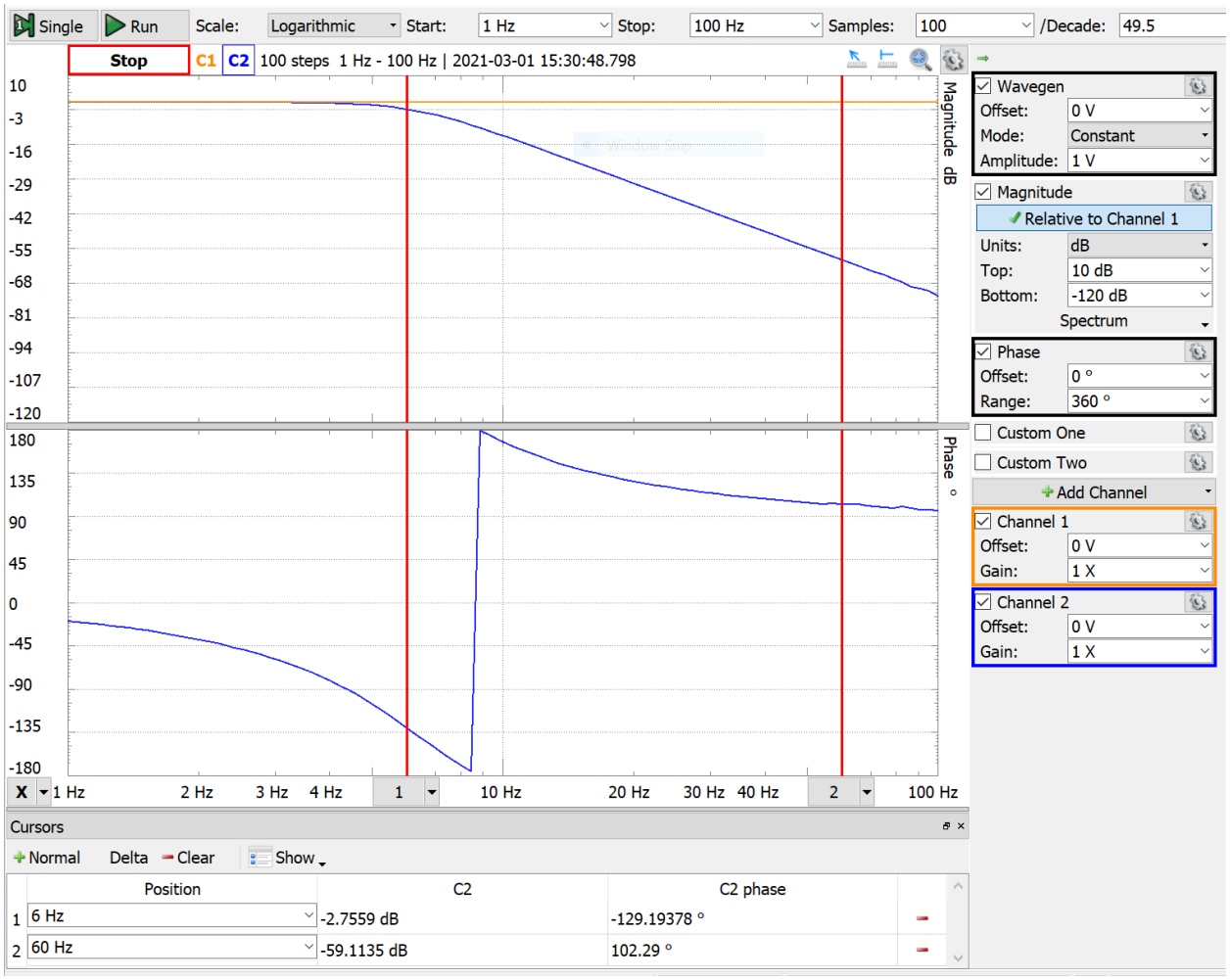

Figure 13: Network analysis of physical filtering circuit.

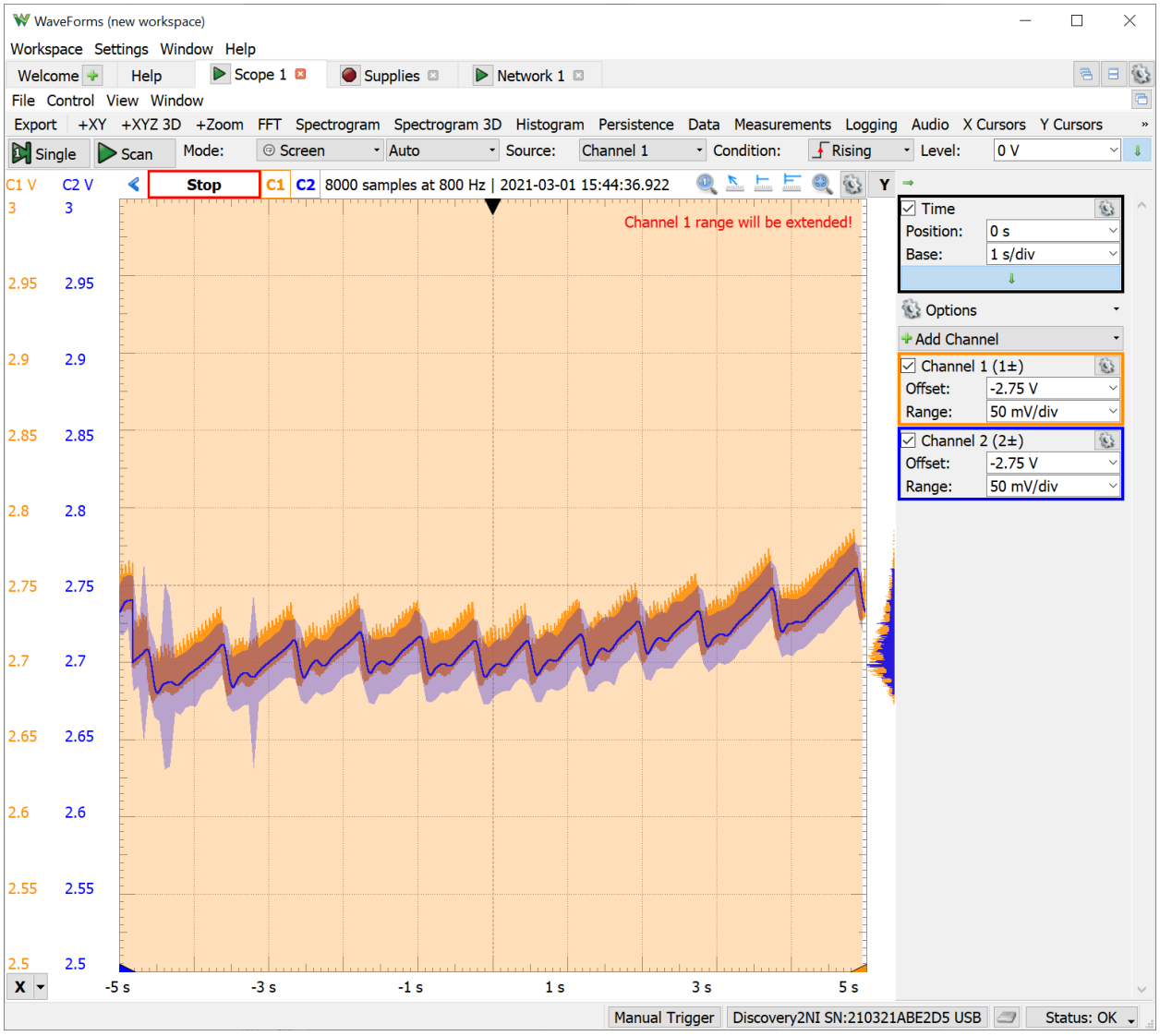

Figure 14: Filtered output, blue, against non-filtered input signal, orange.

High Pass Filtering, DC Offset, and Gain

The filtered AC signal is around 50mV peak-to-peak, which gives 10 data points since the Arduino can only read 4.9mV differences. This gives a need to amplify the AC signal to give a larger range of voltages, 500mV will give a suitable voltage range. Additionally, the output is constantly changing its DC offset, which will give problems when attempting to calculate the AC signal peak, therefore the DC offset needs to be removed.

Buffered High-Pass Filter

The desired cutoff frequency is 0.1 Hz, which requires an extremely high capacitance and resistance, so the best cutoff frequency is one using the largest resistor, 1MΩ, and largest mylar film capacitor, two 0.47uF in parallel. This circuit can be seen in Figure 15. Using the

components listed, the circuit results in a -3dB gain at 168mHz and -60dB at 168uF, shown in Figure 16. When implemented into the circuit the DC offset was properly removed with the signal intact, shown in Figure 17.

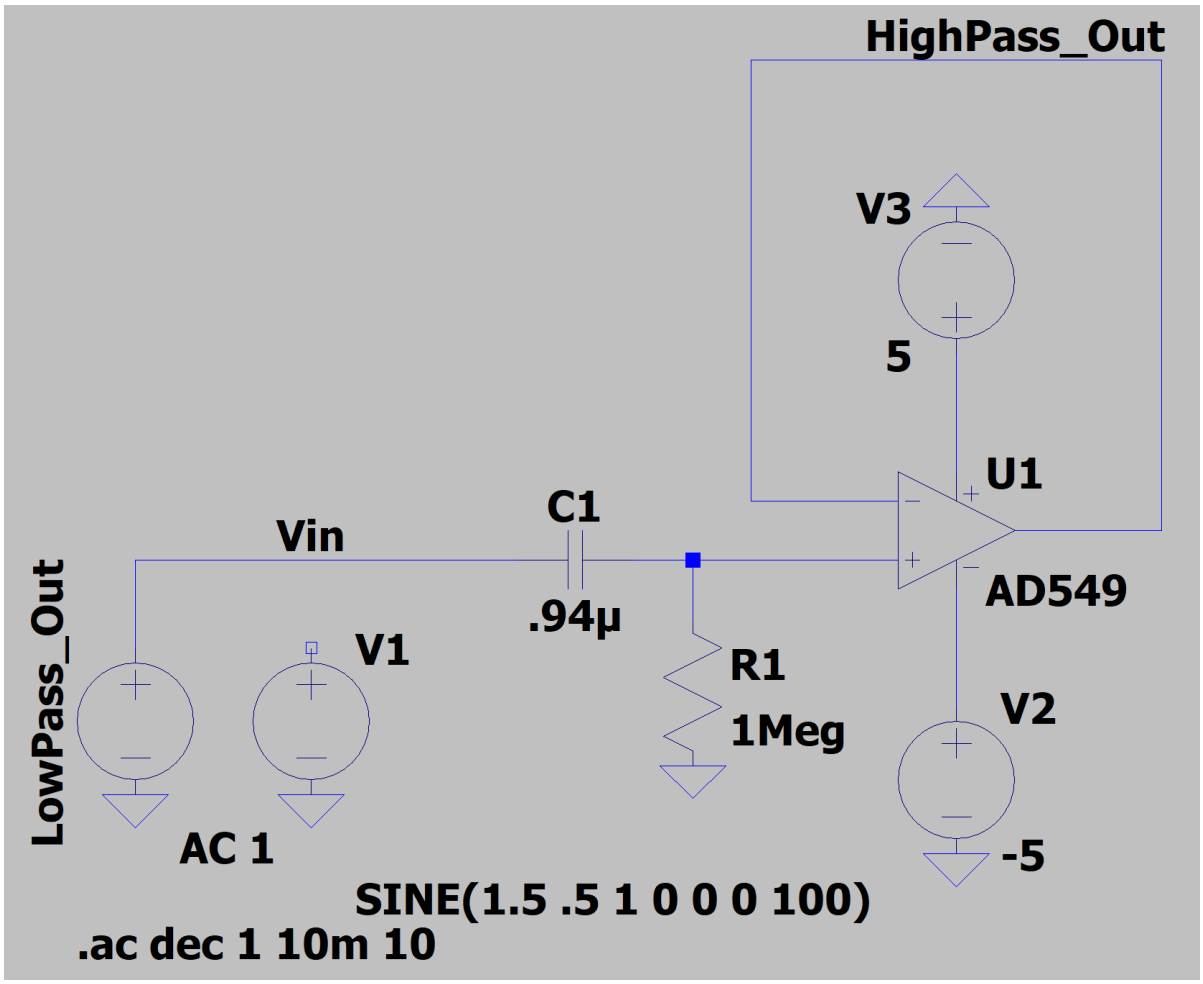

Figure 15: High-pass filter to remove DC offset

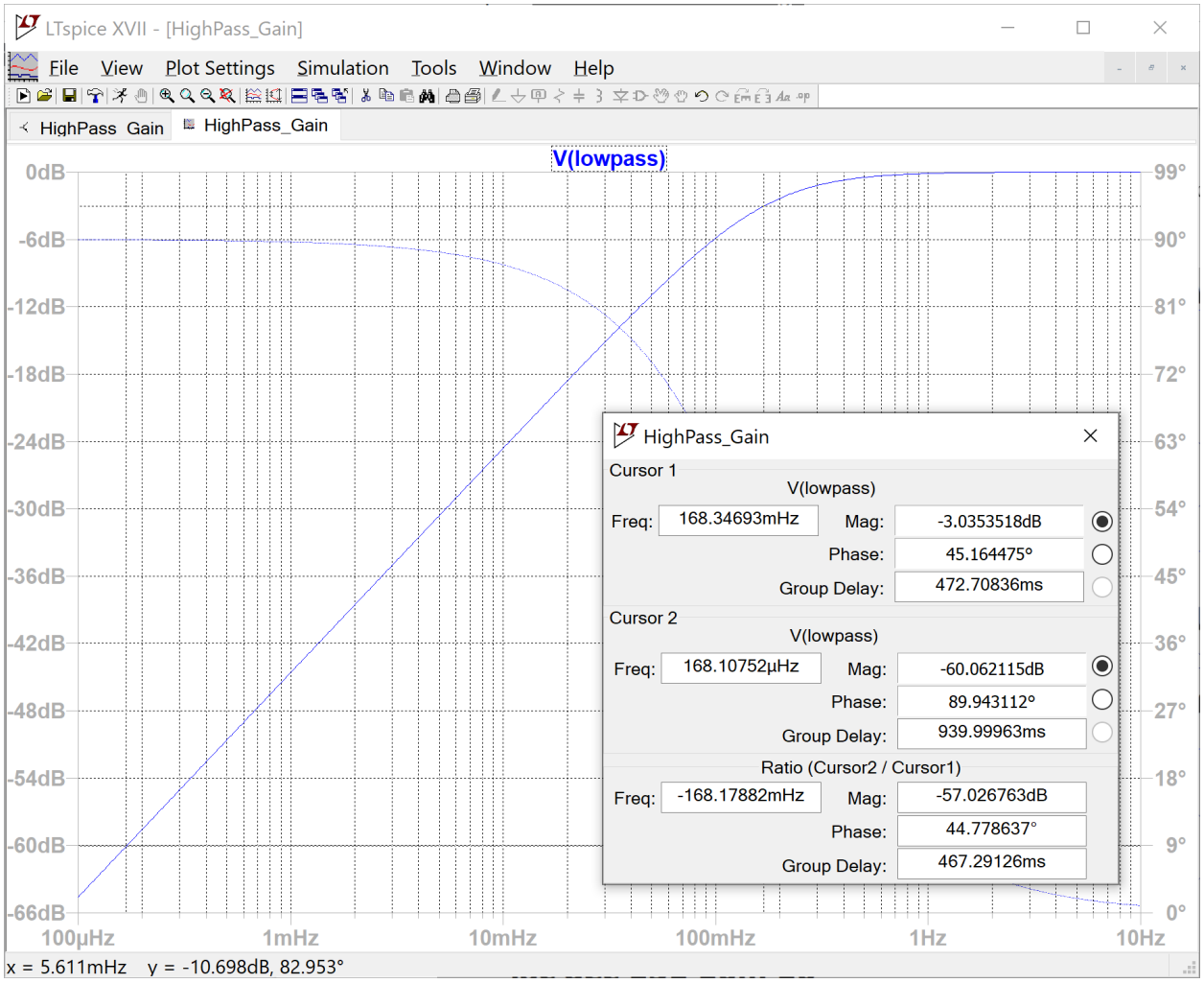

Figure 16: Frequency response of high-pass filter in Figure 5.

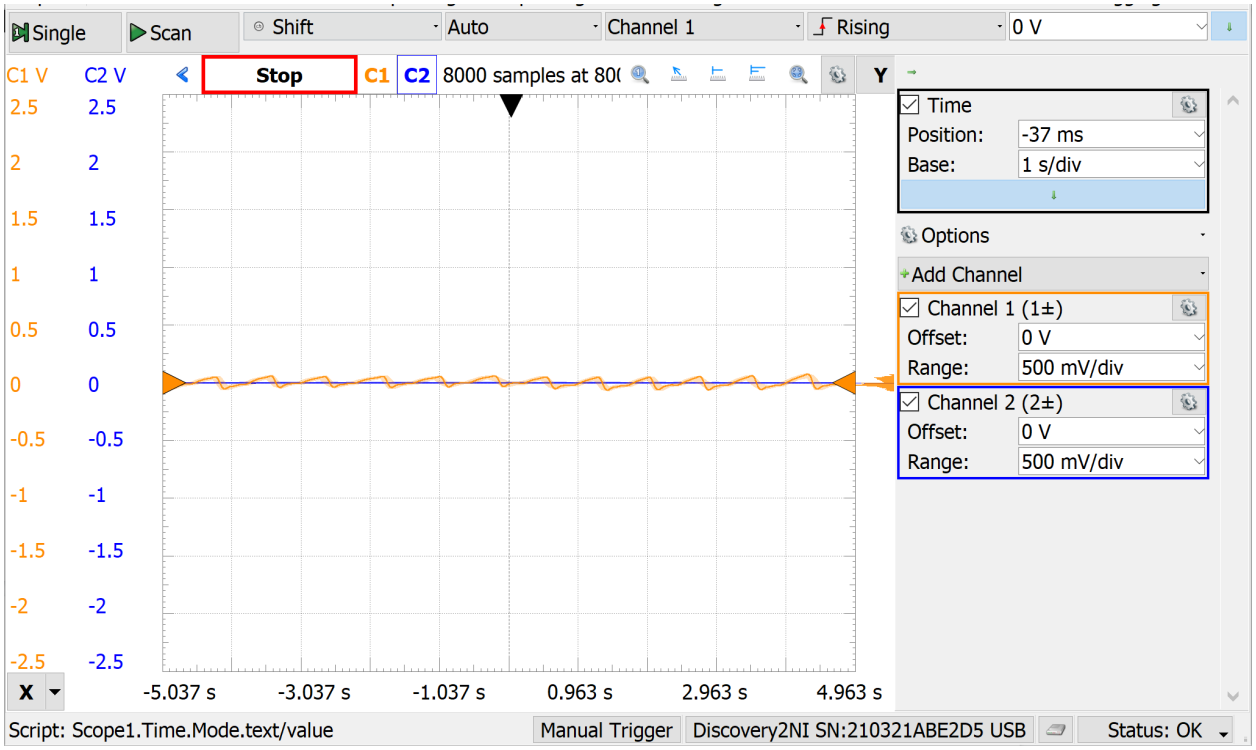

Figure 17: Input (orange) and output (blue)of high-pass filter.

Gain and DC Offset

The high-passed AC signal is around 50mV peak-to-peak, which gives around 10 data points into the Arduino, since the Arduino can only read 4.9mV differences. This gives a need to amplify the AC signal to give a larger range of voltages, 500mV will give a suitable voltage range. Additionally, the output has no DC offset, which will give problems when attempting to read the signal as the Arduino can not read negative voltages. Therefore, a new DC offset will need to be added, around 1.5V is suitable. These requirements can be achieved using a differential amplifier, resulting in a signal with a 250mV amplitude riding on a 1.5V bias.

The differential amplifier is governed by the following equation: $\frac{R_f}{R_i}(V_1 - V_2)$. Since the signal is currently 50mV and the desired signal is 500mV, a gain of 10 is needed. Therefore, the ratio between R_f and R_i needs to be 10. This is easily achieved using a R_f value of 10kΩ and a R_i value of 1kΩ. V_1 will be the signal from the high-pass filter, so V_2 must be the DC offset signal. Since a positive DC offset is wanted and the V_2 is subtracted from V_1, V_2 must be a negative voltage. With a gain of 10 and a desired voltage of 1.5V, V_2 must be -0.15V. The -0.15V can be achieved using a voltage divider with thevenin voltage source. This is required to reduce the number of resistors and interference by other components in the circuit. To make the thevenin source, the voltage divider must output -1.5V and have a thevenin resistance of 1kΩ.

These parameters should result in the proper signal output. The schematic is shown in Figure 18. When the circuit is simulated, it results in a DC bias of 1.5V with a 500mV peak-to-peak voltage as intended, shown in Figure 19. When connected to the high-pass filter's output, the signal is riding on 1.5V and generally 500mV peak-to-peak, this will vary depending on the user's heartbeat. The signal in Figure 20is perfectly acceptable since it is a positive voltage and gives many sampling points for the Arduino.

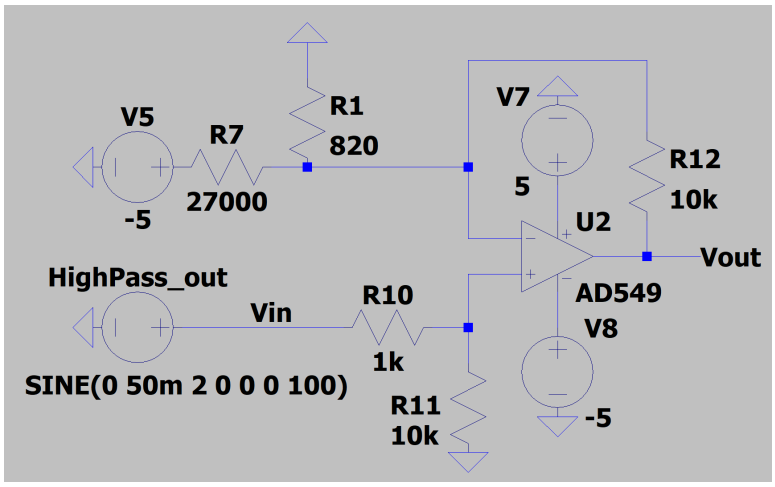

Figure 18: Schematic of implemented differential amplifier.

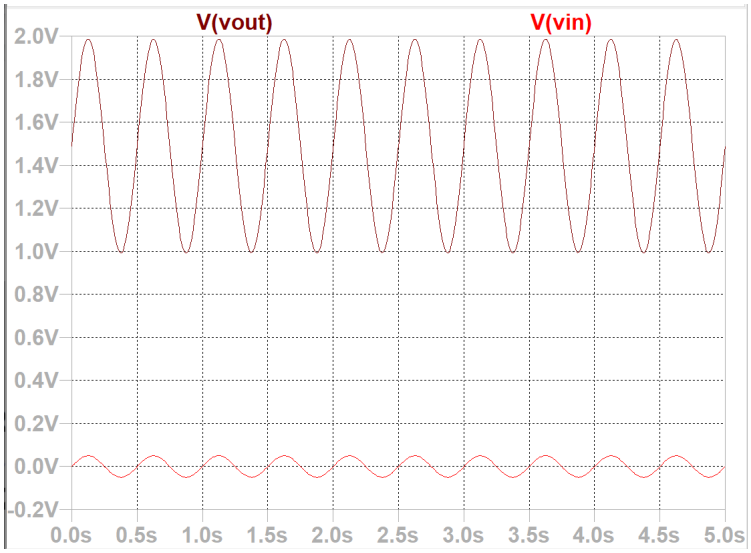

Figure 19: Simulated output of differential amplifier.

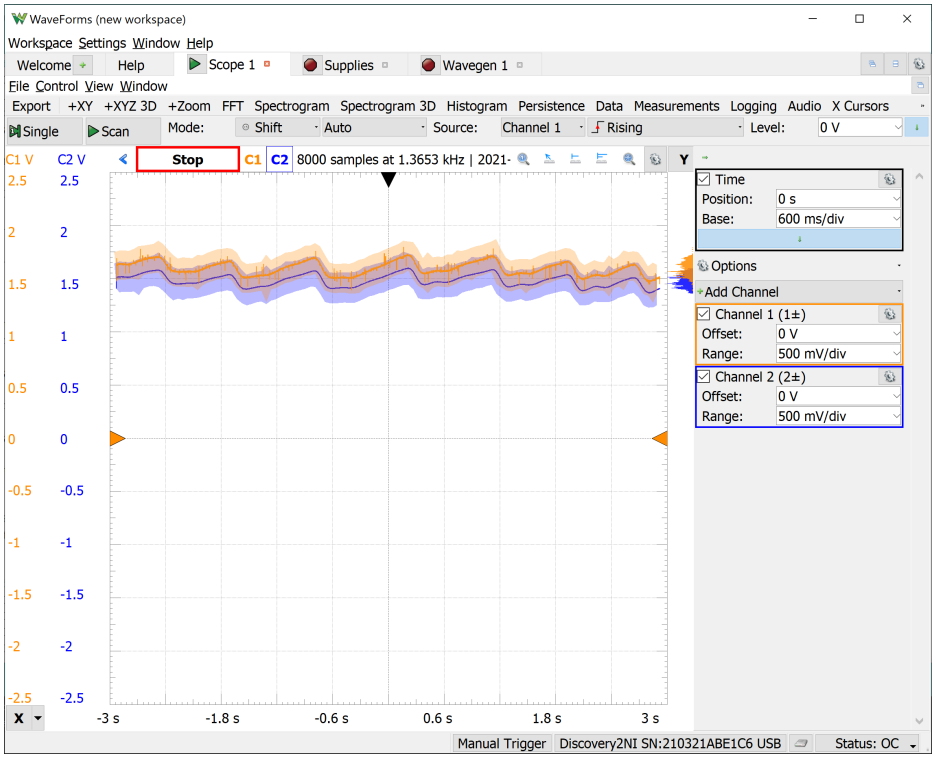

Figure 20: Actual output of differential amplifier with IR and Red LED inputs.

Alert Speaker

A typical feature of devices monitoring vitals is some kind of alert noise. A high frequency noise is desired when SpO2 is at a low level. In order to create a suitably loud sound, the NJM4556A Dual High Current Op-Amp can be used. This op-amp contains two op-amps that output an extremely high current. The speaker is rated 8Ω and 0.5W, so no current above 250mA can be provided to the speaker. 1000Hz is a typical alert sound since it is generally annoying to the human ear and many ages are able to hear the signal, so that is the frequency used to drive the speaker. The volume should also be controlled, which can be done using a potentiometer. The amplification system described is shown below in Figure 21. When implemented the op-amps supplied 157.8mA RMS, which provided an adequately loud and alerting signal.

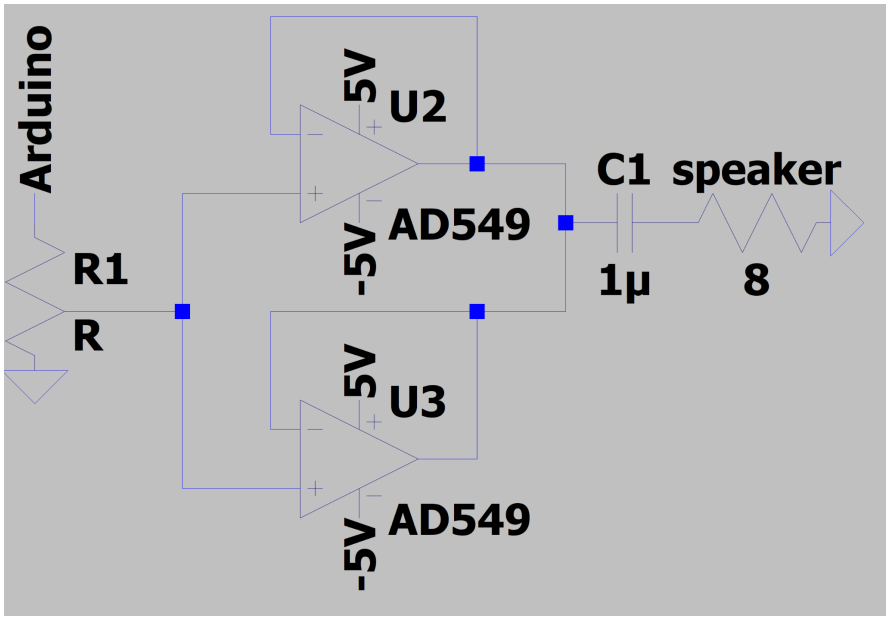

Figure 21: Schematic of speaker amplifier circuit.

Sensor Software

Setup & Loop Function

```
1 void setup() {
2  Serial.begin(9600);
3  displaySetup(0, 0, 0, {});
4 }
```

```
1  void loop() {
2    int hrData[300];
3    int irData[300];
4    for (int i = 0; i < 300; i++) {
5      hrData[i] = analogRead(0);
6      irData[i] = analogRead(1);
7      delay(20);
8    }
9    int HR = calculateHR(hrData);
10   int SpO = calculateSpO2(hrData, irData);
11   float pulseIntensity = calculatePI(hrData);
12   displaySetup(HR, SpO, pulseIntensity, hrData);
13 }
```

The setup function begins serial transmission for debugging purposes. Additionally, it initializes the display.

The loop function continuously collects data and calls all required functions. Lines **4** to **8** read in red LED and IR LED data from analog pins A0 and A1, respectively. This is run at an interval of 300 data points every 6 seconds, for a speed of 50 data points per second. The calculations are run at six second intervals to get enough data to be quick and accurate. Lines **9** to **11** call the calculation functions for HR, SpO_2%, and PI%. Finally, line **12** displays the data on the IIC OLED.

HR Function

```
1  int calculateHR(int hrData[]) {
2     int peakDiff[20] = {};
3     int peakDiffCounter = 0;
4     int peakDiffIndex = 0;
5     for (int i = 3; i < 297; i++) {
6       if (hrData[i - 1] <= hrData[i] && hrData[i + 1] <= hrData[i] && hrData[i - 2] <= hrData[i] && hrData[i +
2] <= hrData[i] && hrData[i - 3] <= hrData[i] && hrData[i + 3] <= hrData[i] && hrData[i] != 0 && hrData[i] >
300) {
7         if (peakDiffCounter > 20 && peakDiffCounter < 60) {
8           peakDiff[peakDiffIndex] += peakDiffCounter;
9           peakDiffIndex++;
10        }
11        peakDiffCounter = 0;
12      } else {
13        peakDiffCounter++;
14      }
15    }
16
17    int peakDiffSum = 0;
18    int peakDiffMin = peakDiff[0];
19
20    for (int i = 1; i < peakDiffIndex; i++) {
21    if (peakDiff[i] < peakDiffMin)
22      peakDiffMin = peakDiff[i];
23    }
24    for (int i = 0; i < peakDiffIndex; i++) {
25      if (peakDiff[i] != peakDiffMin)
26        peakDiffSum += peakDiff[i];
27     }
28     peakDiffSum /= (peakDiffIndex - 1);
29     peakDiffSum = 60*(1/(peakDiffSum/50.0));
30     Serial.print("HR: ");
31     Serial.println(peakDiffSum);
32     return peakDiffSum;
33 }
```

The calculateHR function determines the user's HR based on input from the red LED using basic signal properties. Knowing that a 1 Hz signal has a period of 1 second, the period of the input signal in BPM can be calculated by inverting the frequency of the red LED input signal and multiplying by 60 seconds.

To determine the frequency of the red LED input signal lines **5** through **15** find where a peak exists and store the data in the array peakDiff. The logic statement on line **6** has parameters to determine the location of the peak. The logic statement on line **7** determines if the peak can exist. If a peak is too close together or far apart, it is deemed impossible. Lines **17** to **28** will remove any outlying values and average the distance between peaks. Line **29** calculates the HR in BPM. Knowing data is taken in every 50 times per second, we can divide the average of the peaks by 50, then invert to a frequency, and multiply by 60 to get the HR. Lines **30** and **31** print the resulting data to the serial monitor for debugging purposes. Finally, line **32** returns the HR value to be displayed.

SpO2% Function

```
1  int calculateSpO2(int hrData[], int irData[]) {
2    float minHR = hrData[0];
3    float minIR = irData[0];
4    float maxHR = hrData[0];
5    float maxIR = hrData[0];
6    double avgHR = 0;
7    double avgIR = 0;
8
9    for (int i = 1; i < 300; i++) {
10     if (hrData[i] < minHR)
11       minHR = hrData[i];
12     if (irData[i] < minIR)
13       minIR = irData[i];
14     if (hrData[i] > maxHR)
15       maxHR = hrData[i];
16     if (irData[i] > maxIR)
17       maxIR = irData[i];
18   avgHR += hrData[i];
19   avgIR += irData[i];
20   }
21
22   float ppHR = maxHR - minHR;
23   float ppIR = maxIR - minIR;
24
25   avgHR /= 300.0;
26   avgIR /= 300.0;
27
```

```
28  float R = (ppHR/avgHR)/(ppIR/avgIR);
29
30  int SpO = ((110) + (-12*R));
31  Serial.print("SpO2: ");
32  Serial.println(SpO);
33
34  if (SpO < 93)
35    analogWrite(5, 100);
36  else
37    analogWrite(5, 0);
38
39  return SpO;
40 }
```

The calculateSpO2 function determines the user's SpO_2 based on input from the red LED and IR LED using researched equations. The main portion of calculateSpO2 is determining the ratio input for the final SpO_2 equation as shown in figure 22.

$$R = \frac{\frac{AC_{red}}{DC_{red}}}{\frac{AC_{ired}}{DC_{ired}}}, \quad SpO_2 = ((110) + (-12 \cdot R))$$

Figure 22: SpO2 Ratio Equation[2] and Final Equation[3]

Lines **9** to **26** determine the DC and AC portions of the red LED and IR LED input signal. To find DC components, the average of 300 data points is taken. To find AC components, first the maximum and minimum values of the given signal are found. Then, the minimum is subtracted from the maximum. This will give the AC or peak-to-peak voltage. Line **28** calculates the ratio equation with the derived values. Line **30** calculator the SpO_2 as a percentage. Lines **31** and **32** print the resulting data to the serial monitor for debugging purposes. Lines **34** to **37** write a PWM signal to the warning speaker. Finally, line **39** returns the SpO_2 value to be displayed.

PI% Function

```
1  double calculatePI(int hrData[]) {
2    float minHR = hrData[0];
3    float maxHR = hrData[0];
4    double avgHR = 0;
5
6    for (int i = 1; i < 300; i++) {
7      if (hrData[i] < minHR)
8        minHR = hrData[i];
9      if (hrData[i] > maxHR)
10       maxHR = hrData[i];
11     avgHR += hrData[i];
```

```
12  }
13  float ppHR = maxHR - minHR;
14
15  avgHR /= 300.0;
16  avgHR -= 82;
17
18  float pulseIntensity = ((ppHR/avgHR)*100.0)/10.0;
19  Serial.print("PI: ");
20  Serial.println(pulseIntensity);
21  return pulseIntensity;
22 }
```

The calculatePI function determines the user's PI based on input from the red LED using researched Equations. "The ratio of the AC component to the DC component, expressed as a percentage, is known as the (peripheral) perfusion index (PI) for a pulse, and typically has a range of 0.02% to 20%.[4]"

Lines **6** to **15** determine the DC and AC portions of the red LED input signal. To find DC components, the average of 300 data points is taken. To find AC components, first the maximum and minimum values of the given signal are found. Then, the minimum is subtracted from the maximum. This will give the AC or peak-to-peak voltage. Line **16** subtracts 0.4V. This is because when no voltage is given to the circuit there is an innate 0.4V. Line **18** calculates the PI as a percentage and then divides by the same factor as the signal was initially amplified. Lines **19** and **20** print the resulting data to the serial monitor for debugging purposes. Finally, line **22** returns the PI value to be displayed.

Display Function

```
1  void displaySetup(int HR, int SpO, float pulseIntensity, int hrData[]) {
2    if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
3      Serial.println(F("SSD1306 allocation failed"));
4      for(;;); // Don't proceed, loop forever
5    }
6
7    display.clearDisplay();
8
9    display.setTextSize(1);
10   display.setTextColor(WHITE);
11   display.setTextWrap(false);
12
13   display.setCursor(0, 25);
14   display.print("Heart Rate: ");
15   display.print(HR);
16   display.print(" BPM");
```

```
17
18  display.setCursor(0, 40);
19  display.print("SpO2: ");
20  display.print(SpO);
21  display.print("%");
22
23  display.setCursor(0, 55);
24  display.print("PI: ");
25  display.print(pulseIntensity);
26  display.print("%");
27
28  for (int i = 0; i < 128; i++) {
29    display.drawPixel(i, hrData[i]/15, WHITE);
30  }
31
32  display.display();
33 }
```

The displaySetup function displays all relevant data onto the IIC OLED.

Lines **2** to **5** are for checking if the IIC OLED is connected to the Arduino properly.. If the display doesn't respond to data being sent to the screen address, in this case 0x3C, then the function will print an error message and not turn on. Line **7** clears the previous display. Lines **9** to **26** add all relevant data from calculations to the display buffer. Lines **28** to **30** add the HR signal to the display buffer. Finally, line **32** displays all data in the buffer onto the IIC OLED.

Validation of Overall Project

The developed pulse oximeter was tested against a physical established pulse oximeter as well as the one built into the Samsung Galaxy S10+.

Figure 23: Testing Against Established Pulse Oximeter

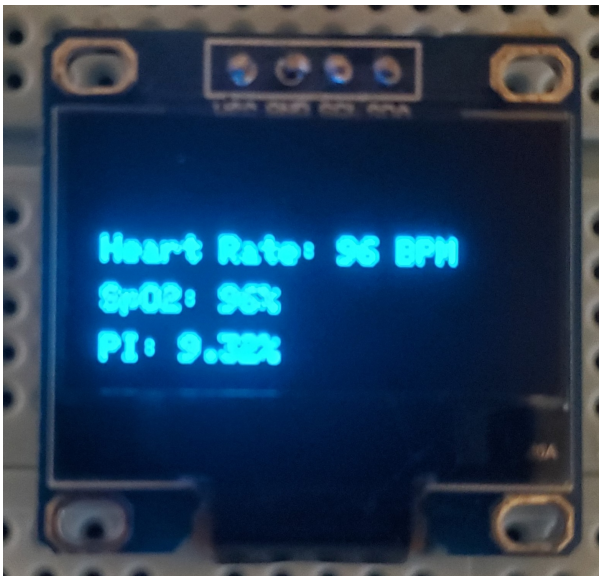

Figure 24: Testing Against Galaxy S10+ Pt1

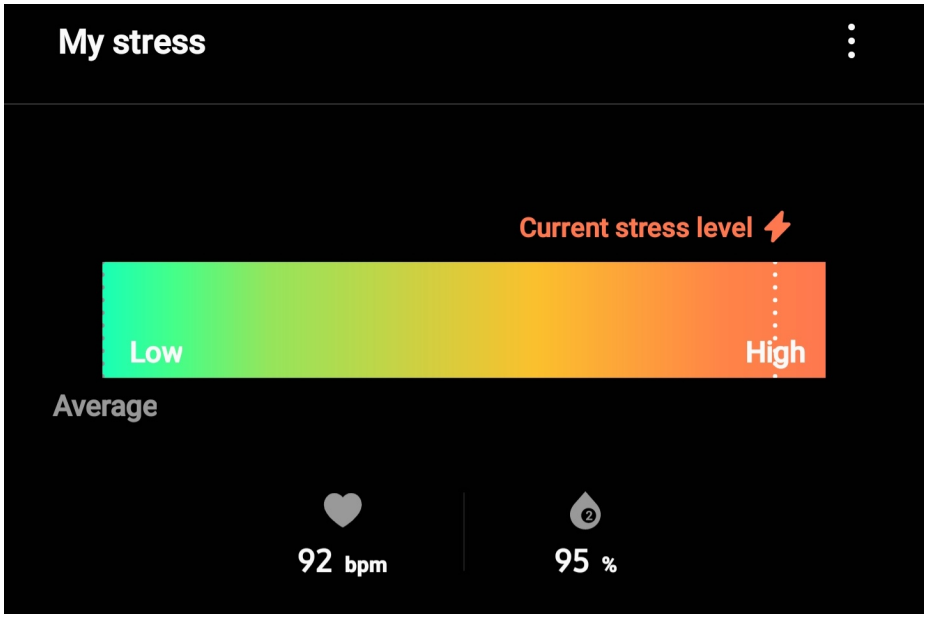

Figure 25: Testing Against Galaxy S10+ Pt2

```
21:58:07.107 -> HR: 100
21:58:07.107 -> SpO2: 95
21:58:07.142 -> PI: 4.89
21:58:13.260 -> HR: 93
21:58:13.294 -> SpO2: 95
21:58:13.294 -> PI: 4.40
21:58:19.416 -> HR: 136
21:58:19.450 -> SpO2: 94
21:58:19.450 -> PI: 4.83
21:58:25.603 -> HR: 93
21:58:25.603 -> SpO2: 103
21:58:25.638 -> PI: 3.63
21:58:31.762 -> HR: 111
21:58:31.762 -> SpO2: 102
21:58:31.796 -> PI: 4.63
21:58:37.913 -> HR: 100
21:58:37.948 -> SpO2: 98
21:58:37.948 -> PI: 4.96
21:58:44.109 -> HR: 96
21:58:44.109 -> SpO2: 94
21:58:44.109 -> PI: 9.15
21:58:50.259 -> HR: 93
21:58:50.294 -> SpO2: 98
21:58:50.294 -> PI: 4.72
```

Figure 26: Actual output to the serial monitor. Checkmarks signify proper behavior while an X shows computational error.

When testing against the established pulse oximeter, shown in Figure 23, the software output valid results within the 2% error required. An additional test was performed against the Galaxy S10+ pulse oximeter as a comparison to an application many people have access to. The Galaxy had nearly the same SpO2%, but a different heart rate. For the majority the results from the SpO2 meter were valid against these other applications, however, any amount of movement by the user while taking data produced invalid results, as shown in Figure 26.

Overall, all initial goals for the project were completed, excluding the HR graph. Over a period of two months the portion of the code that writes the graph to the display was tested and debugged in a multitude of ways over tens of hours. However, no matter what changes were made the graph code (which works perfectly independently) did not work as expected (with the rest of the code).

Conclusion

The Independent Design Project course is not meant to be a straight forward class. We are meant to apply the concepts learned in prior courses to an open-ended engineering problem. This process allows students to learn the gaps in the hard skills such as circuit analysis and coding, while also developing soft skills such as communication, documentation, and planning. Throughout the project we learned our original schedule was too ambitious. On the surface level the project was very simple, but when designing circuits and coding, we realized there were many errors in our work that needed to be solved.

What differentiates us from other groups, is that we were able to learn much about pulse oximetry, its importance, and its underlying design. On the hard skills side, we learned not to use electrolytic capacitors in signal processing applications and proper circuit design. Additionally, we learned the importance of non-blocking code and how to modify PWM frequencies of the Arduino Uno.

If we were to do this project again, we would have taken a much less ambitious approach in order to refine each component without rushing. In the final weeks of the project, we were strapped for time trying to refine the circuits and code. If Coronavirus were not around, we would have also met in person much more often to collaborate. However, the lack of in person meetings improved our online communication skills, which will help for future projects.

Overall, our group made a successful pulse oximeter. The pulse oximeter circuitry was able to output a very strong heartbeat signal. While the pulse oximeter software was able to compute the SpO2% and BPM within an error of 2%. We were able to go beyond the original goals of the project in some areas, the perfusion index measurement and alert noise, but were lacking on one of the original goals, graphing the heartbeat signal on the display. We were able to learn from this by realizing that we overextended ourselves in some areas, while ignoring others since they seemed simple.

Authorship

Introduction: Written by William, revised by Nathan.

High Level Design: Hardware written by Nathan, software written by William. Reviewed by both.

Detailed Design: Hardware written by Nathan, software written by William. Reviewed by both.

Validation: Written by William. Reviewed by Nathan.

Conclusion: Written by Nathan, revised by William.

Citations

1. https://blog.tunstallhealthcare.com.au/wp-content/uploads/2017/09/Pulse-oximeter-working-3-1024x707.png
2. https://drive.google.com/file/d/15oR4be0DGURf-FjBz7VxKxoRbTysrstL/view?usp=sharing
3. https://www.researchgate.net/publication/298204889_Design_and_implementation_of_an_oximetry_monitoring_device
4. https://en.wikipedia.org/wiki/Pulse_oximetry#Derived_measurements
5. http://lednique.com/tag/led/
6. http://lednique.com/tag/led/
7. https://www.analog.com/en/technical-articles/optimizing-precision-photodiode-sensor-circuit-design.html

Complete Code

```
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64

Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, -1);

void setup() {
  Serial.begin(9600);
  displaySetup(0, 0, 0, {});
}

void loop() {
  int hrData[300];
  int irData[300];
  for (int i = 0; i < 300; i++) {
    hrData[i] = analogRead(0);
    irData[i] = analogRead(1);
    delay(20);
  }
  int HR = calculateHR(hrData);
  int SpO = calculateSpO2(hrData, irData);
  float pulseIntensity = calculatePI(hrData);
  displaySetup(HR, SpO, pulseIntensity, hrData);
}

int calculateHR(int hrData[]) {
    int peakDiff[20] = {};
    int peakDiffCounter = 0;
    int peakDiffIndex = 0;
    for (int i = 3; i < 297; i++) {
      if (hrData[i - 1] <= hrData[i] && hrData[i + 1] <= hrData[i] && hrData[i
- 2] <= hrData[i] && hrData[i + 2] <= hrData[i] && hrData[i - 3] <= hrData[i]
&& hrData[i + 3] <= hrData[i] && hrData[i] != 0 && hrData[i] > 300) {
        if (peakDiffCounter > 20 && peakDiffCounter < 60) {
          peakDiff[peakDiffIndex] += peakDiffCounter;
          peakDiffIndex++;
        }
        peakDiffCounter = 0;
      } else {
        peakDiffCounter++;
      }
    }

    int peakDiffSum = 0;
```

```
    int peakDiffMin = peakDiff[0];

    for (int i = 1; i < peakDiffIndex; i++) {
      if (peakDiff[i] < peakDiffMin)
        peakDiffMin = peakDiff[i];
    }
    for (int i = 0; i < peakDiffIndex; i++) {
      if (peakDiff[i] != peakDiffMin)
        peakDiffSum += peakDiff[i];
    }
    peakDiffSum /= (peakDiffIndex - 1);
    peakDiffSum = 60*(1/(peakDiffSum/50.0));
    Serial.print("HR: ");
    Serial.println(peakDiffSum);
    return peakDiffSum;
}

int calculateSpO2(int hrData[], int irData[]) {
  float minHR = hrData[0];
  float minIR = irData[0];
  float maxHR = hrData[0];
  float maxIR = hrData[0];
  double avgHR = 0;
  double avgIR = 0;

  for (int i = 1; i < 300; i++) {
    if (hrData[i] < minHR)
      minHR = hrData[i];
    if (irData[i] < minIR)
      minIR = irData[i];
    if (hrData[i] > maxHR)
      maxHR = hrData[i];
    if (irData[i] > maxIR)
      maxIR = irData[i];
  avgHR += hrData[i];
  avgIR += irData[i];
  }

  float ppHR = maxHR - minHR;
  float ppIR = maxIR - minIR;

  avgHR /= 300.0;
  avgIR /= 300.0;

  float R = (ppHR/avgHR)/(ppIR/avgIR);

  int SpO = ((110) + (-12*R));
  Serial.print("SpO2: ");
  Serial.println(SpO);
```

```
  if (SpO < 93)
    analogWrite(5, 100);
  else
    analogWrite(5, 0);

  return SpO;
}

double calculatePI(int hrData[]) {
  float minHR = hrData[0];
  float maxHR = hrData[0];
  double avgHR = 0;

  for (int i = 1; i < 300; i++) {
    if (hrData[i] < minHR)
      minHR = hrData[i];
    if (hrData[i] > maxHR)
      maxHR = hrData[i];
    avgHR += hrData[i];
  }
  float ppHR = maxHR - minHR;

  avgHR /= 300.0;
  avgHR -= 82;

  float pulseIntensity = ((ppHR/avgHR)*100.0)/10.0;
  Serial.print("PI: ");
  Serial.println(pulseIntensity);
  return pulseIntensity;
}

void displaySetup(int HR, int SpO, float pulseIntensity, int hrData[]) {
  if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
    Serial.println(F("SSD1306 allocation failed"));
    for(;;); // Don't proceed, loop forever
  }

  display.clearDisplay();

  display.setTextSize(1);
  display.setTextColor(WHITE);
  display.setTextWrap(false);

  display.setCursor(0, 25);
  display.print("Heart Rate: ");
  display.print(HR);
  display.print(" BPM");

  display.setCursor(0, 40);
  display.print("SpO2: ");
```

```
  display.print(SpO);
  display.print("%");

  display.setCursor(0, 55);
  display.print("PI: ");
  display.print(pulseIntensity);
  display.print("%");

  for (int i = 0; i < 128; i++) {
    display.drawPixel(i, hrData[i]/15, WHITE);
  }

  display.display();
}
```