Wireless, At-Home Heart Monitoring:

Final Documentation

Jackson Bautch, Alex Beck, Josh O’Brien, Megan O’Donnell

Table of Contents

1 Introduction 4

1.1 The Problem Being Solved 4

1.2 High Level Design Description 4

1.3 Design Expectations 6

2 Detailed System Requirements 9

3 Detailed project description 10

3.1 System theory of operation 10

3.2 System Block diagram 11

3.3 Detailed design/operation of Electrocardiogram Subsystem 11

3.4 Detailed design/operation of Heart Rate Monitoring Subsystem 19

3.5 Detailed Design/Operation of Internet of Things Subsystem 27

3.6 Detailed Design/Operation of User Interface Subsystem 39

3.7 Interfaces 53

4 System Integration Testing 55

4.1 Describe how the integrated set of subsystems was tested. 55

4.2 Show how the testing demonstrates that the overall system meets the design requirements 55

5 Users Manual/Installation manual 55

5.1 How to install your product 55

5.2 How to setup your product 56

7.3 How the User Can Tell if the Product Is Working 58

7.4 How the User Can Troubleshoot the Product 59

6 To-Market Design Changes 60

7 Conclusions 61

8 Appendices 61

Component Data Sheets 61

Hardware Schematics 62

Complete Software Listings 62

1 Introduction

1.1 The Problem Being Solved

For our senior design project, we wanted to design a medical device that could potentially improve how healthcare is delivered to patients and expand upon the rapidly growing industry of wireless, wearable medical devices.

In the United States, cardiovascular disease is the number one killer of Americans. The clinical standard for detecting heart abnormalities is an electrocardiogram (ECG) which is a device that can record electrical activity of the heart allowing for heart health to be measured in a non-invasive manner. For patients with chronic heart problems who present with intermittent symptoms, it can be difficult to record an ECG in the clinic or hospital while the patient is presenting with symptoms as he or she may not be experiencing any problems at that point in time. This can cause a patient to go undiagnosed with a potentially life-threatening ailment.

A solution, then, is to allow the patient to wear a wireless ECG monitor throughout the day or for a period of multiple days while it can record the electrical activity of the heart whenever or wherever. With this solution, a patient can start an ECG recording whenever he or she is experiencing symptoms whether that be at work, home, or anywhere else. The hope is that this device provides greater access to much needed medical testing equipment in a relatively inexpensive manner.

1.2 High Level Design Description

Overall, the solution to the aforementioned problem involves using three gel electrodes placed in the standard right arm (RA), left arm (LA), and left leg (LL) positions to create a single ECG lead. The electrodes are able to pick up on the electrical activity present on the surface of

the skin which can then be converted into an electrical signal in a wire by clipping on ECG cables. Because the body acts as a large antenna and can pick up on many electrical signals in the person's environment, such as the 60 Hz signal used in the United State's power grid, the electric potential being measured on the skin is filled with unwanted noise and common mode signals. In order to get clean ECG data, the biopotential data was filtered and amplified which is described in more detail in Section 3 of this document.

After the biopotential data has been cleaned and amplified, a microcontroller, the ESP32-S3-WROOM-N8R8, performs an analog-to-digital conversion and stores this ECG data in memory.

In addition to an ECG, we thought it would be interesting to add on additional, relevant biological data to the measurement, as it would improve a physician's understanding of the recording. A sensor that can perform pulse oximetry was used to collect both heart rate and blood oxygen levels at the time of the ECG recording. The ESP32 can then collect the data from this sensor using serial communication. The pulse oximetry sensor was put on another small PCB to allow for the user to put the sensor on his or her finger.

Finally, the solution to the problem we are attempting to solve requires that this system be completely wireless, so the microcontroller used has Wi-Fi capability with a built-in antenna. When powered on, the device will connect to the user's Wi-Fi and create a web server. This web server is accessible to the user and allows the user to see the ECG trace after a recording along with the user's blood oxygen saturation levels (SpO2) in terms of a percentage and heart rate in beats per minute at the time of the ECG recording.

1.3 Design Expectations

Throughout the course of developing our project, our expectations changed quite a bit. Our original idea was to create an electroencephalography (EEG) alarm that would track a user's sleep cycle to wake them at their lightest stage of sleep and avoid sleep inertia. In addition to this, it would provide information about a user's sleep that would allow a trained professional to gain insight into the user's sleep health at home without the user having to sleep in a laboratory. Similarly to how an ECG measures the heart's electrical signals, an EEG measures the electrical signals of the brain by way of voltage changes on the patient's scalp. For reasons outside of the scope of this document, we found that after much testing, it would not be realistic to finish this project in our constrained time-frame and limited budget. The original idea behind this project was always to create an at-home biological data monitoring system, and with this in mind we readjusted our aim from an EEG Alarm to an ECG Monitor. Please refer to the "Project Proposal Change" document on our team's website for a more detailed description of why the project was modified.

For the wireless ECG, we expected that we would be able to display around five seconds of an ECG trace on the web server. Knowing that we would not be able to create a clinical grade ECG during the semester, we did not expect the trace to be clean enough to use in a clinical setting. However, we expected that we would be able to clearly visualize the P and T waves of the ECG in addition to the QRS complex shown in Figure 1.1 below.

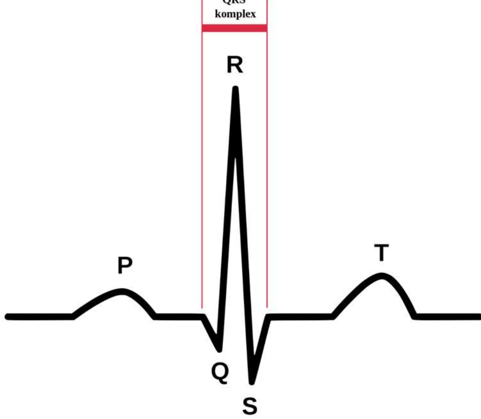


Figure 1.1 Labeled sketch of an ideal ECG trace for one cardiac cycle

After completion of the board, an ECG trace could be obtained from a person by placing the electrodes in the standard RA, LA, and LL configurations. Five seconds of data was recorded of Alex's heart electrical activity shown in Figure 1.2. Although there is noticeable noise, most significantly in the TP interval, all five landmarks - the P, Q, R, S, and T waves - are very distinct, and it is more accurate than expected. There are significant noise problems whenever the user is moving due to electrical activity from muscle action potentials being picked up, so the user must remain relatively still and relaxed during measurements.

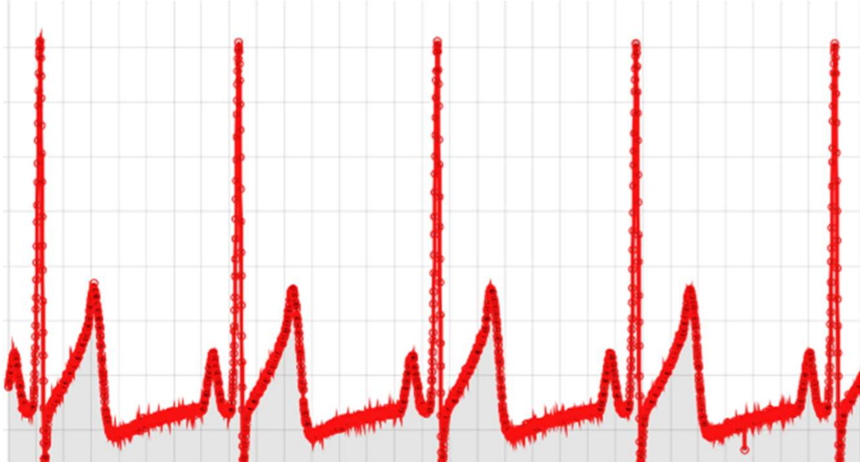


Figure 1.2 Five seconds of Alex's ECG trace recorded on the final board

Because the system needed to be wireless, the boards were all originally designed to run off of one 9V battery. After testing, the current consumption of our two boards exceeded our expectations and was not able to run off of one 9V battery. Instead, the 9V battery was used to supply the ECG circuit while a 3.7V lithium polymer battery was used to power the ESP32 and pulse oximetry system.

While the ECG, microcontroller, and wireless portion of the project eventually worked as expected, the pulse oximetry system was not nearly as accurate as first thought, especially in calculating heart rate. In addition to the heart rate being off many times, the system also takes a long time (~15-20 seconds) to record a valid sample which was much longer than originally anticipated.

2 Detailed System Requirements

Electrocardiogram (ECG)

- Measure user's cardiac signals
- Calculate the difference in electrical signals at different points in the body
- Separate cardiac signals from environmental noise in the user's body

Heart Rate

- Sensitive to signals from user's finger
- Correctly diagnoses heart rate and pulse oximetry (SpO2)
- Does not interfere with user's comfortability during sleep

Internet of Things (IoT)

- Accept user input on when to begin data collection
- Communicate ECG data to user interface
- Communicate heart rate to user interface

User Interface

- Easy to operate buttons (an/or other inputs)
- Simple yet informative display of measured information
- Timestamping data collection and saving information of last measurement

3 Detailed project description

3.1 System theory of operation

Heart contractions are the result of a wave of cellular depolarization throughout the heart, starting from the atria to the apex of the heart. The depolarization is the result of increased membrane permeability of cardiomyocytes which allows an influx of ions into the cell, meaning that an ionic current is flowing throughout the heart. This electrical signal travels through the body and is able to be picked up on the skin. Gel electrodes allow this ionic current to be transferred into current into a wire. If you measure the electric potential on the skin at two different points on the body whose vector between the points goes past the heart, one can take the difference in the two signals, amplify the differential signal, and determine the functioning of the heart from these signals. This is the basic theory of electrocardiograms.

For the pulse oximetry portion of the project, the theory of operation is that when light is allowed to penetrate into tissue, it follows a complex path of propagation including multiple scattering and absorption events. Many molecules in the body, called chromophores, can absorb light at the red and infrared wavelengths. One such important molecule is hemoglobin which carries oxygen in the bloodstream, and its absorption spectra is dependent on whether or not its heme groups are bound to oxygen or not. Pulse oximetry uses this difference in oxy- and deoxyhemoglobin absorption patterns to calculate the percentage of hemoglobin molecules in the bloodstream from which SpO2 can be found. In addition, the pulsatile motion of blood due to one's heart beat allows for the pulse oximetry system to measure relative velocity of blood. The frequency of the pulsatile blood velocity is used to calculate heart rate.

3.2 *System Block diagram*

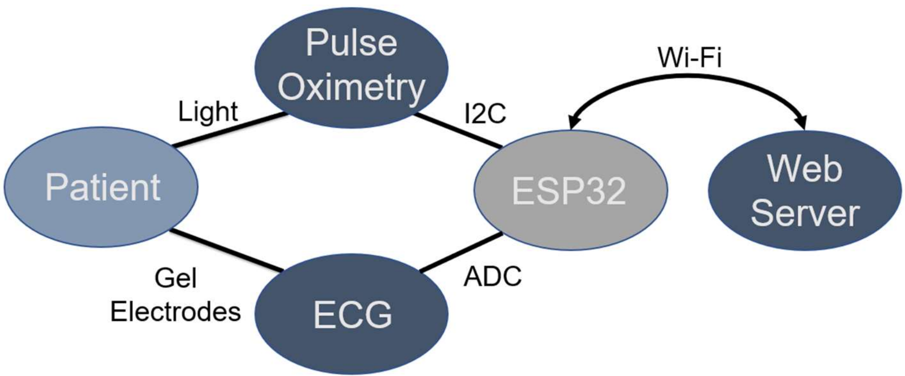


Figure 3.2.1 ECG Monitoring System Block Diagram\

The system block diagram displays the device in use as the systems interact with one another. The electrocardiogram subsystem collects data via electrodes worn on the user's chest and abdomen. The pulse oximetry sensor is pressed against the fingertip to measure blood-flow. The Internet of Things subsystem collects data from both of these devices, communicating it to the user interface over Wi-Fi. The user interface, displayed on the wireless device, collects ECG data and heart rate information for the user.

The ESP32-S3-WROOM-1-N8R8 used in this product has dual cores, so one core manages all of the wireless communication while the other core manages the rest of the processing.

3.3 *Detailed design/operation of Electrocardiogram Subsystem*

At the heart of the electrocardiogram subsystem is a highly specified JFET input instrumentation amplifier designed for just this purpose - the AD8220 by Analog Devices. Its simplified schematic is shown in Figure 3.3.1. In order to extract the heart's electrical potential activity from the rest of the noise going on in the body and environment, a system is needed that can amplify the difference between two leads while getting rid of all common mode signals. An

instrumentation amplifier does exactly this, and the one used in this project has a high common mode rejection ratio (CMRR) of at least 100 dB and an adjustable gain from 1 to 1000.

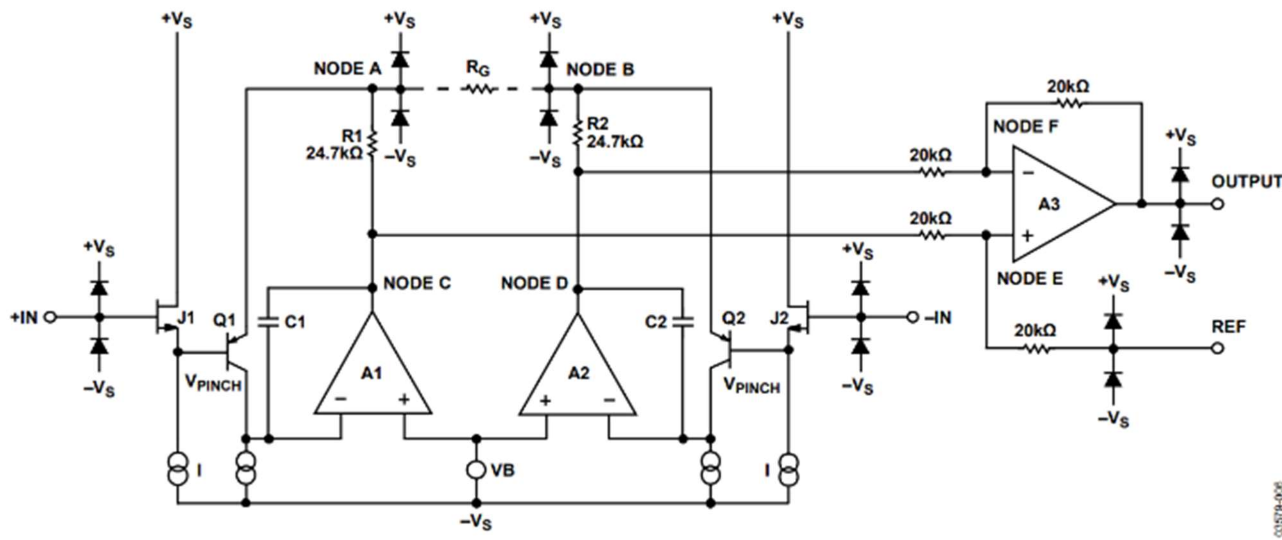


Figure 3.3.1 Simplified schematic of the AD8220 instrumentation amplifier

The gain of the instrumentation amplifier can be adjusted by changing the external gain resistor (R_G), and the gain can be calculated using the equation below.

$$Gain = 1 + \frac{49.4k\Omega}{R_G}$$

Using the purchased ECG simulator from Physio-Control shown in Figure 3.3.2, the gain resistor was set to 100 Ohms with a 1% precision resistor. If you wanted to record an ECG from a patient, you had to replace the 100 Ohm resistor with a 220 Ohm resistor to increase the gain significantly as the signal from the simulator was much larger than the signal from a patient's body. Using a patient simulator allowed for much easier testing as we knew what the signal was supposed to be. In addition it has two different heart rhythms it can output: either normal sinus rhythm (NSR) or ventricular fibrillation (VF), a life-threatening arrhythmia.

Figure 3.3.2. Patient simulator purchased off of EBay

Since the AD8220 is highly specialized, there are few peripheral components that need to be added to this portion of the system. Two of the electrode wires coming from either the patient's body or the simulator were fed directly into the two inputs of the instrumentation amplifier while the reference electrode was tied to ground through a 0 Ohm resistor which allowed for the removal of the reference electrode when the simulator was being used. The three leads attaching from the electrodes are shown in Figure 3.3.3. As can be seen, the leads end in a 3.5mm male audio connector. This male connector was connected to the board via a 3.5mm female audio connector as in Figure 3.3.4.

Figure 3.3.4. ECG sensor cables used

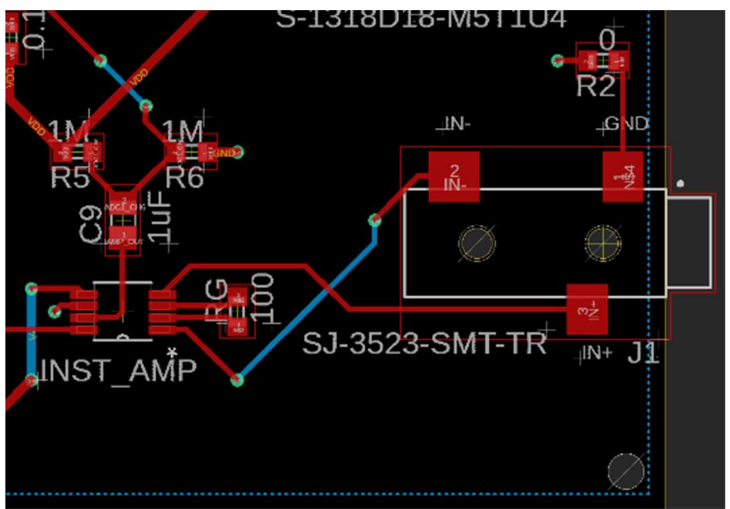


Figure 3.3.5. Board layout of ECG electronics.

Although the instrumentation amplifier should have theoretically removed any DC offset between the two leads, it was found during initial testing that the output of the AD8220 had a DC offset of a few hundred millivolts. To get rid of this DC offset, a 1 uF capacitor was placed on the output of the amplifier. In addition, the ECG signal at the output of the op-amp had a voltage range between -0.5V and 1V. The ESP32's ADC input range, however, is 0-3.3V, and this posed a problem. To solve this, a simple voltage divider was created with two 1 MΩ resistors in order to provide an offset of about 1.65V. The schematic for the ECG subsystem is presented in Figure

3.3.6 where the output of the amplifier is fed into pin 7 of the ESP32 which has a built-in analog-to-digital converter.

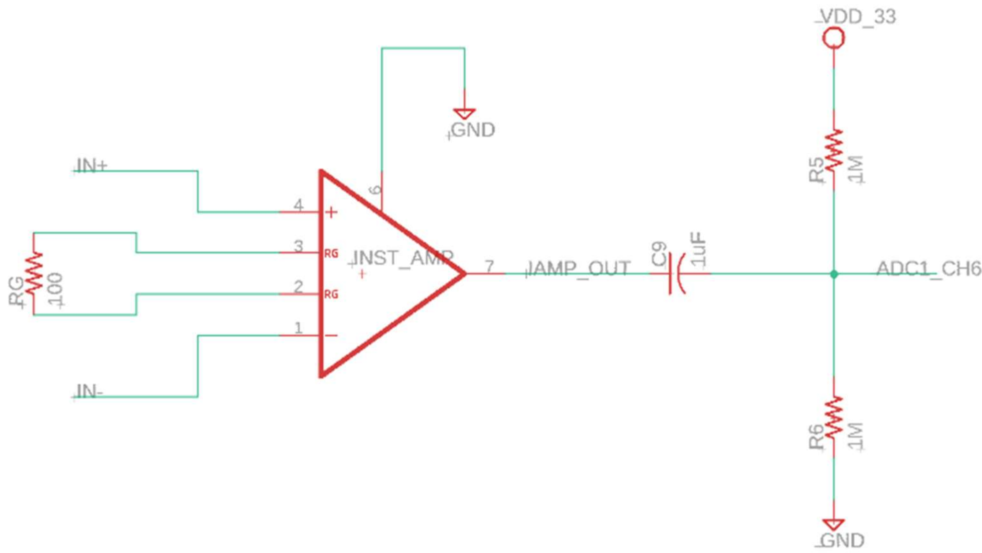


Figure 3.3.6. Schematic of the ECG subsystem

Not shown in Figure 3.3.6 is the power supply of the instrumentation amplifier. Since it is preferable to provide a dual supply voltage to the AD8220, a 9V battery was used where a virtual ground was created to supply the amplifier with ±4.5V. In order to create a virtual ground, a general purpose operational amplifier was used along with a voltage divider. Two 10 kΩ resistors were used where the non-inverting node of the op-amp was connected between the two resistors. The inverting node and the output were connected together and that signal became the ground for the rest of the circuit. The resistor selection here is important as using too small of resistors would result in a significant amount of current running through the resistors meaning that the battery would be drained at a much faster rate. Choosing too large of a resistor would mean that the current running through the two resistors would be close to the input current of the non-inverting node of the op-amp, and the input current of the op-amp could no longer be negligible. This would result in more current flowing through the top resistor in Figure 3.3.7 than

the bottom resistor, and then the voltage dropped over the two resistors would be different, no longer creating the ±4.5V. dual supply. The LM358DT was used as the general purpose op-amp, but it comes in package with dual op-amps so the unused amplifier had its non-inverting node tied to ground while the output and inverting node were connected together to minimize noise in the circuit from an unconnected op-amp. The 9V battery was connected to the PCB through a Molex 2-pin header. Two 0.1 uF decoupling capacitors were used to limit voltage fluctuations.

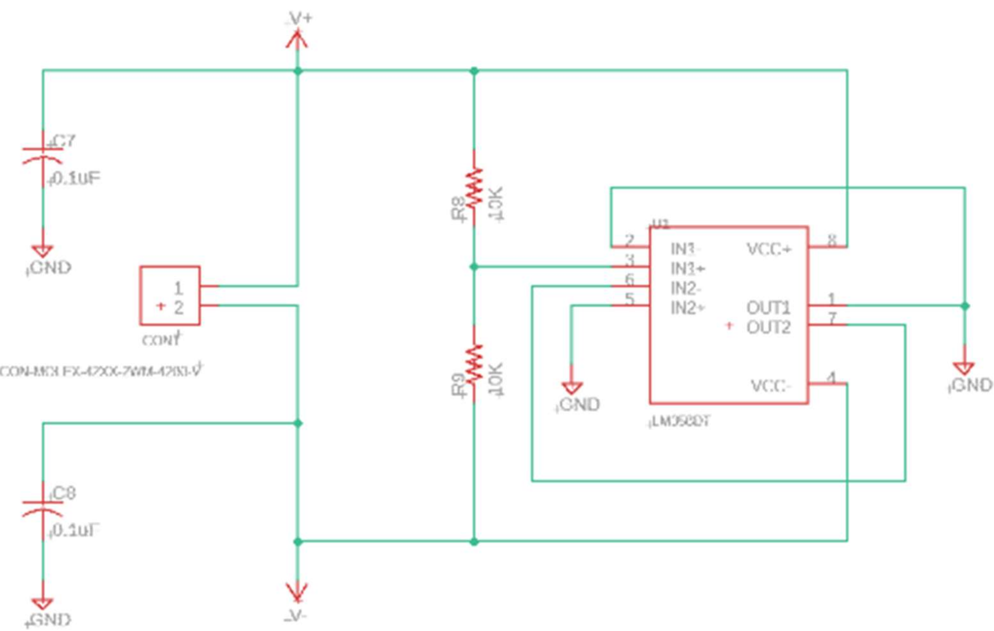


Figure 3.3.7. Schematic for the 9V battery connection, creation of a virtual ground, and splitting the battery into ±4.5V

Originally, the design was for the 9V battery to supply all the power to the board. Because the ESP32 needs a 3.3V power supply, a linear voltage regulator (XC6220) was used to drop the 4.5V from the battery down to 3.3V. Two decoupling capacitors were added per the recommendation of the XC6220 datasheet as seen in Figure 3.3.8. The 2-pin jumper header was added so that when programming the ESP32 using the USB to UART converter boards, the 3.3V

could be supplied to the ESP32 from the USB cord and not the battery. Leaving that header pin unconnected was done while programming the device. The plan was to short the two pins labeled JP1 on Figure 3.3.8 when we wanted to run the board solely off of battery power. It was found after receiving the PCB that the 9V battery was not able to source enough current to run the program on the ESP32 along with the pulse oximetry circuitry. As discussed in another section of this paper, a 3.7V 1100mAh lithium polymer battery was used to power the ESP32 and all the circuitry associated with the pulse oximetry board while the 9V battery was still being used for the ECG circuit. In the final version of the project, the jumper pins in the figure below were left open. In another iteration of this project, this voltage regulator would not be needed if one were still using both a 9V battery and a 3.7V lithium battery.

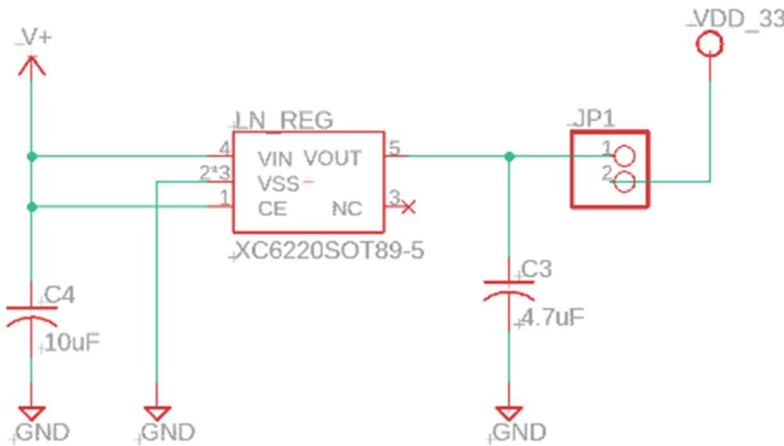


Figure 3.3.8. XC6220 linear voltage regulator converting 4.5V from the battery into 3.3V.

Now that the hardware aspect of the ECG subsystem has been discussed, there is also a critical software component to this subsystem. The ESP32-S3-WROOM-1-N8-R8 used has dual cores where one core handles all the Wi-Fi interactions and the other core is devoted to everything else - mainly data collection and processing. Since the ECG was filtered and amplified using analog methods, all the ESP32 had to do was use its ADC capabilities to read the

voltage coming in and store that data into an array. Typical ECG frequency ranges are from 0.1-100 Hz, so it is necessary according to the Nyquist Theorem to sample the data at over twice the frequency to avoid aliasing. This is why a sampling frequency of approximately 1kHz was used in the RTOS task in Figure 3.3.9. In the clinic or hospital setting, only a few seconds of ECG data is recorded at a time, so 5000 samples were collected at a sampling rate of 1kHz to produce an ECG trace 5 seconds long.

It was important to make sure that this task wasn't interrupted, so the other tasks on this core were suspended until the 5000 samples were collected, then the analogDataTask unblocked the task to start collecting heart rate and SpO2 before suspending itself.

```
void analogDataTask(void *parameter) {
  int currentIndex = 0;
  uint16_t avg;
  vTaskSuspend(analogDataTask_Handle);
  while(1) {
    uint16_t analogValue = analogRead(ADC_PIN);
    float voltage = (analogValue / 4095.0) * 3.3;
    buffer[currentIndex] = voltage;
    currentIndex++;

    if (currentIndex == ADC_BUFFER_SIZE){
      currentIndex = 0;
      printf("Buffer full\n");
      bool status = writeArrayToFile(buffer, ADC_BUFFER_SIZE);
      printf("ADC Task has been suspended\n");
      // Start collecting pulse ox data
      vTaskResume(BPM_SPO2_Handle);
      vTaskSuspend(analogDataTask_Handle);
      printf("New ADC task in progress\n");
      currentIndex = 0;
    }

    // Wait for a short period of time to achieve a sampling frequency of about 1kHz
    vTaskDelay(1 / portTICK_PERIOD_MS);
  }
}
```

Figure 3.3.9. Task to sample ECG data and store 5000 samples into an array

Our design for the project was not to have the ECG constantly be recording data, as that would be too much data for a clinician to look over if the user was wearing this device for multiple days. Instead, the user presses a recording button on the board whenever he or she

begins to have symptoms or just periodically throughout the day and this action triggers the unblocking of the task which begins recording data as shown in Figure 3.3.10. A red LED on the board indicates that a recording is in progress. In addition, the user must wait 10 seconds after one recording is done in order to begin another recording.

```
void checkButtonTask(void *parameter){

  for(;;){

    if( digitalRead(REC_PIN) == LOW ){
      vTaskDelay( 100 / portTICK_PERIOD_MS );
      if( digitalRead(REC_PIN) == LOW){
        vTaskResume(analogDataTask_Handle);
        digitalWrite(RED_LED_PIN, HIGH);
        vTaskSuspend(buttonTask_Handle);
        digitalWrite(RED_LED_PIN, LOW);
        printf("Wait a little bit...\n");
        // When the adc is done reading, wait 10 sec before user can record again
        vTaskDelay( 10000 / portTICK_PERIOD_MS );
        printf("Ready for new sample\n");
      }
    }

    vTaskDelay( 200 / portTICK_PERIOD_MS );
  }

}
```

Figure 3.3.10. RTOS task which checks for if a button is being pressed.

3.4 Detailed design/operation of Heart Rate Monitoring Subsystem

The Heart Rate Monitor Subsystem is the portion of this project that will detect and analyze the fluctuating blood flow in the wrist in order to determine the user's current heart rate, as well as calculating the user's pulse oximetry. Figure 3.4.1 displays the system diagram from blood flow to wireless monitor. The integral component of this subsystem is the Maxim Integrated MAX30101 high-sensitivity pulse oximeter and heart rate sensor shown in Figure 3.4.2, along with a simplified functional diagram in Figure 3.4.3. In order to determine the heart rate, the photosensor in the MAX30101 relies on reflected and scattered light from the LEDs that penetrates into the user's tissue. The red, 660nm, and infrared (IR), 880nm, LEDs are used for measurements in this subsystem. To measure SpO2, red and IR light reflects off of oxygenated and deoxygenated hemoglobin in the red blood cells of arterial pulsatile volumetric blood flow. Deoxyhemoglobin absorbs red light while oxyhemoglobin absorbs infrared light.

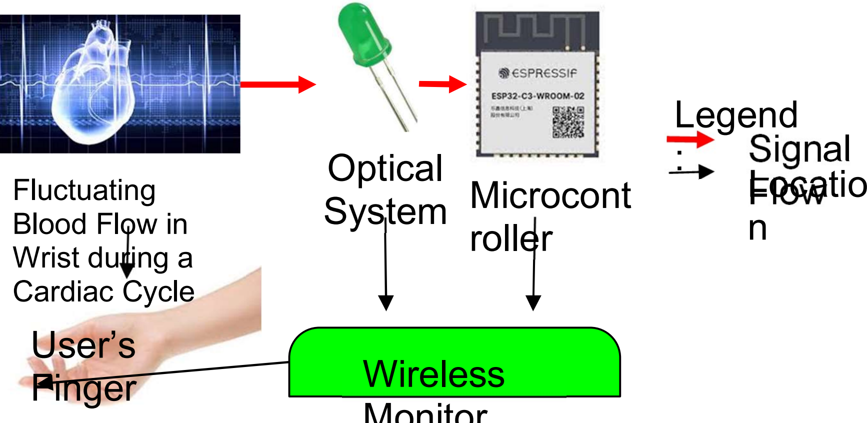


Figure 3.4.1. Heart Rate Monitor Subsystem Block Diagram

Figure 3.4.2. MAX30101 Sensor with dimensions 5.6mm x 3.3mm x 1.5mm

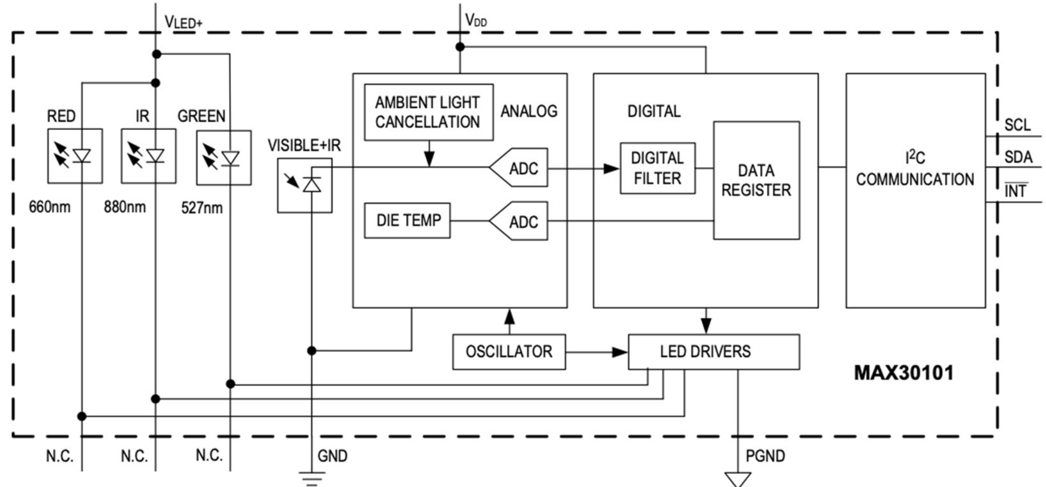


Figure 3.4.3. Functional diagram of MAX30101 Sensor

The heart rate subsystem is located on its own, small board to allow for easier access to the sensor when the user places their finger on it. The board layout is pictured in Figure 3.4.4 and the following figures display the individual schematics for each component of the subsystem. The MAX30101 sensor is blue in the figure because it is placed on the back of the board to ensure other components do not interfere with the user’s finger when placed on the sensor. In Figure 3.4.5, the schematic for the sensor displays the 1.8V power supply for the sensor’s logic and the separate 5V supply solely for the LEDs. Additionally, a 0.1uF and a 20uF capacitor are used on the 1.8V and 5V power supply lines, respectively, to reduce noise in the sensor circuit. Not pictured on this board is the voltage regulator and DC-DC boost converter used to turn the original 3.3V power supply into 1.8V and 5V, respectively. These components are placed on the main board as seen in Figure 3.4.6.

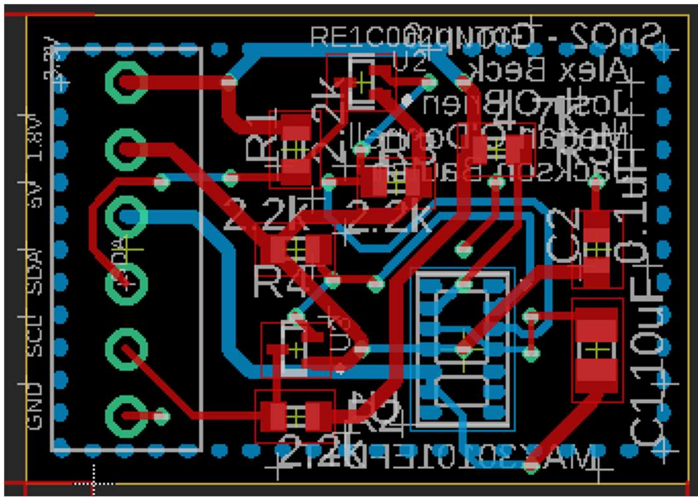


Figure 3.4.4. Board layout with dimensions 17.77mm x 25.07mm

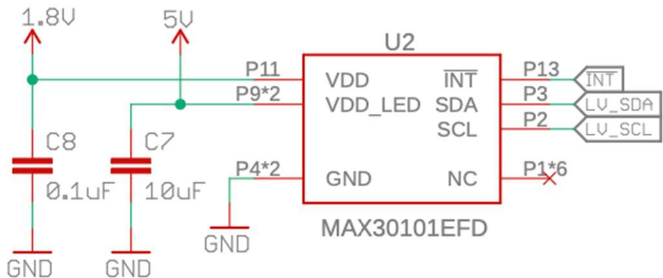


Figure 3.4.5 MAX30101 sensor schematic

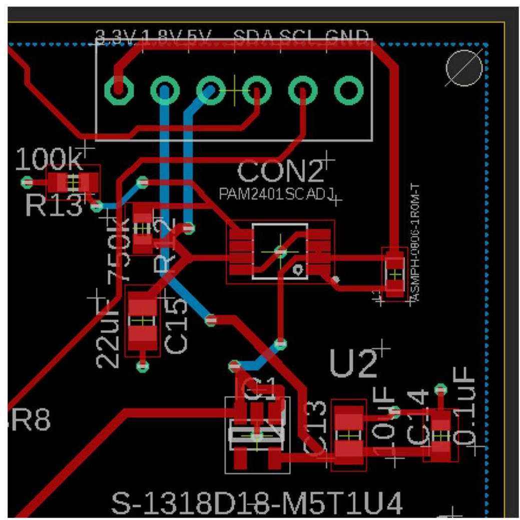


Figure 3.4.6 Portion of main board with voltage regulator (bottom right) and DC-DC boost converter (middle) for heart rate subsystem

The voltage regulator and DC-DC boost converter were placed on the main board to reduce clutter and the size of the heart beat subsystem board. These components operate on the main board and then feed voltage through the header pins seen in the top of Figure 3.4.6 to the heart beat board in Figure 3.4.4. The voltage regulator in Figure 3.4.7 takes the 3.3V power supply and turns it into a 1.8V supply to run the I2C logic in the sensor. Conversely, the PAM2401 DC-DC boost converter in Figure 3.4.8 takes the 3.3V power supply and converts it into a 5V power supply for the LEDs. As usual, capacitors are used in both circuits to reduce noise. The final part of the heart beat sensor board are the Metal-Oxide-Semiconductor Field-Effect Transistors (MOSFETs). The MOSFETs are used to translate the 1.8V I2C lines to 3.3V before the SDA and SCL signals are sent back to the ESP32, as shown in Figure 3.4.9.

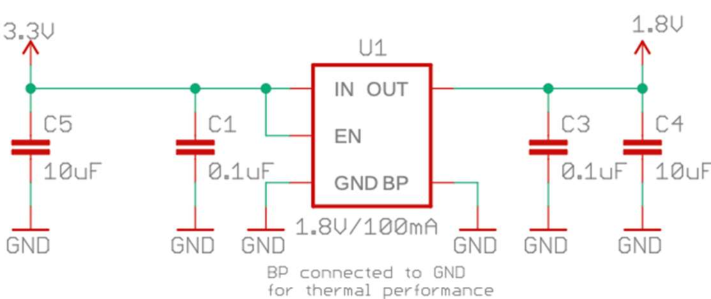


Figure 3.4.7 1.8V/100mA voltage regulator schematic

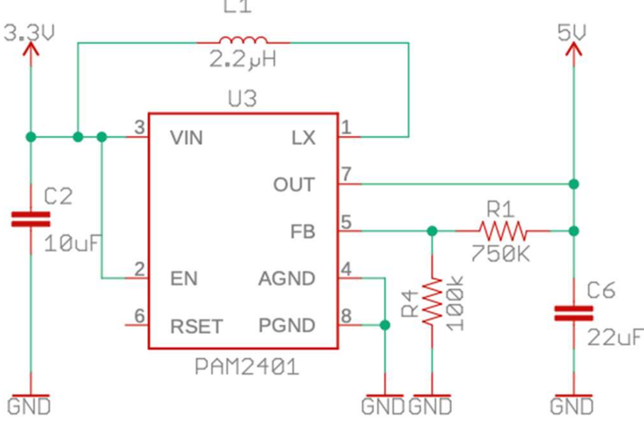


Figure 3.4.8 PAM2401 DC-DC boost converter schematic

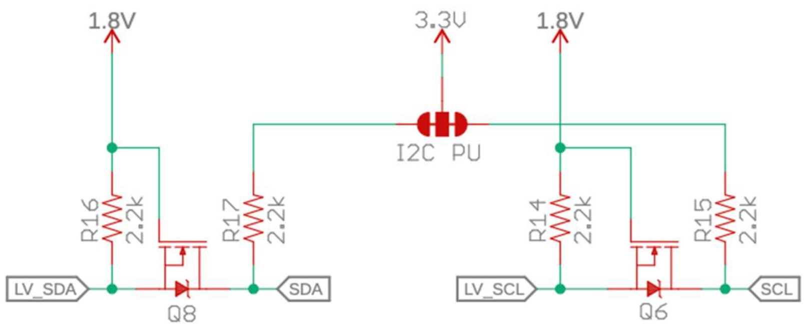


Figure 3.4.9 1.8V to 3.3V MOSFETs schematic

Now that the hardware description is complete, there is a critical software component to ensure the ESP32 processor and MAX30101 sensor are operating together to sample reflection and scattering data. When the program is running, the photosensor on the MAX30101 is capturing the intensity of light and communicating with the ESP32. In the BPM_SPO2 RTOS task, there are three instances where 100 samples are recorded. The first 100 samples, as seen in Figure 3.4.10, are recorded to determine the SpO2 signal range for a particular user. Using the MAX30101 library, the program checks to make sure the sensor is available before data collection. Then, it begins grabbing data from the sensor for the red and IR values and storing them into arrays titled redBuffer and irBuffer. Once the signal range is determined, this process is repeated to record data for the SpO2 calculation, as pictured in Figure 3.4.11. Using the MAX30101 library, the SpO2 is calculated and stored in the value “spo2.”

```
void BPM_SPO2(void *parameters)
{
  for(;;){
    vTaskSuspend(BPM_SPO2_Handle);
    printf("SPO2 Task Running\n");

    bufferLength = 100; // Buffer length of 100 stores 4 seconds of samples running at 25sps

    // Read the first 100 samples, and determine the signal range
    for (byte i = 0 ; i < bufferLength ; i++){
      while (particleSensor.available() == false){ // Do we have new data?
         particleSensor.check(); // Check the sensor for new data
      }

      redBuffer[i] = particleSensor.getRed();
      irBuffer[i] = particleSensor.getIR();
      particleSensor.nextSample(); // Move to next sample

      //printf("red=");
      //printf("%d",redBuffer[i]);
      //printf(", ir=");
      //printf("%d\n",irBuffer[i]);
    }
```

Figure 3.4.10 First 100 samples are recorded to determine SpO2 signal range

```
printf("Signal Range Dertermined (MAX30101)\n");

// Read the second 100 samples, and determine SPO2
for (byte i = 0 ; i < bufferLength ; i++){
  while (particleSensor.available() == false) // Do we have new data?
     particleSensor.check(); // Check the sensor for new data

  redBuffer[i] = particleSensor.getRed();
  irBuffer[i] = particleSensor.getIR();
  particleSensor.nextSample(); // Move to next sample

  //printf("red=");
  //printf("%d",redBuffer[i]);
  //printf(", ir=");
  //printf("%d\n",irBuffer[i]);
}
// Calculate heart rate and Sp02 after first 100 samples (first 4 seconds of samples)
maxim_heart_rate_and_oxygen_saturation(irBuffer, bufferLength, redBuffer, &spo2, &validSPO2, &heartRate, &validHeartRate);
printf("Sp02 Calculated\n");
```

Figure 3.4.11 Second set of 100 samples are recorded to calculate SpO2

The second part of the BPM_SPO2 task is to calculate the heart rate in beats per minute (BPM) as shown in Figure 3.4.12. To start, the arrayBPM must be zeroed out from the previous recording so that the next calculation is a result of the current user's data. A key difference between the SpO2 readings and heart rate readings is that the heart rate only uses IR samples because it is measuring a difference in pulses in the arterial blood flow. After grabbing IR data,

the checkForBeat function determines if there was enough pulsatile blood flow to determine a beat. Then, it calculates the BPM and stores it in the arrayBPM before calculating the average of the collected BPM measurements in Figure 3.4.13. One issue we had was that the heart rate data collection would continue running, even when the user's finger was removed from the sensor. The arrayBPM only records a value when a heartbeat is detected, so at times the program would run forever without sensing a beat. To fix this, we added a for loop that terminates the data collection once 650 data samples have been collected. Finally, the average of the calculated BPMs is recorded into an integer, beatAvg. At this point, the beatAvg and spo2 integers are written onto a JSON file to be sent over Wifi to the user interface. Please refer to the detailed design of the Internet of Things subsystem for an explanation on how the data is transferred over Wifi.

```
//Zero out array from last recording
for(int i=0; i>25; i++){
  arrayBPM[i]=0;
}

// Read the third set of samples, and determine HR
int i = 0;
int j = 0;
while(i < 25){
  while (particleSensor.available() == false) // Do we have new data?
     particleSensor.check(); // Check the sensor for new data

  long irValue = particleSensor.getIR();
  particleSensor.nextSample(); // Move to next sample

  //printf("ir=");
  //printf("%d\n",irValue);

  if (checkForBeat(irValue) == true){
    //We sensed a beat!
    long delta = millis() - lastBeat;
    lastBeat = millis();
    beatsPerMinute = 60 / (delta / 1000.0);
    arrayBPM[i] = (byte)beatsPerMinute;
    printf("%d", i);
    printf(" - ");
    printf("%d\n",arrayBPM[i]);
    i++;
  }
  j++;
  if(j>650){
    i = 25;
  }
}
printf("BPM Measured\n");
```

Figure 3.4.12 Third set of 100 samples are recorded to calculate heart rate

```
    //Take average of readings
    beatAvg = 0;
    count = 0;
    for (int x = 0 ; x < 25 ; x++){
      if(arrayBPM[x]>44 && arrayBPM[x]<151){
        beatAvg += arrayBPM[x];
        count++;
      }
    }
    if (beatAvg == 0) {
      beatAvg = -1;
      count = 1;
    }
    beatAvg /= count;
    printf("Average computation done\n");

    maxim_heart_rate_and_oxygen_saturation(irBuffer, bufferLength, redBuffer, &spo2, &validSPO2, &heartRate, &validHeartRate);

    writeIntToFile(beatAvg);
    writeIntToFile2(spo2);
    printf("Integers sent over wifi");
    printf(" - bpm= %d", beatAvg);
    printf(", spo2= %d\n", spo2);
    vTaskResume(buttonTask_Handle);
  }
}
```

Figure 3.4.13 Heart rate samples are averaged and sent to user interface over WiFi

During testing, there were a few problems that we encountered. To start, there always seemed to be an I2C issue. Whether it was a software or hardware problem, the semester was defined by fixing I2C connections and determining which pins to use. Additionally, there was a point where the samples that the sensor was collecting were maxing out at a certain value, leaving us with corrupted data and thus, incorrect BPM and SpO2 calculations. To fix this, we turned down the brightness on the LEDs because we determined that the reflection and scattering from the tissues in the finger were too large for the photosensor to handle. Finally, one of the most difficult components to solder was the MAX30101 sensor because of its small metal contacts located on the back of the device. We accidentally melted the first sensor beyond repair. With one more sensor left, we were able to solder it on to where it at least looked good. However, once testing began, we were not receiving any data even though the LEDs were on. Initially, we believed one or both of the SDA and SCL pins were not connected. Upon further inspection, we determined that the 1.8V logic power supply was slightly unconnected. To fix this, we had to rub a small amount of solder next to the pin and then jam it underneath the small,

less than 0.3mm gap between the sensor's VCC metal contact and the board. Then, using the heat gun, we cured the solder enough to begin receiving signals again.

3.5 Detailed Design/Operation of Internet of Things Subsystem

The Internet of Things (IoT) subsystem will allow wireless integration of all subsystems and communication with the user's device. The requirements for this subsystem include:

- Accept user input on when to begin data collection
- Communicate ECG data to user interface
- Communicate heart rate to user interface

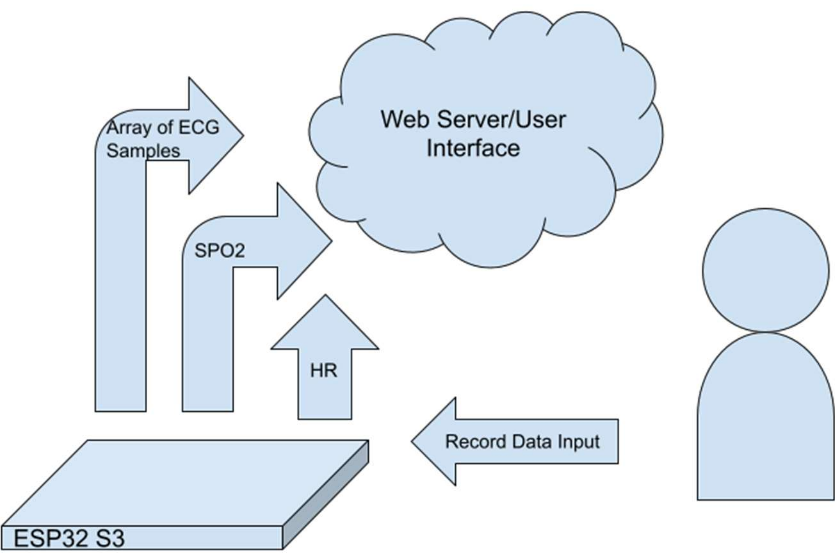


Figure 3.5.1. Internet of Things Subsystem Diagram

The Internet of Things Subsystem required a litany of design decisions. First, the creation of a local server was needed to have a destination for both the ECG data and the heart rate/blood oxygen saturation metrics. To do so, we built the server over Wi-Fi, utilizing capabilities of the arduino Wi-Fi, ESP Asynchronous Web Servers, SPIFFs, and Websockets.

The Arduino Wi-Fi library is a core library that connects the ESP32 S3 to Wi-Fi networks and lets the microcontroller communicate with other devices over the internet. It is designed to work with a wide range of Wi-Fi modules and shields, including the popular ESP32

modules. This library allows the device to connect to the SDNet Wi-Fi provided in the Stinson Remick 205 classroom. Once connected to a Wi-Fi network, the device can utilize the library's built-in client and server classes to communicate with other devices over the internet. The client class provides a simple way to establish TCP connections to remote hosts, while the server class allows for the creation of your own TCP servers and for the handling of incoming connections. In summary, the Arduino Wi-Fi library provides a simple and efficient way to connect the ESP32 S3 to Wi-Fi networks and communicate with other devices over the internet. Its built-in client and server classes make it easy to create a wide range of internet-connected applications, such as our wireless heart monitoring device.

ESP Async Webserver is a library for the ESP32 microcontrollers that allows for the creation of web servers and handles HTTP requests asynchronously. One of the key features of the ESP Async Web Server is its asynchronous design, which means that it can handle multiple requests simultaneously without blocking the execution of the rest of the program. Asynchronous web servers respond to multiple requests without slowing down the performance of your device. In addition to handling HTTP requests, the ESP Async Web Server also provides support for serving static files from your device's file system. This is useful when you want to serve HTML, CSS, and JavaScript files to clients that connect to your server, which our project does using SPIFFS, which is a piece of the project explained below. To conclude, the ESP Async Webserver is a powerful and efficient library for creating web servers on ESP32 devices. Its asynchronous design and support for serving static files make it a great fit for our project.

Serial Peripheral Interface Flash File System (SPIFFS) is a file system for embedded devices that use flash memory to store data. ESP32s have built-in flash memory that can be used to store application code and other data. The SPIFFS file system is designed to be lightweight

and easy to use according to online sources, with a simple process for reading, writing, and deleting files stored on the flash memory. It works by dividing the flash memory into fixed-size blocks and organizing these blocks into a file system structure that allows files to be stored and retrieved quickly and efficiently. This project uses SPIFFS to configure the HTML, CSS, and Javascript web server files. Its efficient use of flash memory makes it an ideal choice for small-scale embedded projects that need to store data and files locally. Therefore, SPIFFS is a powerful and easy-to-use file system for embedded devices that use flash memory and a successful design decision for our project.

The last element of the web server is websockets. WebSockets are a protocol for bidirectional, real-time communication between a client and a server over a single connection. They provide real-time updates and real-time data visualization for our project. WebSockets work by establishing a persistent connection between the client and the server, allowing data to be sent and received in real time without the need for repeated HTTP requests. The protocol is built on top of HTTP, using an initial HTTP handshake to establish the connection and upgrade the protocol to the WebSocket protocol. Once the connection is established, both the client and the server can send data to each other at any time, without the need for a request/response cycle. The WebSocket protocol provides a number of advantages over traditional HTTP-based communication, including lower latency, reduced bandwidth usage, and better support for real-time applications. It also allows for bi-directional communication, meaning that both the client and the server can send data to each other, as opposed to traditional HTTP, where the client initiates a request and the server responds. To conclude, WebSockets provide a powerful and efficient way to enable real-time communication between a client and a server in web

applications. Their benefits for real-time applications make this an obvious choice for our project.

The software for creating and maintaining the web server is shown in the following figures. Figure 3.5.2 shows where the constants were defined including the HTTP port, Wi-Fi SSID, and Wi-Fi password. Figure 3.5.3 shows the initialization of the SPIFFS and flashes the LED on the board if there is an error. Figure 3.5.4 is the code for connecting to the Wi-Fi and delays the rest of the program until the Wi-Fi connects. Figure 3.5.5 is the code for intializing the web server by, on the root request, sending the SPIFFS and beginning the server connection over Wi-Fi. Figure 3.5.6 shows the program for notifying the clients of the server status, by utilizing JSON strings, which will be explained in detail below. Figure 3.5.7 shows the program for handling the asynchronous web socket messages including an error check. Figure 3.5.8 is the function for handling the asynchronous server events including clients connecting and disconnecting from the server. Figure 3.5.9 is the code for initializing the web socket and Figure 3.5.10 is the code for updating the clients with the cleanupClients() function in the websocket library.

```
#define HTTP_PORT 80
static const BaseType_t app_cpu = 1;
const char *WIFI_SSID = "SDNet";
const char *WIFI_PASS = "CapstoneProject";
AsyncWebServer server(HTTP_PORT);
AsyncWebSocket ws("/ws");
```

Figure 3.5.2. Initialization of Web Server Constants

```
//Initialize SPIFFS
void initSPIFFS() {
  if (!SPIFFS.begin()) {
    printf("Cannot mount SPIFFS volume...\n");
    while (1) {
        onboard_led.on = millis() % 200 < 50;
        onboard_led.update();
    }
  }
}
```

Figure 3.5.3. Initialize SPIFFS Function

```
//Connect to Wifi
void initWiFi() {
  WiFi.mode(WIFI_STA);
  WiFi.begin(WIFI_SSID, WIFI_PASS);
  printf("Trying to connect [%s] ", WiFi.macAddress().c_str());
  while (WiFi.status() != WL_CONNECTED) {
      printf(".");
      delay(500);
  }
  printf(" %s\n", WiFi.localIP().toString().c_str());
}
```

Figure 3.5.4. Connect to Wi-Fi Program

```
//Initialize Web Server
String processor(const String &var) {
    return String(var == "STATE" && onboard_led.on ? "on" : "off");
}

void onRootRequest(AsyncWebServerRequest *request) {
  request->send(SPIFFS, "/index.html", "text/html", false, processor);
}

void initWebServer() {
    server.on("/", onRootRequest);
    server.serveStatic("/", SPIFFS, "/");
    server.begin();
}
```

Figure 3.5.5. Initializing Web Server Functions

```
//Initialize Web Socket
void notifyClients() {
    const uint8_t size = JSON_OBJECT_SIZE(1);
    StaticJsonDocument<size> json;
    json["status"] = onboard_led.on ? "on" : "off";

    char buffer[17];
    size_t len = serializeJson(json, buffer);
    ws.textAll(buffer, len);
}
```

Figure 3.5.6. Notify Server Clients Function

```
void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
    AwsFrameInfo *info = (AwsFrameInfo*)arg;
    if (info->final && info->index == 0 && info->len == len && info->opcode == WS_TEXT) {

        const uint8_t size = JSON_OBJECT_SIZE(1);
        StaticJsonDocument<size> json;
        DeserializationError err = deserializeJson(json, data);
        if (err) {
            printf("deserializeJson() failed with code ");
            printf("%s\n", err.c_str());
            return;
        }

        const char *action = json["action"];
        if (strcmp(action, "toggle") == 0) {
            //led.on = !led.on;
            notifyClients();
        }

    }
}
```

Figure 3.5.7. Handling Web Socket Messages Function

```
void onEvent(AsyncWebSocket *server,AsyncWebSocketClient *client,AwsEventType type,void *arg,uint8_t *data,size_t len) {
    switch (type) {
        case WS_EVT_CONNECT:
            printf("WebSocket client #%u connected from %s\n", client->id(), client->remoteIP().toString().c_str());
            break;
        case WS_EVT_DISCONNECT:
            printf("WebSocket client #%u disconnected\n", client->id());
            break;
        case WS_EVT_DATA:
            handleWebSocketMessage(arg, data, len);
            break;
        case WS_EVT_PONG:
        case WS_EVT_ERROR:
            break;
    }
}
```

Figure 3.5.8. Handling Asynchronous Events Function

```
void initWebSocket() {
    ws.onEvent(onEvent);
    server.addHandler(&ws);
}
```

Figure 3.5.9. Initialize Web Sockets Program

```
void updateClients(void *parameters){
  while(1){
    ws.cleanupClients();
    onboard_led.on = millis() % 1000 < 50;
    onboard_led.update();
  }
}
```

Figure 3.5.10. Update Server Clients Function

To meet the requirements of this subsystem, our project needed to complete three separate tasks. First is to accept user input on when to begin data collection. To do this, the program checks for the signal on the microcontroller's digital input to be pulled low by the user selecting the button. This will trigger the response of beginning the recording process. The real time operating task associated with this button and the user input it collects is displayed in the figure below. As you can see, the button press initiates the analog data task, which collects data from the ECG, turns a LED on that lets the user know recording is in progress, and then finally suspends checking the button until the recording is complete.

```
void checkButtonTask(void *parameter){

  for(;;){

    if( digitalRead(REC_PIN) == LOW ){
      vTaskDelay( 100 / portTICK_PERIOD_MS );
      if( digitalRead(REC_PIN) == LOW){
        vTaskResume(analogDataTask_Handle);
        digitalWrite(RED_LED_PIN, HIGH);
        vTaskSuspend(buttonTask_Handle);
        digitalWrite(RED_LED_PIN, LOW);
        printf("Wait a little bit...\n");
        // When the adc and MAX3130 is done reading, wait 10 sec before user can record again
        vTaskDelay( 10000 / portTICK_PERIOD_MS );
        printf("Ready for new sample\n");
      }
    }
    vTaskDelay( 200 / portTICK_PERIOD_MS );
  }
}
```

Figure 3.5.11. Check Button Task Code

The second requirement to fulfill is to communicate ECG data to the user interface. To do this we created a JavaScript Object Notation (JSON) string and wrote it to a text file that makes data in the C++ domain available to the JavaScript/SPIFFS domain. JSON is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is often used in web applications as a way to transmit data between a client and a server. JSON data is typically represented as a collection of key-value pairs, similar to a dictionary or hash table in other programming languages. The key is always a string, enclosed in double quotes, and the value can be any valid JSON data type, including numbers, strings, booleans, arrays, or even other objects. The JSON string of ECG data is created from the array the ECG subsystem records. This program can be seen in the figure below.

```
bool writeArrayToFile(float arr[], int size) {
  // Initialize SPIFFS
  if (!SPIFFS.begin()) {
    printf("Failed to mount SPIFFS\n");
    return false;
  }

  // Serialize array as JSON
  DynamicJsonDocument doc(160000);
  JsonArray array = doc.to<JsonArray>();
  for (int i = 0; i < size; i++) {
    array.add(arr[i]);
  }
  String jsonStr;
  serializeJson(array, jsonStr);

  // Write JSON string to file
  File file = SPIFFS.open("/temp.txt", "w");
  if (!file) {
    printf("Failed to open file for writing\n");
    return false;
  }
  file.print(jsonStr);
  file.close();

  return true;
}
```

Figure 3.5.12. Send ECG Array to Web Server Program

JSON data can be easily parsed and generated using a wide range of programming languages, making it a popular choice for web applications that need to transmit data between a client and a server. Most modern web APIs use JSON as their primary data interchange format, allowing developers to easily consume and manipulate data from a wide range of sources. Our project parses the string into a new array in the JavaSript domain. The code for this process is shown in the figure below.

```
function recieveArray(){
  fetch('/temp.txt')
  .then(response => response.text())
  .then(jsonString => {
    const data = JSON.parse(jsonString);
    let i=0;
    data.forEach(value => {
      const div = document.createElement('div');
      div.textContent = value;
      temp[i]=value;
      i++;
    });
    document.getElementById("temp").innerHTML = temp;
    data = temp;
  });

}
```

Figure 3.5.13. Receive ECG Array from Microcontroller Program

The last requirement to fulfill is to communicate heart rate and SpO2 to the user interface. These communications are done the same way because they are both just integer values. This process is also a simpler version of the process outlined above for communicating the ECG data. Figures 3.5.14 and 3.5.15 show the two functions for creating the JSON string that holds the measured values and Figure 3.5.16 shows the JavaScript functions for parsing the string into a variable to be displayed on the user interface.

```
void writeIntToFile(int var){
  // Create a DynamicJsonDocument object
  DynamicJsonDocument doc(1024);
  doc["value"] = var;

  // Serialize the DynamicJsonDocument object into a JSON string
  String jsonString;
  serializeJson(doc, jsonString);

  // Write JSON string to file
  File file = SPIFFS.open("/heartRate.json", "w");
  if (!file) {
    Serial.println("Failed to open file for writing");
  }
  file.print(jsonString);
  file.close();
}
```

Figure 3.5.12. Send Heart Rate to Web Server Program

```
void writeIntToFile2(int var){
  // Create a DynamicJsonDocument object
  DynamicJsonDocument doc(1024);
  doc["value"] = var;

  // Serialize the DynamicJsonDocument object into a JSON string
  String jsonString;
  serializeJson(doc, jsonString);

  // Write JSON string to file
  File file = SPIFFS.open("/SPO2.json", "w");
  if (!file) {
    Serial.println("Failed to open file for writing");
  }
  file.print(jsonString);
  file.close();
}
```

Figure 3.5.15. Send SpO2 to Web Server Program

```
let mydata;
function recieveHR(){
  fetch('/heartRate.json')
  .then(response => response.json())
  .then(data => {
      mydata = data;
    });
}


let mydata2;
function recieveSPO2(){
  fetch('/SPO2.json')
  .then(response => response.json())
  .then(data => {
      mydata2 = data;
    });
}
```

Figure 3.5.16. Receive Heart Rate and SpO2 from Microcontroller Program

The only hardware involved in this subsystem is the microcontroller and aforementioned button. Testing involved in this subsystem includes testing the server connection and websocket capabilities with online tutorial provided HTML files. Also, it involves sending hard-coded integers and a hard-coded array of integers of floats using the functions outlined above.

3.6 Detailed Design/Operation of User Interface Subsystem

The user interface will display data collected by the ECG and heart rate monitor to the device's user. The requirements for this subsystem include:

- Easy to operate buttons (and/or other inputs)
- Simple yet informative display of measured information
- Timestamping data collection and saving information of last measurement

The user interface will be a website accessible from any internet browser. Figure 3.6.1 shows the zoomed out view of the website.

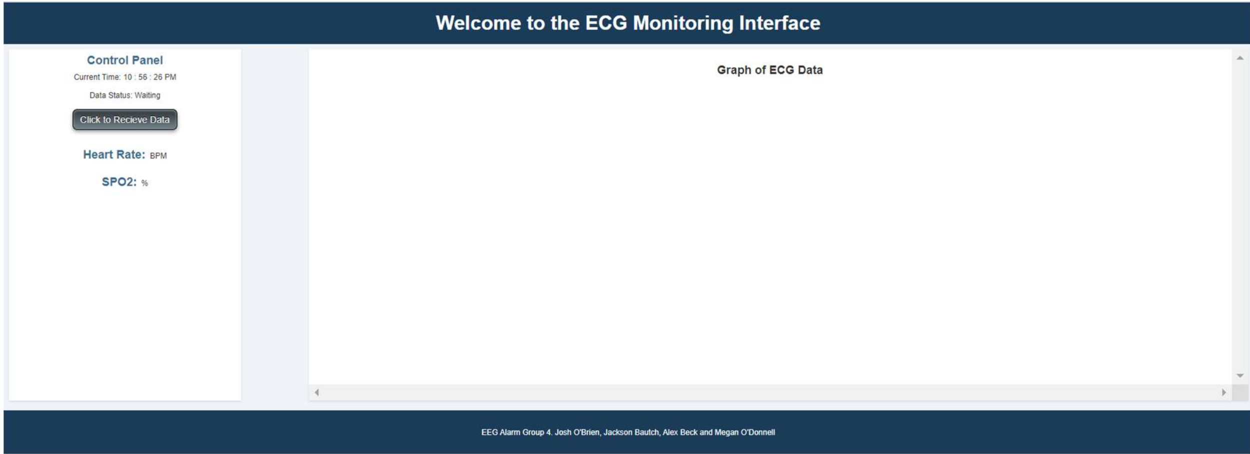


Figure 3.6.1. Zoomed Out User Interface

The first section is the header. This is the blue stripe at the top with the title “Welcome to the ECG Monitoring Interface.” The footer is the blue stripe at the bottom with the text “EEG Alarm Group 4. Josh O’Brien, Jackson Bautch, Alex Beck, and Megan O’Donnell.” The next part is Section 1, or the Control Panel. A better view of this section is in the figure below. This section’s home state includes the control panel title, a live updating current time, the status of the data to be viewed, a button to control the reception of the recorded data, and a place to display the heart rate and blood oxygen saturation.

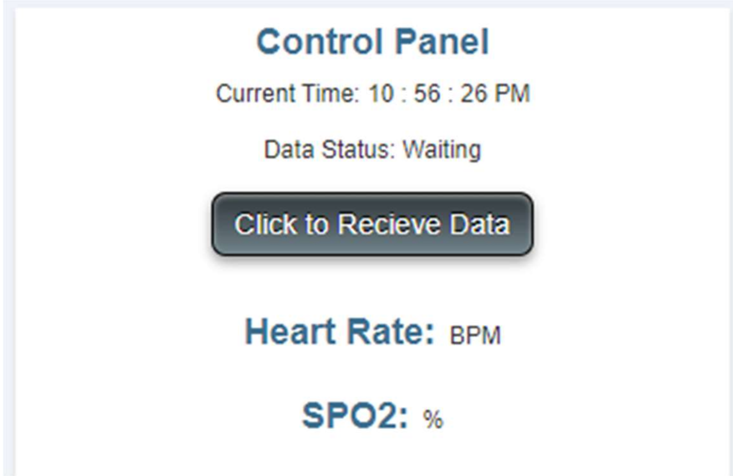


Figure 3.6.2. Section 1: Control Panel

If the user selects the button, the interface of the control panel changes a bit. The changes can be seen in the next figure below. As you can see, the data status updates to reflect that the data has been received, the button changes to now initiate the website update that allows the user to view the BPM and SPO2 measurements, there is a timestamp of when the data was collected, and there is a new button that allows the user to view the graph of the ECG data.

Figure 3.6.3. Section 1: Control Panel After Button Click

If the user selects the "Click to View HR and SPO2" button, then that data will be presented in the control panel as the figure below depicts. Additionally, the instructions for refreshing the page before trying to receive a new set of data is displayed at the bottom.

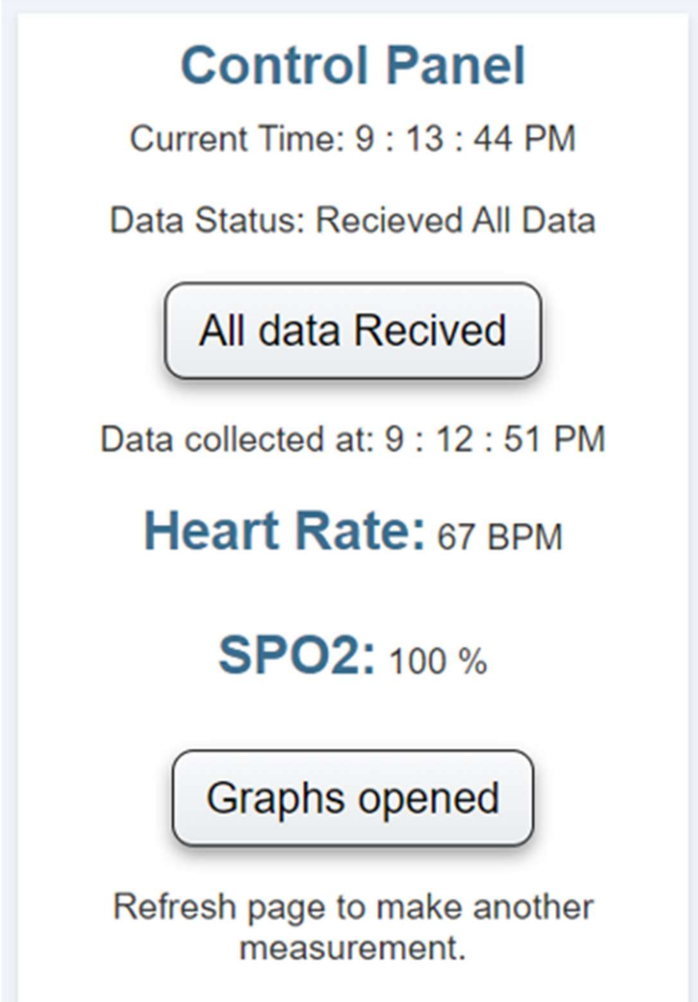


Figure 3.6.4. Section 1: Control Panel After Two Button Clicks

Before we discuss the next button click, we need to introduce Section 2, or the Graph of ECG Data. This is the section on the right that is blank until the user selects the "View Graphed Data" button. Upon clicking the graph will appear, as shown in the figure below.

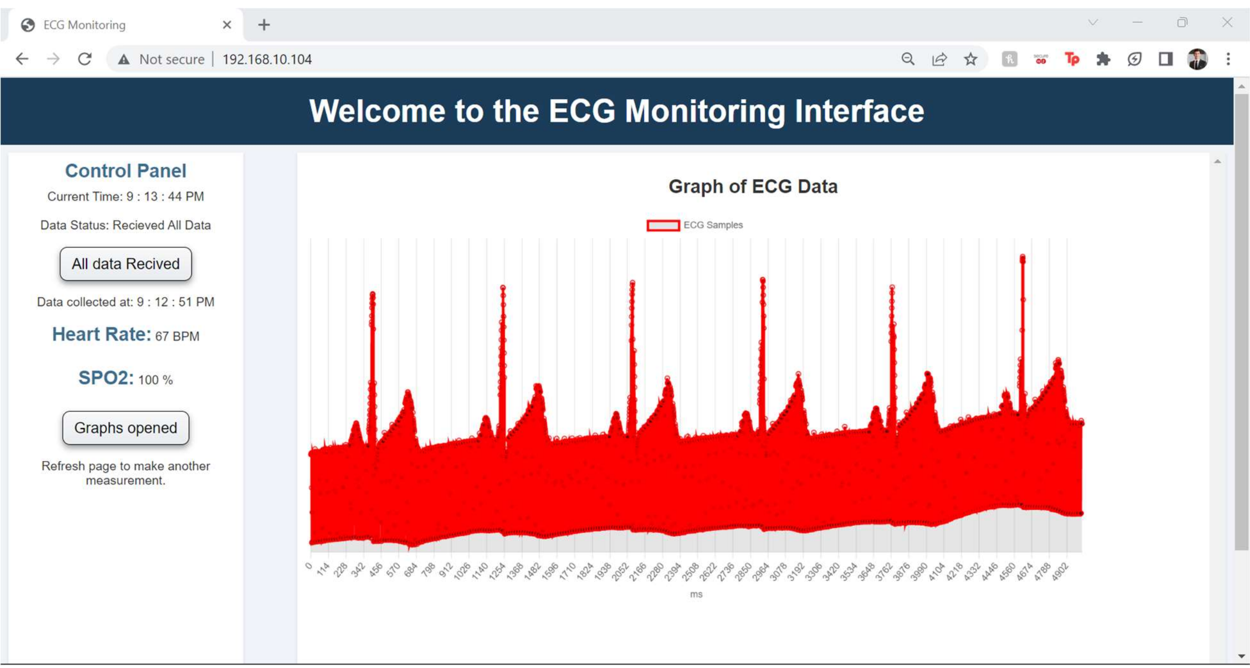


Figure 3.6.5. Section 2: Graph of ECG Data

There is no hardware involved in this subsystem. The software consists of four files, one in HTML, one in JavaScript, one in CSS and one icon , which can be found in the data file in the project code zip file. These programs are shown in the figures below. The icon is used for the small image on the web browser tab.

To begin, we will discuss the HTML file titled, index.html. HTML (Hypertext Markup Language) is the standard markup language used to create web pages and web applications. It provides a set of markup tags and attributes that define the structure, content, and appearance of web pages. HTML documents are made up of a series of elements, which are represented by tags. For example, the `<html>` tag represents the beginning of an HTML document, the `<head>` tag contains information about the document such as its title, and the `<body>` tag contains the actual content that is displayed to the user. Figure 3.6.6 shows the header part of the index.html program.

```
<!DOCTYPE html>
<html>
<head>
  <title>ECG Monitoring</title>
  <link rel="stylesheet" href="index.css">
</head>
```

Figure 3.6.6. index.html <head> Code

This sets the title of the website to be "ECG Monitoring" and attaches the CSS file, which will be discussed below, to control the style of the website. The body of the website is broken up into four sections, the Header, Section 1, Section 2, and the Footer. The HTML file code for the header is shown below in Figure 3.6.7 and the code for Section 1 is shown in Figure 3.6.8.

```
<body>
  <header>
    <h1>Welcome to the ECG Monitoring Interface</h1>
  </header>
```

Figure 3.6.7. HTML Code for Header

```
  <section>
    <h2>Control Panel</h2>
    <!-- Current Date and Time -->
    <div class="container">
      <div class = "child">Current Time:</div>
      <div class="time">
        <span id="hours"></span> :
        <span id="minutes"></span> :
        <span id="seconds"></span>
        <span id="meridian"></span>
      </div>
    </div>
    <br>
    <div class = "container">
      <div class = "child">Data Status:</div>
      <div class = "child" id="data_status"></div>
    </div>
    <br>
    <button onclick="recieveData()" id="button_text"></button>
    <br>
    <br>
    <div class="container" id="timestamp" style="display: none;">
      <div class = "child">Data collected at:</div>
      <div class="time">
        <span id="stamp_hours"></span> :
        <span id="stamp_minutes"></span> :
        <span id="stamp_seconds"></span>
        <span id="stamp_meridian"></span>
      </div>
    </div>
  <div class = "container">
    <h2 class = "child">Heart Rate: </h2>
    <div class = "child" id="HR"></div>
    <div class = "child"> BPM</div>
  </div>
  <br>
  <div class = "container">
    <h2 class = "child">SPO2: </h2>
    <div class = "child" id="SPO2"></div>
    <div class = "child">%</div>
  </div>
  <br>
  <button onclick="openGraphs()" id="open_graph" style="display: none;">View Graphed Data</button>
  <br>
  <div class = "container" id="message" style="display: none;">Refresh page to make another measurement.</di
  <!-- <button onclick="resetFiles()" id="reset" style="display: none;">Reset</button> -->
</section>
```

Figure 3.6.7. HTML Code for Section 1

In the code above, you can see the placement of the current time, data status, receive data button, timestamp, heart rate display, SpO2 display, open graphs button, and refresh instruction elements in Section 1, the control panel. The code for Section 2, the graph of the ECG samples is shown in the figure below. Finally, the code for the footer is shown in Figure 3.6.9.

```
<section2>
  <h2>Graph of ECG Data</h2>
  <div id ="temp" style="display: none"></div>
  <div class="chart-container">
    <canvas id="ECG"></canvas>
  </div>
</section2>
```

Figure 3.6.8. HTML Code for Section 2

```
<footer>
  <p>EEG Alarm Group 4. Josh O'Brien, Jackson Bautch, Alex Beck and Megan O'Donnell</p>
</footer>
```

Figure 3.6.9. HTML Code for Footer

Finally, the HTML code attaches the Javascript files that control the functionality of the website. The code for this attachment is found in the figure below. The first file controls the animation of the graph and the second file, index.js, will be discussed below.

```
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <script src="index.js">   </script>
</html>
```

Figure 3.6.10. HTML Code for Linking JavaScript Files

The next file to discuss is the CSS file, index.css. CSS (Cascading Style Sheets) is a style sheet language used to describe the presentation and visual styling of HTML (Hypertext Markup Language) documents. CSS provides a set of rules and properties that define how HTML elements should be displayed in web browsers. CSS works by targeting HTML elements and applying styles to them. Styles can be applied to individual elements, groups of elements, or all elements on a page and can be used to control a wide range of visual properties, including font

size, font style, font color, background color, padding, margins, borders, and more. The CSS file that controls the styles of the website in Figures 3.6.1 to 3.6.5 is shown in the figures below.

```css
html, body {
  height: 100%;
  background-color: #f0f4f8;
  font-family: Roboto, sans-serif;
  font-size: 12pt;
  overflow: auto;
  color: #333;
  margin: 0;
  padding: 0;
  align-items: center;
  justify-items: center;
  overflow-x: auto;
}
.panel {
  display: grid;
  grid-gap: 3em;
  justify-items: center;
}
header {
  background-color: #1c3d5a;
  color: #fff;
  padding: 20px;
  text-align: center;
}
h1 {
  margin: 0;
  font-size: 40px;
  text-align: center;
}
button {
  padding: .5em .75em;
  font-size: 1.2rem;
  color: #fff;
  text-shadow: 0 -1px 1px #000;
  border: 1px solid #000;
  border-radius: .5em;
  background-image: linear-gradient(#2e3538, #73848c);
  box-shadow: inset 0 2px 4px rgba(255, 255, 255, 0.5), 0 0.2em 0.4em rgba(0, 0, 0, 0.4);
  outline: none;
  margin: 0 auto;
}
.chart-container {
  width: 1000px;
  height:600px;
}
```

Figure 3.6.11. index.css Part 1

```
  display: inline-block;
}
.child {
  display: inline-block;
}
.divScroll {
  overflow:scroll;
  }
main {
  padding: 10px;
  display: flex;
  flex-wrap: wrap;
  justify-content: space-between;
}
section {
  background-color: #fff;
  box-shadow: 0 2px 4px rgba(0,0,0,0.1);
  margin-bottom: 10px;
  width: 18%;
  padding: 10px;
  text-align: center;
}
section2 {
  background-color: #fff;
  box-shadow: 0 2px 4px rgba(0,0,0,0.1);
  margin-bottom: 10px;
  width: 75%;
  padding: 10px;
  text-align: center;
  overflow:scroll;
}
section h2 {
  color: #3c6e8e;
  font-size: 24px;
  margin: 0;
  margin-bottom: 10px;
}
section p {
  margin: 0;
}
  background-color: #1c3d5a;
  color: #fff;
  padding: 20px;
  text-align: center;
}
```

Figure 3.6.12. index.css Part 2

The final piece of the puzzle is the JavaScript file, index.js. JavaScript is often used with HTML to create interactive web pages and web applications. JavaScript code can be embedded directly into an HTML document using the `<script>` tag, or included as a separate file that is

referenced in the HTML document. JavaScript can also be used to manipulate the HTML content of a web page, add or remove HTML elements, fetch data from a server, and perform other advanced tasks. When used in combination with HTML and CSS, JavaScript can create rich, interactive web pages and web applications.

Index.js controls a number of website functionality, including accepting the ECG array, Heart Rate measurement, and SpO2 measurement, which is explained in the Section 3.5. The first undiscussed functionality is keeping the server operational through five functions shown in Figure 3.6.13 below. The functions either handle messages from the microcontroller, respond to specific events such as start up and close, or maintain the websocket connection.

```
var gateway = `ws://${window.location.hostname}/ws`;
var websocket;
// Initialization
window.addEventListener('load', onLoad);

function onLoad(event) {
    initWebSocket();
    initButton();
}

// WebSocket handling
function initWebSocket() {
    console.log('Trying to open a WebSocket connection...');
    websocket = new WebSocket(gateway);
    websocket.onopen    = onOpen;
    websocket.onclose   = onClose;
    websocket.onmessage = onMessage;
}

function onOpen(event) {
    console.log('Connection opened');
}

function onClose(event) {
    console.log('Connection closed');
    setTimeout(initWebSocket, 2000);
}

function onMessage(event) {
    let data = JSON.parse(event.data);
    document.getElementById('led').className = data.status;
}
```

Figure 3.6.13. JavaScript for Server Operation

The next function that index.js fulfills is tracking the time. It does so by running the JavaScript builtin Date() function every second. This function grabs the date and time from the device running the web browser. Our updateTime() function converts the time into standard (non-military) format and adds leading zeros to single digit values. Then it assigns the time to the HTML element to be displayed on the website. Similarly, for the timestamping application of our device, the timestamp() function is run once on the button click and assigns that time to the HTML element. The code for both these functions is shown in the figure below.

```
let hours = document.getElementById("hours");
let minutes = document.getElementById("minutes");
let seconds = document.getElementById("seconds");

function updateTime() {
  let date = new Date();
  if (date.getHours()>12){
    hours.innerText = date.getHours() % 12;
    meridian.innerText = "PM";
  } else{
    hours.innerText = date.getHours();
    meridian.innerText = "AM";
  }
  // Add leading zero to minutes if necessary
  if (date.getMinutes() < 10) {
    minutes.innerText = "0" + date.getMinutes();
``} else{
    minutes.innerText = date.getMinutes();
  }
  // Add leading zero to seconds if necessary
  if (date.getSeconds() < 10) {
    seconds.innerText = "0" + date.getSeconds();
``} else{
    seconds.innerText = date.getSeconds();
  }

}
setInterval(updateTime, 1000);
```

Figure 3.6.14. JavaScript for Clock

```
let stamp_hours = document.getElementById("stamp_hours");
let stamp_minutes = document.getElementById("stamp_minutes");
let stamp_seconds = document.getElementById("stamp_seconds");
let stamp_meridian = document.getElementById("stamp_meridian");
function timestamp(){
  let date = new Date();
  if (date.getHours()>12){
    stamp_hours.innerText = date.getHours() % 12;
    stamp_meridian.innerText = "PM";
  } else{
    stamp_hours.innerText = date.getHours();
    stamp_meridian.innerText = "AM";
  }
  // Add leading zero to minutes if necessary
  if (date.getMinutes() < 10) {
    stamp_minutes.innerText = "0" + date.getMinutes();
``} else{
    stamp_minutes.innerText = date.getMinutes();
  }
  // Add leading zero to seconds if necessary
  if (date.getSeconds() < 10) {
    stamp_seconds.innerText = "0" + date.getSeconds();
``} else{
    stamp_seconds.innerText = date.getSeconds();
  }

}
```

Figure 3.6.15. JavaScript for Timestamping

The last functionality is presenting the ECG samples in a graph. The code for this section is shown below, utilizing the JavaScript Chart() function. The code establishes the x and y axes and also controls some stylistic aspects of the graph.

```
function openGraphs(){
  let graph1 = new Chart(document.getElementById("ECG"), {
        type: "line",
        data: {
          labels: labels,
          datasets: [
            {
              label: "ECG Samples",
              data: temp,
              borderColor: "red",
              fill: true,
            },
          ],
        },
        options : {
          scales: {
            y: {
              display: false,
              title: {
                display: false,
                text: 'voltage'
              }
            },
            x: {
              title: {
                display: true,
                text: 'ms'
              }
            }
          }
        }
      });
    document.getElementById("open_graph").innerHTML = "Graphs opened";
    document.getElementById("open_graph").style.backgroundImage = "linear-gradient(#f8f9fa, #e9ecef)";
    document.getElementById("open_graph").style.color = "black";
    document.getElementById("open_graph").style.textShadow = "none";
  }
```

Figure 3.6.16. JavaScript for Graph

○

3.7 Interfaces

The heart's electrical signal is separated from environmental noise within the user's body by an AD8220 instrumentation amplifier. The output of this amplifier is written to the ESP32 at pin 7, where there is a built-in analog-to-digital converter. A 1 uF capacitor was used at the output of the amplifier to eliminate a small DC offset, and a voltage divider was created using two 1 MΩ resistors. The voltage divider served to shift the output voltage, which was originally in the range of -0.5V to 1V into the range for the ESP32's ADC input range of 0-3.3V.

Data is shared between the subsystems by way of Inter-Integrated Circuit (I2C) Protocol. I2C is a serial communication protocol that allows for communication between "slave," and "master," devices. Because it is a serial communication protocol, I2C sends data between devices using two wires. Serial Data (SDA) sends and receives data between the master and the slave devices, and Serial Clock (SCL) sends the clock signal, which is controlled by the master. Data is transferred via messages, and messages are broken up into frames of data. One of these frames contains the binary address of the slave sending the message. This comes after the start condition that starts the message, which switches the SDA wire from a high voltage to a low voltage. The message ends with the stop condition, which switches the SDA wire back to a high voltage. Following the address frame is a single bit indicating whether the master is sending the message to a slave (0) or receiving the message from a slave (1). Each frame has an ACK/NACK bit to indicate whether the frame was received (acknowledge, ACK) or not (no-acknowledge, NACK). In order for the next frame of the message to be sent, the ACK bit must be received from the device (either slave or master) sending the message. All slave devices connected to the master receive the message. Only the device with an address matching the address frame will return a low voltage ACK bit to the master following the address frame; the SDA wire will remain at a high voltage for all devices that are not being addressed.

When connected to SDNet WiFi in Stinson Remick, the Arduino WiFi library allows the connection of multiple ESP32 S3 devices over the internet. The ESP Async Webserver library establishes web servers and can handle multiple HTTP requests asynchronously. Serial Peripheral Interface Flash File System (SPIFFS) stores files in the flash memory of an embedded device. Our project uses SPIFFS to configure the HTML, CSS, and Javascript web server files because of how quickly it can retrieve files. Our project makes use of websockets for real time

data visualization because it allows data to be communicated between a client and server, without need for HTTP requests.

The user begins data collection by pressing a button on the board. Pressing the button pulls the microcontroller's digital input low, signifying to the RTOS task to begin data collection. ECG, heart rate, and SpO2 data are written to a C++ file by way of a JSON file, which is made available to the SPIFFS domain in order to be displayed on the user interface.

4 System Integration Testing

4.1 Describe how the integrated set of subsystems was tested.

The system integration testing was an arduous process that entailed all of us combining our work into one system. One thing we immediately discovered was the need to run our code in RTOS because we needed to separate some functions of the system onto the two different cores of the ESP32. One core handled the WiFi interaction while the other core handled the data collection and data processing before sending it to the user interface. Then, we tried to run our data collection tasks at the same time. That is, we wanted to run all of our data collection in one task. This was not feasible and caused the ESP32 to crash. To fix this, we created different tasks for the functions of data collection. Additionally, we discovered that functions that send data over Wifi to the user interface could not be put at the end of our code – they needed to be within the specific tasks they correspond to.

When testing our board and hardware involved, we checked all of the voltage levels to ensure we were using the correct power supply. We discovered that once the pulse oximetry sensor was attached, the 9V battery power supply did not provide enough power to the board. To fix this, we added a 3.7V lithium-polymer battery to supplement the 9V battery. Whenever it seemed there was a communication issue, we used the Saleae Logic Analyzer to determine what

signals were being sent over the lines. Finally, to confirm that our two sensors were functioning efficiently, we used a simulator and oscilloscope to accurately measure the ECG and we used a commercially available pulse oximeter to confirm our MAX30101 results.

4.2 Show how the testing demonstrates that the overall system meets the design requirements

The testing meets the overall system requirements mentioned in Section 2 of this report. The ECG correctly records 5 seconds of data to display to the website from the heart simulator. This data was only incorrect when there was a disconnect between the sensor and the user. We accomplished the design requirements because we successfully measure cardiac signals on different parts of the body while canceling out environmental noise. The heart rate subsystem requirements were met because the sensor accurately detected and then calculated the heart rate and blood oxygenation level of the client without getting in the way of their everyday activities. Next, the internet of things subsystem requirements were achieved when the user presses the button on the board to begin recording, and then about 30 seconds later they are able to view the three data points (ECG, BPM, SpO2) on the user interface. Finally, the user interface was easy to use and displayed accurate data with a timestamp. These requirements are met when the user views the website where their data is displayed and is able to accurately see their readings with a timestamp after easily pressing the buttons on the screen.

5 Users Manual/Installation manual

5.1 How to install your product

In order to set up the product from a new board which has not had the program downloaded, one must first download the project library from our team's website (http://seniordesign.ee.nd.edu/2023/DesignTeams/eegalarm/top_page.html). From there, use

the Visual Studio Code application with PlatformIO installed to open up the project. Since PlatformIO does not have a board file for the ESP32-S3-WROOM-1-N8R8, one must ensure that the folder called “boards” is in the project directory which includes a json file titled “esp32s3dev.json” that was created to match the memory specifications of the ESP32 being used.

Using a USB to UART converter, connect the USB to one's personal computer and the 6 pins on the UART converter to the main board as shown below in Figure 5.1.1. Note that while using the programmer, no batteries need to be connected to the board as the power is being supplied from the USB to UART converter.

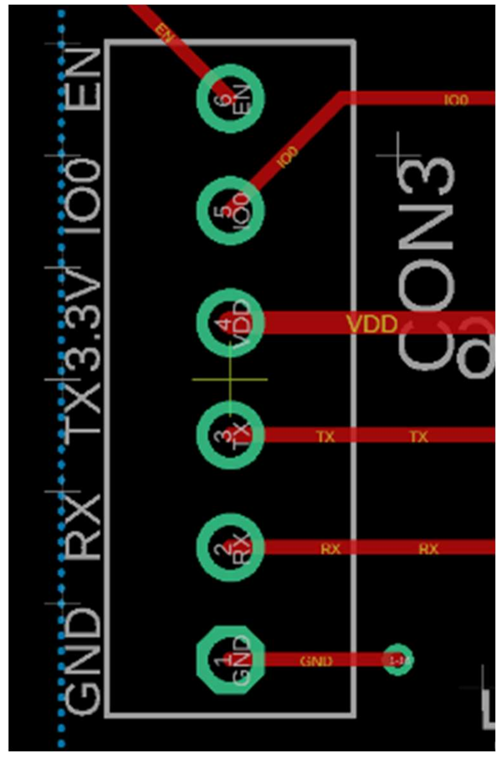


Figure 5.1.1 Header pins for programming. From top to bottom the pins are “Enable”, “GPIO0”, “3.3V”, “TX”, “RX”, and “GND”

After the connections are all made correctly, one must simply press the “Upload” button on the Visual Studio Code user interface. From there, the software installation is complete.

Note that if this were an actual product, the software would be uploaded to the ESP32 before being sold to the user.

5.2 How to setup your product

Assuming that the software installation has gone smoothly, one can now move on to powering up the product and making the essential connections between the main board and the second, smaller board which contains the MAX30101 sensor. To connect the main board to the secondary board, there will be 6 wires with each end connected to a 6-pin female Molex connector such as the one shown in Figure 5.2.1. The headers are designed such that they can only clip in one direction, so one must not need to worry about making sure the direction is correct. After connecting the two boards from the 6-pin connector on the main board in Figure 5.2.2, one can move on to powering up the product.

Figure 5.2.1. Example Molex connector

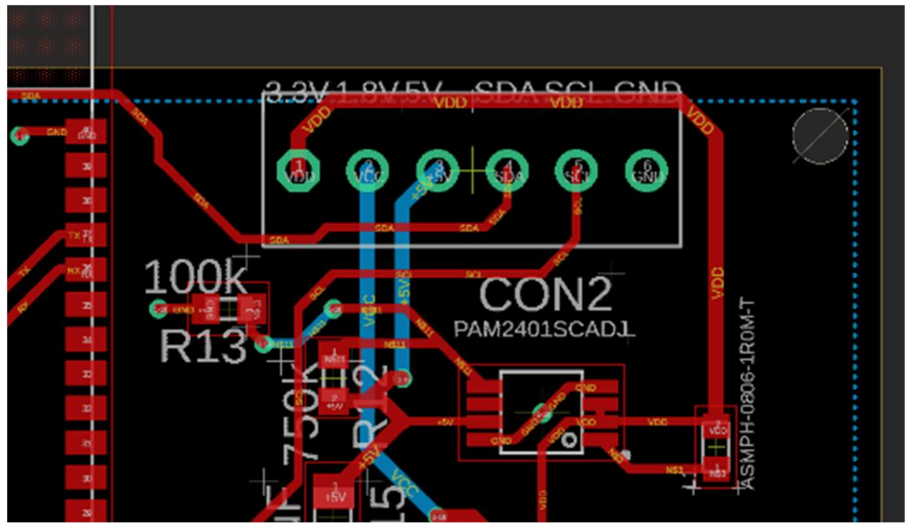


Figure 5.2.2. Header pins to connect to the MAX30101 sensor board

All of the ECG circuitry is supplied by a 9V battery as mentioned in Section 3.3, so the 9V battery should be the first of the two batteries to be connected as it will not power up the ESP32. Place the 9V battery in the smaller rectangular space in the 3D-printed housing on the top right of Figure 5.2.3. Using a two-pin Molex connector, attach the battery in the configuration mentioned in the caption of Figure 5.2.4 .

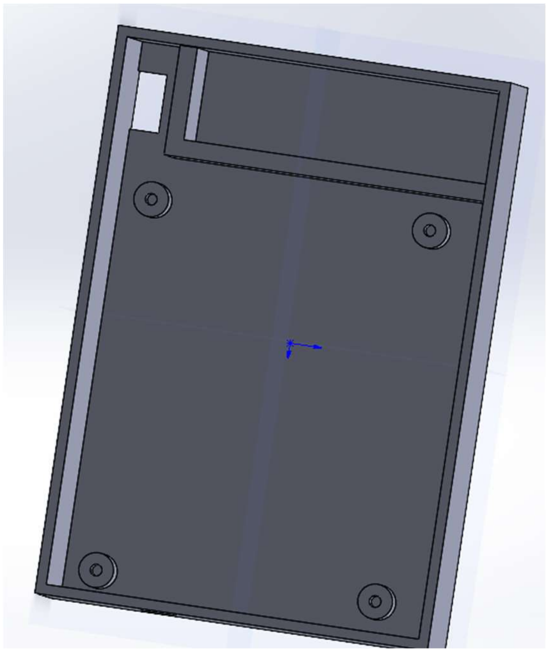


Figure 5.2.3. Inside of housing for product

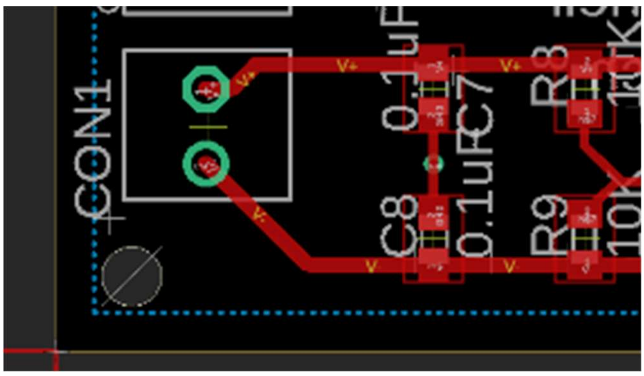


Figure 5.2.4. Connection for the 9V battery on the main board. Note that the positive terminal of the battery connects to the header pin on the top of the figure while the negative terminal of the battery connects to the bottom pin in the figure.

○ 7.3 How the User Can Tell if the Product Is Working

After following the instructions in sections 5.1 and 5.2, one should connect three gel electrodes to one's body to test if it is working in the 3-lead configuration shown here: https://litfl.com/ecg-lead-positioning/. Clip the red wire electrode wire to the RA position, the blue electrode to the LA position, and the black electrode to the LL position. From there, plug in the 3.5mm audio jack cable into the female audio connector on the side of the product. Connect the 3.7V lithium battery to the board after this, and if the green LED is lit, that means the power is working. A few seconds later, the red LED should turn on for about 8 seconds then turn off. This is a sign that the board has started up and is getting reading for data collection. When the red LED turns off, try and connect to the website at the board's assigned IP address. If this works, then press the button labeled "REC" on the board. The red LED should then light up signifying that it is taking 5 seconds of ECG data. Then the red LED will flash meaning that ECG data collection has ended and the pulse oximetry system will be taking measurements to calculate heart rate and SpO2. After the red LED has turned off, data collection has ceased, and one should ensure that the measurements and ECG chart plot on the website.

○ *7.4 How the User Can Troubleshoot the Product*

The most frequent problem encountered with this product all revolve around the biointerface connection. If ECG readings seem to be inaccurate, ensure that new electrodes are being used and there is minimal hair where the electrodes are placed. When the RA and LA leads are swapped, the ECG appears inverted so just unclip the electrode wires from RA and LA and switch them. This product has not been optimized to filter out movement, so a common problem is that ECG readings appear messy if the user is not completely still while the 5 second sample is taking place. If using the simulator, be aware that the electrode wires don't clip on correctly, and the bad connection results in the signal going erratic. Please ensure the electrodes are making firm contact with the simulator leads.

If the pulse oximetry system is returning a heart rate or oxygen saturation of a negative number, this means no valid samples were collected. This commonly occurs when one either does not press hard enough on the sensor or that one presses too hard on the sensor. Only gentle pressure is needed to obtain accurate readings. Finally, if the 2-pin jumper in the middle of the board is shorted, open the jumper as these should never be connected together.

6 To-Market Design Changes

The first improvement needed for commercial sale would be increased accuracy of the heart rate sensor. For our project we used the MAX30101 Pulse Oximeter and Heart Rate Sensor, which often gave inaccurate heart rate readings. The heart simulator that our budget allowed has room for improvement, but the ECG software itself provided accurate readings.

In terms of power supply of the product, we realized after ordering the board that the 9V battery could not supply sufficient current to run our power-intensive program. In future

iterations, one could replace the 9V battery with two 3.7V 1100mAh lithium polymer batteries connected in series to give a 7.4V (2200mAh) power supply. The connection between the two batteries can be tied to ground to give a dual power supply of ±3.7V which can supply the instrumentation amplifier. The ESP32 can take in 3.7V as a power supply so no linear regulator is needed. No circuitry needs to be changed for the pulse oximetry system; however, a smaller connector and a different shape for the pulse ox board would be advantageous. Another thought would be to have the sensor be a part of the housing of the main system which is strapped to the chest. Then, the user wouldn't need to clip anything onto his or her finger - simply hold one's finger up to the sensor on the chest while a recording is being taken.

Taking this product to market would require it to pass rigorous FDA requirements if it were to be used for clinical purposes. The current ECG circuitry would need to become more advanced in order to get to the level where it would pass FDA regulations.

7 Conclusions

Overall, all of the requirements that we set for the subsystems in section 2 of this document were met. While there are numerous problems to be addressed in a future iteration of this project such as modifying the power supply to reduce space, create a better housing for the components, overall we are confident in saying that the project was successful. Our main problems came about due to connection issues with the ECG simulator or not pressing onto the pulse ox sensor correctly rather than any engineering or design decision. While it would have been nice to implement more features, the brevity of the course limited us to what we could do given just a few months. We hope that another senior design group in the future will pick up where we left off and make improvements to our product.

8 Appendices

Component Data Sheets

ESP32-S3-WROOM-1: https://www.espressif.com/sites/default/files/documentation/esp32-s3-wroom-1_wroom-1u_datasheet_en.pdf

Instrumentation Amplifier: https://www.analog.com/media/en/technical-documentation/data-sheets/ad8220.pdf

MAX30101: https://www.analog.com/media/en/technical-documentation/data-sheets/MAX30101.pdf

General Purpose Op-Amp:
https://www.ti.com/lit/ds/symlink/lm358.pdf?ts=1683166707496&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FLM358%253Futm_source%253Dgoogle%2526utm_medium%253Dcpc%2526utm_campaign%253Dasc-null-null-GPN_EN-cpc-evm-google-wwe_cons%2526utm_content%253DLM358%2526ds_k%253DLM358%2526DCM%253Dyes%2526gclid%253DCjwKCAjwjMiiBhA4EiwAZe6jQzWeN0bZKTfKQDirQ6eZn7a6s3fUKE_TELKZdLA4XbFg0U5pBD8frxoCm08QAvD_BwE%2526gclsrc%253Daw.ds

3.3V Linear Regulator: https://media.digikey.com/pdf/Data%20Sheets/Torex/XC6220.pdf

PAM2401 (step-up DC/DC converter): https://www.diodes.com/assets/Datasheets/PAM2401.pdf

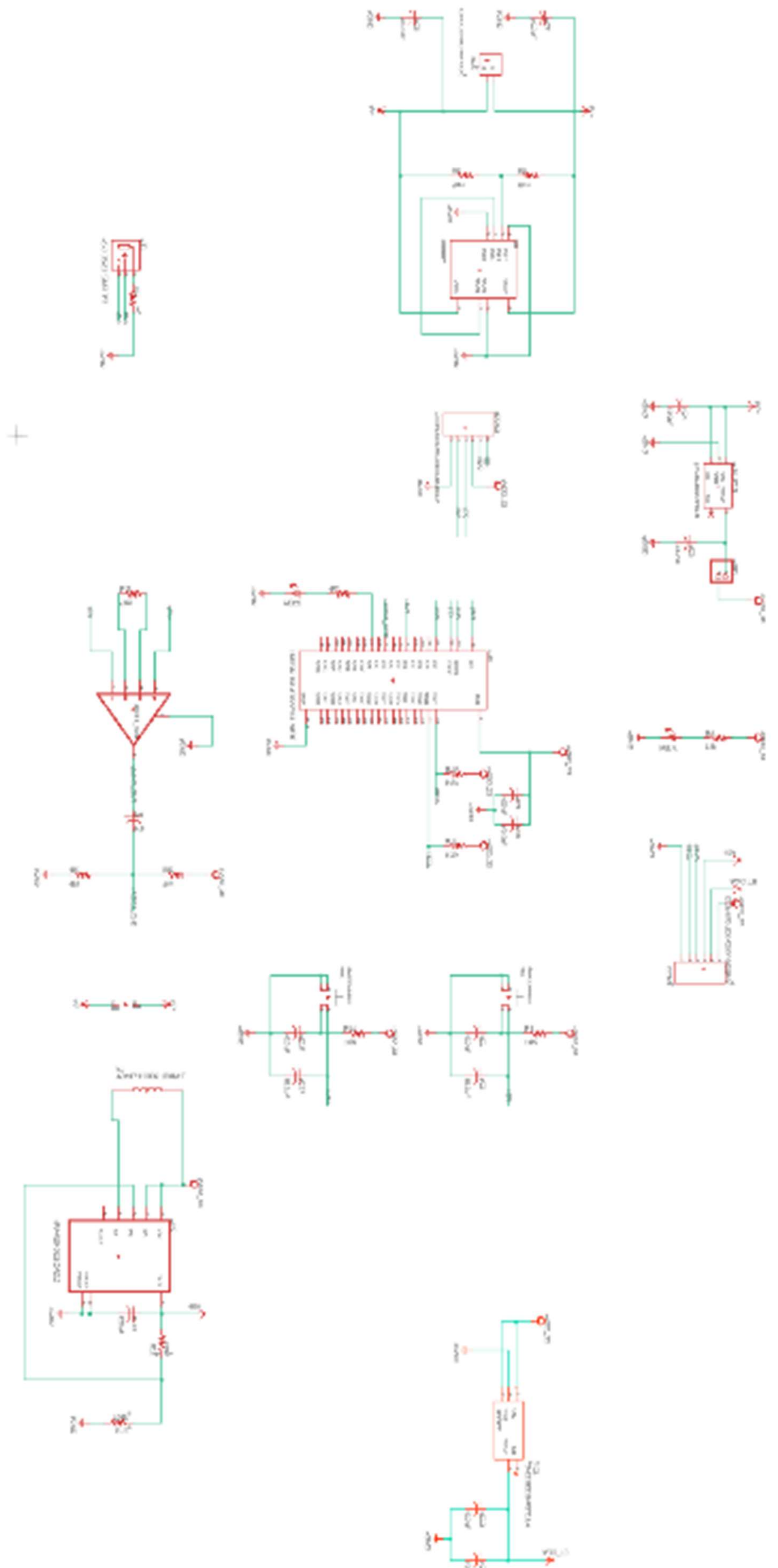
1.8V Regulator: https://www.ablic.com/en/doc/datasheet/voltage_regulator/S1318_E.pdf

N-Channel MOSFET:
https://fscdn.rohm.com/en/products/databook/datasheet/discrete/transistor/mosfet/re1c002untcl-e.pdf

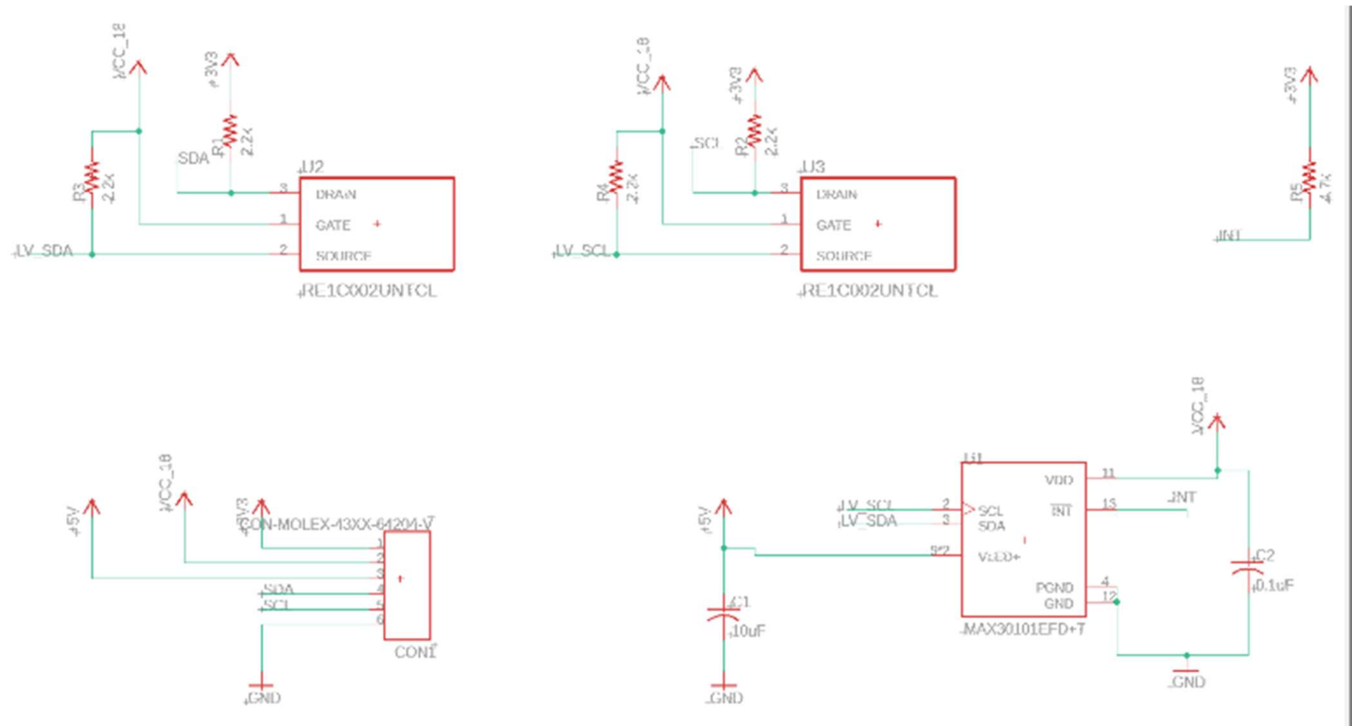
Hardware Schematics

NOTE: All schematics & board files can be found on our team's website

Main Board



Pulse Oximetry Board



Complete Software Listings

main.cpp

```
#include <Arduino.h>
#include <SPIFFS.h>
#include <WiFi.h>
#include <ESPAsyncWebServer.h>
#include <ArduinoJson.h>
#include <stdio.h>
#include "esp_log.h"
#include "MAX30105.h"
#include "spo2_algorithm.h"
#include "heartRate.h"
#include <Wire.h>


// Define input/output pins
#define REC_PIN 4
#define ADC_PIN 7
#define RED_LED_PIN 9
#define ADC_BUFFER_SIZE 5000
#define I2C_SDA 17
#define I2C_SCL 18
```

```
#define HTTP_PORT 80
static const BaseType_t app_cpu = 1;
const char *WIFI_SSID = "SDNet";
const char *WIFI_PASS = "CapstoneProject";
AsyncWebServer server(HTTP_PORT);
AsyncWebSocket ws("/ws");

TaskHandle_t analogDataTask_Handle;
TaskHandle_t buttonTask_Handle;
TaskHandle_t BPM_SPO2_Handle;

// Create a buffer to hold the analog readings
float buffer[ADC_BUFFER_SIZE];

MAX30105 particleSensor;
uint32_t redTemp;
uint32_t irTemp;

uint32_t irBuffer[100]; // IR LED sensor data
uint32_t redBuffer[100]; // Red LED sensor data
int32_t bufferLength; // Data length
int32_t spo2; // SPO2 value
int8_t validSPO2; // Indicator to show if the SPO2 calculation is valid
int32_t heartRate; // Heart rate value
int8_t validHeartRate; // Indicator to show if the heart rate calculation is valid

byte ledBrightness = 50; // Options: 0=Off to 255=50mA
byte sampleAverage = 4; // Options: 1, 2, 4, 8, 16, 32
byte ledMode = 2; // Options: 1 = Red only, 2 = Red + IR, 3 = Red + IR + Green
byte sampleRate = 100; // Options: 50, 100, 200, 400, 800, 1000, 1600, 3200
int pulseWidth = 411; // Options: 69, 118, 215, 411
int adcRange = 4096; // Options: 2048, 4096, 8192, 16384

// Heartbeat code
const byte RATE_SIZE = 4; //Increase this for more averaging. 4 is good.
byte rates[RATE_SIZE]; //Array of heart rates
byte rateSpot = 0;
long lastBeat = 0; //Time at which the last beat occurred
float beatsPerMinute;
int beatAvg;
int count;

byte arrayBPM[25]; // BPM value array
```

```
//Initialize On Board LED
struct Led {
    // state variables
    uint8_t pin;
    bool    on;

    // methods
    void update() {
        digitalWrite(pin, on ? HIGH : LOW);
    }
};
Led    onboard_led = { LED_BUILTIN, false };

//Initialize SPIFFS
void initSPIFFS() {
  if (!SPIFFS.begin()) {
    printf("Cannot mount SPIFFS volume...\n");
    while (1) {
        onboard_led.on = millis() % 200 < 50;
        onboard_led.update();
    }
  }
}

//Connect to Wifi
void initWiFi() {
  WiFi.mode(WIFI_STA);
  WiFi.begin(WIFI_SSID, WIFI_PASS);
  printf("Trying to connect [%s] ", WiFi.macAddress().c_str());
  while (WiFi.status() != WL_CONNECTED) {
      printf(".");
      delay(500);
  }
  printf(" %s\n", WiFi.localIP().toString().c_str());
}

//Initialize Web Server
String processor(const String &var) {
    return String(var == "STATE" && onboard_led.on ? "on" : "off");
}

void onRootRequest(AsyncWebServerRequest *request) {
  request->send(SPIFFS, "/index.html", "text/html", false, processor);
}
```

```
void initWebServer() {
   server.on("/", onRootRequest);
   server.serveStatic("/", SPIFFS, "/");
   server.begin();
}

//Initialize Web Socket
void notifyClients() {
   const uint8_t size = JSON_OBJECT_SIZE(1);
   StaticJsonDocument<size> json;
   json["status"] = onboard_led.on ? "on" : "off";

   char buffer[17];
   size_t len = serializeJson(json, buffer);
   ws.textAll(buffer, len);
}

void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
   AwsFrameInfo *info = (AwsFrameInfo*)arg;
   if (info->final && info->index == 0 && info->len == len && info->opcode == WS_TEXT)
{

      const uint8_t size = JSON_OBJECT_SIZE(1);
      StaticJsonDocument<size> json;
      DeserializationError err = deserializeJson(json, data);
      if (err) {
         printf("deserializeJson() failed with code ");
         printf("%s\n", err.c_str());
         return;
      }

      const char *action = json["action"];
      if (strcmp(action, "toggle") == 0) {
         //led.on = !led.on;
         notifyClients();
      }

   }
}

void onEvent(AsyncWebSocket *server,AsyncWebSocketClient *client,AwsEventType
type,void *arg,uint8_t *data,size_t len) {
   switch (type) {
      case WS_EVT_CONNECT:
         printf("WebSocket client #%u connected from %s\n", client->id(), client-
>remoteIP().toString().c_str());
```

```
            break;
        case WS_EVT_DISCONNECT:
            printf("WebSocket client #%u disconnected\n", client->id());
            break;
        case WS_EVT_DATA:
            handleWebSocketMessage(arg, data, len);
            break;
        case WS_EVT_PONG:
        case WS_EVT_ERROR:
            break;
    }
}

void initWebSocket() {
    ws.onEvent(onEvent);
    server.addHandler(&ws);
}

void updateClients(void *parameters){
  while(1){
      ws.cleanupClients();
      onboard_led.on = millis() % 1000 < 50;
      onboard_led.update();
  }
}

bool writeArrayToFile(float arr[], int size) {
  // Initialize SPIFFS
  if (!SPIFFS.begin()) {
    printf("Failed to mount SPIFFS\n");
    return false;
  }

  // Serialize array as JSON
  DynamicJsonDocument doc(160000);
  JsonArray array = doc.to<JsonArray>();
  for (int i = 0; i < size; i++) {
    array.add(arr[i]);
  }
  String jsonStr;
  serializeJson(array, jsonStr);

  // Write JSON string to file
  File file = SPIFFS.open("/temp.txt", "w");
  if (!file) {
    printf("Failed to open file for writing\n");
```

```
    return false;
  }
  file.print(jsonStr);
  file.close();

  return true;
}

void writeIntToFile(int var){
  // Create a DynamicJsonDocument object
  DynamicJsonDocument doc(1024);
  doc["value"] = var;

  // Serialize the DynamicJsonDocument object into a JSON string
  String jsonString;
  serializeJson(doc, jsonString);

  // Write JSON string to file
  File file = SPIFFS.open("/heartRate.json", "w");
  if (!file) {
    Serial.println("Failed to open file for writing");
  }
  file.print(jsonString);
  file.close();
}

void writeIntToFile2(int var){
  // Create a DynamicJsonDocument object
  DynamicJsonDocument doc(1024);
  doc["value"] = var;

  // Serialize the DynamicJsonDocument object into a JSON string
  String jsonString;
  serializeJson(doc, jsonString);

  // Write JSON string to file
  File file = SPIFFS.open("/SPO2.json", "w");
  if (!file) {
    Serial.println("Failed to open file for writing");
  }
  file.print(jsonString);
  file.close();
}

void checkButtonTask(void *parameter){
```

```
  for(;;){

    if( digitalRead(REC_PIN) == LOW ){
      vTaskDelay( 100 / portTICK_PERIOD_MS );
      if( digitalRead(REC_PIN) == LOW){
        vTaskResume(analogDataTask_Handle);
        digitalWrite(RED_LED_PIN, HIGH);
        vTaskSuspend(buttonTask_Handle);
        digitalWrite(RED_LED_PIN, LOW);
        printf("Wait a little bit...\n");
        // When the adc and MAX3130 is done reading, wait 10 sec before user can record
again
        vTaskDelay( 10000 / portTICK_PERIOD_MS );
        printf("Ready for new sample\n");
      }
    }
    vTaskDelay( 200 / portTICK_PERIOD_MS );
  }
}

// Define the task that collects analog data
void analogDataTask(void *parameter) {
  int currentIndex = 0;
  uint16_t avg;
  vTaskSuspend(analogDataTask_Handle);
  while(1) {
    uint16_t analogValue = analogRead(ADC_PIN);
    float voltage = (analogValue / 4095.0) * 3.3;
    buffer[currentIndex] = voltage;
    currentIndex++;

    if (currentIndex == ADC_BUFFER_SIZE){
      currentIndex = 0;
      printf("Buffer full\n");
      // for(i = 0; i < ADC_BUFFER_SIZE; i++){
      //   Serial.print((float)buffer[i]);
      //   Serial.print(" - ");
      //   Serial.println(i);
      // }

      bool status = writeArrayToFile(buffer, ADC_BUFFER_SIZE);
      printf("ADC Task has been suspended\n");
      digitalWrite(RED_LED_PIN, LOW);
      vTaskDelay(500/portTICK_PERIOD_MS);
      digitalWrite(RED_LED_PIN, HIGH);
      // Start BPM/SPO2 task
```

```
      vTaskResume(BPM_SPO2_Handle);
      vTaskSuspend(analogDataTask_Handle);
      printf("New ADC task in progress\n");
      currentIndex = 0;
    }
    // Wait for a short period of time to achieve a sampling frequency of about 44kHz
    vTaskDelay(1 / portTICK_PERIOD_MS);
  }
}

void BPM_SPO2(void *parameters)
{
  for(;;){
    vTaskSuspend(BPM_SPO2_Handle);
    printf("SPO2 Task Running\n");

    bufferLength = 100; // Buffer length of 100 stores 4 seconds of samples running at
25sps

    // Read the first 100 samples, and determine the signal range
    for (byte i = 0 ; i < bufferLength ; i++){
      while (particleSensor.available() == false){ // Do we have new data?
        particleSensor.check(); // Check the sensor for new data
      }

      redBuffer[i] = particleSensor.getRed();
      irBuffer[i] = particleSensor.getIR();
      particleSensor.nextSample(); // Move to next sample

      //printf("red=");
      //printf("%d",redBuffer[i]);
      //printf(", ir=");
      //printf("%d\n",irBuffer[i]);
    }
    // Calculate heart rate and SpO2 after first 100 samples (first 4 seconds of samples)
    maxim_heart_rate_and_oxygen_saturation(irBuffer, bufferLength, redBuffer, &spo2,
&validSPO2, &heartRate, &validHeartRate);
    printf("Signal Range Dertermined (MAX30101)\n");

    // Read the second 100 samples, and determine SPO2
    for (byte i = 0 ; i < bufferLength ; i++){
      while (particleSensor.available() == false) // Do we have new data?
        particleSensor.check(); // Check the sensor for new data

      redBuffer[i] = particleSensor.getRed();
      irBuffer[i] = particleSensor.getIR();
```

```
      particleSensor.nextSample(); // Move to next sample

      //printf("red=");
      //printf("%d",redBuffer[i]);
      //printf(", ir=");
      //printf("%d\n",irBuffer[i]);
    }
    // Calculate heart rate and SpO2 after first 100 samples (first 4 seconds of samples)
    maxim_heart_rate_and_oxygen_saturation(irBuffer, bufferLength, redBuffer, &spo2,
&validSPO2, &heartRate, &validHeartRate);
    printf("SpO2 Calculated\n");

    //Zero out array from last recording
    for(int i=0; i>25; i++){
      arrayBPM[i]=0;
    }

    // Read the third set of samples, and determine HR
    int i = 0;
    int j = 0;
    while(i < 25){
      while (particleSensor.available() == false) // Do we have new data?
        particleSensor.check(); // Check the sensor for new data

      long irValue = particleSensor.getIR();
      particleSensor.nextSample(); // Move to next sample

      //printf("ir=");
      //printf("%d\n",irValue);

      if (checkForBeat(irValue) == true){
        //We sensed a beat!
        long delta = millis() - lastBeat;
        lastBeat = millis();
        beatsPerMinute = 60 / (delta / 1000.0);
        arrayBPM[i] = (byte)beatsPerMinute;
        printf("%d", i);
        printf(" - ");
        printf("%d\n",arrayBPM[i]);
        i++;
      }
      j++;
      if(j>650){
        i = 25;
      }
    }
```

```
    printf("BPM Measured\n");

    //Take average of readings
    beatAvg = 0;
    count = 0;
    for (int x = 0 ; x < 25 ; x++){
      if(arrayBPM[x]>44 && arrayBPM[x]<151){
        beatAvg += arrayBPM[x];
        count++;
      }
    }
    if (beatAvg == 0) {
      beatAvg = -1;
      count = 1;
    }
    beatAvg /= count;
    printf("Average computation done\n");

    maxim_heart_rate_and_oxygen_saturation(irBuffer, bufferLength, redBuffer, &spo2,
&validSPO2, &heartRate, &validHeartRate);

    writeIntToFile(beatAvg);
    writeIntToFile2(spo2);
    printf("Integers sent over wifi");
    printf(" - bpm= %d", beatAvg);
    printf(", spo2= %d\n", spo2);
    vTaskResume(buttonTask_Handle);
  }
}

void setup() {
  // put your setup code here, to run once:
  pinMode(RED_LED_PIN, OUTPUT);
  pinMode(REC_PIN, INPUT);
  delay(2000);
  digitalWrite(RED_LED_PIN, HIGH);
  Serial.begin(115200);
  delay(8000);
  digitalWrite(RED_LED_PIN, LOW);

  Wire.begin(I2C_SDA, I2C_SCL);
  if (!particleSensor.begin(Wire, I2C_SPEED_FAST, 0x57)) // Initialize sensor
  {
    Serial.println("MAX30101 was not found.");
    while(1);
  }
```

```
  particleSensor.setup(ledBrightness, sampleAverage, ledMode, sampleRate,
pulseWidth, adcRange); // Configure sensor

  initSPIFFS();
  initWiFi();
  initWebSocket();
  initWebServer();

  xTaskCreatePinnedToCore(analogDataTask, "ECG-ADC", 8000, NULL, 2,
&analogDataTask_Handle, app_cpu);
  xTaskCreatePinnedToCore(updateClients, "Update Wifi", 8000, NULL, 1, NULL, 0);
  xTaskCreatePinnedToCore(checkButtonTask, "ButtonTask", 5000, NULL, 3,
&buttonTask_Handle, app_cpu);
  xTaskCreatePinnedToCore(BPM_SPO2, "BPM_SPO2_Calc", 8000, NULL, 1,
&BPM_SPO2_Handle, app_cpu);
  vTaskDelete(NULL);
}


void loop() {
  // put your main code here, to run repeatedly:
}
```

index.html

```
<!DOCTYPE html>
<html>
<head>
	<title>ECG Monitoring</title>
	<link rel="stylesheet" href="index.css">
</head>
<body>
  <header>
    <h1>Welcome to the ECG Monitoring Interface</h1>
  </header>
  <main>
    <section>
      <h2>Control Panel</h2>
      <!-- Current Date and Time -->
      <div class="container">
        <div class = "child">Current Time:</div>
        <div class="time">
          <span id="hours"></span> :
          <span id="minutes"></span> :
          <span id="seconds"></span>
```

```
        <span id="meridian"></span>
      </div>
    </div>
    <br>
    <div class = "container">
      <div class = "child">Data Status:</div>
      <div class = "child" id="data_status"></div>
    </div>
    <br>
    <button onclick="recieveData()" id="button_text"></button>
    <br>
    <br>
    <div class="container" id="timestamp" style="display: none;">
      <div class = "child">Data collected at:</div>
      <div class="time">
        <span id="stamp_hours"></span> :
        <span id="stamp_minutes"></span> :
        <span id="stamp_seconds"></span>
        <span id="stamp_meridian"></span>
      </div>
    </div>
    <br>
    <div class = "container">
      <h2 class = "child">Heart Rate: </h2>
      <div class = "child" id="HR"></div>
      <div class = "child"> BPM</div>
    </div>
    <br>
    <div class = "container">
      <h2 class = "child">SPO2: </h2>
      <div class = "child" id="SPO2"></div>
      <div class = "child">%</div>
    </div>
    <br>
    <button onclick="openGraphs()" id="open_graph" style="display: none;">View
Graphed Data</button>
    <br>
    <div class = "container" id="message" style="display: none;">Refresh page to make
another measurement.</div>
  </section>
  <section2>
    <h2>Graph of ECG Data</h2>
    <div id ="temp" style="display: none"></div>
    <div class="chart-container">
      <canvas id="ECG"></canvas>
    </div>
```

```
    </section2>
  </main>
  <footer>
    <p>EEG Alarm Group 4. Josh O'Brien, Jackson Bautch, Alex Beck and Megan
O'Donnell</p>
  </footer>
</body>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <script src="index.js">   </script>
</html>
```

index.css

```
@charset "UTF-8";
html, body {
  height: 100%;
  background-color: #f0f4f8;
  font-family: Roboto, sans-serif;
  font-size: 12pt;
  overflow: auto;
  color: #333;
          margin: 0;
          padding: 0;
  align-items: center;
  justify-items: center;
  overflow-x: auto;
}
.panel {
  display: grid;
  grid-gap: 3em;
  justify-items: center;
}
header {
  background-color: #1c3d5a;
  color: #fff;
  padding: 20px;
  text-align: center;
}
h1 {
  margin: 0;
  font-size: 40px;
  text-align: center;
}
button {
  padding: .5em .75em;
  font-size: 1.2rem;
```

```
  color: #fff;
  text-shadow: 0 -1px 1px #000;
  border: 1px solid #000;
  border-radius: .5em;
  background-image: linear-gradient(#2e3538, #73848c);
  box-shadow: inset 0 2px 4px rgba(255, 255, 255, 0.5), 0 0.2em 0.4em rgba(0, 0, 0,
0.4);
  outline: none;
  margin: 0 auto;
}
.chart-container {
  width: 1000px;
  height:600px;
}
.time {
  display: inline-block;
}
.child {
  display: inline-block;
}
.divScroll {
  overflow:scroll;
  }
main {
  padding: 10px;
  display: flex;
  flex-wrap: wrap;
  justify-content: space-between;
}
section {
  background-color: #fff;
  box-shadow: 0 2px 4px rgba(0,0,0,0.1);
  margin-bottom: 10px;
  width: 18%;
  padding: 10px;
  text-align: center;
}
section2 {
  background-color: #fff;
  box-shadow: 0 2px 4px rgba(0,0,0,0.1);
  margin-bottom: 10px;
  width: 75%;
  padding: 10px;
  text-align: center;
  overflow:scroll;
}
```

```
section h2 {
  color: #3c6e8e;
  font-size: 24px;
  margin: 0;
  margin-bottom: 10px;
}
section p {
  margin: 0;
}
footer {
  background-color: #1c3d5a;
  color: #fff;
  padding: 20px;
  text-align: center;
}
```

index.js

```
var gateway = `ws://${window.location.hostname}/ws`;
var websocket;
// Initialization
window.addEventListener('load', onLoad);

function onLoad(event) {
    initWebSocket();
    initButton();
}

// WebSocket handling
function initWebSocket() {
    console.log('Trying to open a WebSocket connection...');
    websocket = new WebSocket(gateway);
    websocket.onopen    = onOpen;
    websocket.onclose   = onClose;
    websocket.onmessage = onMessage;
}

function onOpen(event) {
    console.log('Connection opened');
}

function onClose(event) {
    console.log('Connection closed');
    setTimeout(initWebSocket, 2000);
}
```

```
function onMessage(event) {
   let data = JSON.parse(event.data);
   document.getElementById('led').className = data.status;
}

let hours = document.getElementById("hours");
let minutes = document.getElementById("minutes");
let seconds = document.getElementById("seconds");

function updateTime() {
        let date = new Date();
  if (date.getHours()>12){
          hours.innerText = date.getHours() % 12;
    meridian.innerText = "PM";
  } else{
    hours.innerText = date.getHours();
    meridian.innerText = "AM";
  }
        // Add leading zero to minutes if necessary
  if (date.getMinutes() < 10) {
    minutes.innerText = "0" + date.getMinutes();
``} else{
    minutes.innerText = date.getMinutes();
  }
  // Add leading zero to seconds if necessary
  if (date.getSeconds() < 10) {
    seconds.innerText = "0" + date.getSeconds();
``} else{
    seconds.innerText = date.getSeconds();
  }

}
setInterval(updateTime, 1000);

var data_status = "Waiting";
document.getElementById("data_status").innerHTML = data_status;
var button_text = "Click to Recieve Data";
document.getElementById("button_text").innerHTML = button_text;
var temp = new Array(5000);
var data = new Array(5000);
var labels = new Array(5000);
for(let i=0; i<5000; i++) {
  labels[i]=i;
}
```

```
function recieveArray(){
  fetch('/temp.txt')
  .then(response => response.text())
  .then(jsonString => {
    const data = JSON.parse(jsonString);
    let i=0;
    data.forEach(value => {
      const div = document.createElement('div');
      div.textContent = value;
      temp[i]=value;
      i++;
    });
    document.getElementById("temp").innerHTML = temp;
    data = temp;
  });

}

let mydata;
function recieveHR(){
  fetch('/heartRate.json')
  .then(response => response.json())
  .then(data => {
      mydata = data;
    });
}


let mydata2;
function recieveSPO2(){
  fetch('/SPO2.json')
  .then(response => response.json())
  .then(data => {
      mydata2 = data;
    });
}

function recieveData(){
  recieveArray();
  timestamp();
  document.getElementById("timestamp").style.display = "block";
  document.getElementById("open_graph").style.display = "block";
  recieveHR();
  button_text = "Click to view HR and SPO2";
  document.getElementById("button_text").innerHTML = button_text;
  data_status = "Recieved All Data";
```

```
  document.getElementById("data_status").innerHTML = data_status;
  recieveSPO2();
  document.getElementById("HR").innerHTML = mydata.value;
  document.getElementById("SPO2").innerHTML = mydata2.value;
  document.getElementById("button_text").innerHTML = "All data Recived";
  document.getElementById("button_text").style.backgroundImage = "linear-
gradient(#f8f9fa, #e9ecef)";
  document.getElementById("button_text").style.color = "black";
  document.getElementById("button_text").style.textShadow = "none";
  document.getElementById("message").style.display = "block"
}

let stamp_hours = document.getElementById("stamp_hours");
let stamp_minutes = document.getElementById("stamp_minutes");
let stamp_seconds = document.getElementById("stamp_seconds");
let stamp_meridian = document.getElementById("stamp_meridian");
function timestamp(){
  let date = new Date();
  if (date.getHours()>12){
            stamp_hours.innerText = date.getHours() % 12;
    stamp_meridian.innerText = "PM";
  } else{
    stamp_hours.innerText = date.getHours();
    stamp_meridian.innerText = "AM";
  }
         // Add leading zero to minutes if necessary
  if (date.getMinutes() < 10) {
    stamp_minutes.innerText = "0" + date.getMinutes();
``} else{
    stamp_minutes.innerText = date.getMinutes();
  }
  // Add leading zero to seconds if necessary
  if (date.getSeconds() < 10) {
    stamp_seconds.innerText = "0" + date.getSeconds();
``} else{
    stamp_seconds.innerText = date.getSeconds();
  }

}

function openGraphs(){
  let graph1 = new Chart(document.getElementById("ECG"), {
       type: "line",
       data: {
         labels: labels,
         datasets: [
```

```
        {
          label: "ECG Samples",
          data: temp,
          borderColor: "red",
          fill: true,
        },
      ],
    },
    options : {
      scales: {
        y: {
          display: false,
          title: {
            display: false,
            text: 'voltage'
          }
        },
        x: {
          title: {
            display: true,
            text: 'ms'
          }
        }
      }
    }
  });
  document.getElementById("open_graph").innerHTML = "Graphs opened";
  document.getElementById("open_graph").style.backgroundImage = "linear-
gradient(#f8f9fa, #e9ecef)";
  document.getElementById("open_graph").style.color = "black";
  document.getElementById("open_graph").style.textShadow = "none";
 }
```