Software Design Document

Biomedical Sensor Board for Education - MediBrick 2000

ENGR 498B - #24052

Project Leader: Maria Carmella Ocaya (Biomedical Engineer)

Project Procurement Leader: Muad Alsayar (Electrical Engineer)

Michael Chase Morrett (Biosystems Engineer)

Daniel Fabricio Campana Moscoso (Electrical Engineer)

William Alec Newman (Mechanical Engineer)

Sponsor: Urs Utzinger, PhD

Mentor: Steve Larimore

Revision #10

REVISION PAGE

REVISION#	REVISION DATE	REVISION DESCRIPTION	REVISED BY
1	10/17/2023	Created and setup the document	Carmella Ocaya
2	1/17/2024	Added multiple sections	Carmella Ocaya
3	3/30/2024	Edited figure 1	Carmella Ocaya
4	3/31/2024	Updated Sound Sensor Code	Daniel Campana
5	3/31/2024	Added the impedance code	Carmella Ocaya
6	4/30/2024	Updated all the sensor codes (Addresses, clean up)	Carmella Ocaya
7	4/30/2024	Updated the CSCI Diagram	Carmella Ocaya
8	4/30/2024	Updated the STATUS Section	Carmella Ocaya
9	4/30/2024	Update Sound Sensor Code Descriptions	Daniel Campana
10	4/30/2024	Finalized Details	Carmella Ocaya

TABLE OF CONTENTS **Page No.**

1.0 SCOPE

1.1 IDENTIFICATION

The Biomedical Sensor Board (MediBrick 2000) is a sensor board array that measures live physiological signals. This device is an expandable, low-cost, open-design system for anyone who is a senior-standing engineering student and above with soldering and 3D modeling experiences to be able to replicate and repair. This device is designed for laboratory settings to create an educational system for students to learn about these physiological signals. These signals are temperature, electrocardiography, heart and lung sounds, body fat percentage, water composition, heart rate, and plethysmograph. This device aims for the user to learn how to read, interpret, and manipulate the resulting signals.

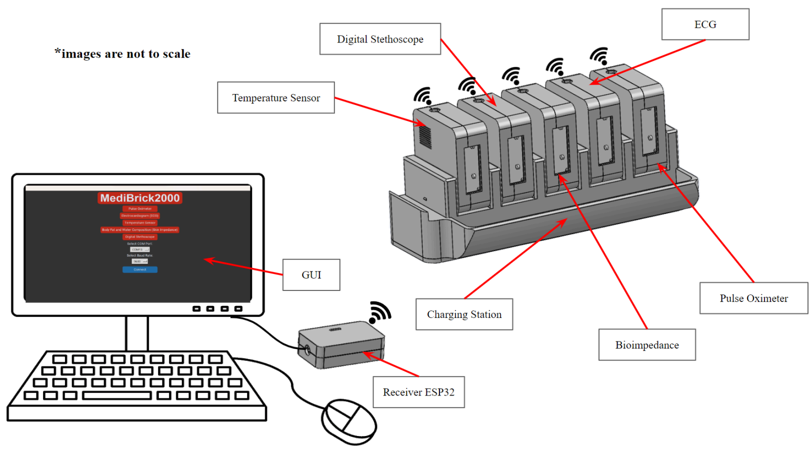

Figure 1. MediBrick Concept Design

1.2 SYSTEM OVERVIEW

The sensors that are carefully chosen to achieve these physiological signals are the AD5933 (Impedance Sensor), AD8232 (ECG Sensor), 527-MA100GG103AN (Temperature Sensor), AFE4490 (Pulse Oximeter), and MAX9814 (Microphone for sound). The sensors will have individual ESP32-S3 Feather microcontrollers to utilize the ESP's capability of performing wireless communication. These sensors will send their signal readings to a separate ESP32-S3 Feather which is connected to the user's computer. The user will then control the received data using a graphical user interface (GUI). The GUI will have multiple abilities such as live graphs, screen capture, and saving options in Excel.

1.3 DOCUMENT OVERVIEW

The Software Design Document describes the purpose, functionality, architecture, interfaces, libraries, protocols, and assembly of the MediBrick 2000. This includes the specifications of the ESP32-S3 Feather, the Arduino codes for each sensor, the wireless communication protocol, the Python Code for the GUI, the battery and OLED system, and more. This document will also cover the aspects regarding safety and privacy requirements since this device involves the recording of health-related information.

2.0 REFERENCED DOCUMENTS

- ESP32-S3 Feather Datasheet:
https://cdn-learn.adafruit.com/downloads/pdf/adafruit-esp32-s3-feather.pdf

- ESP-NOW Protocol:
https://www.espressif.com/en/solutions/low-power-solutions/esp-now#:~:text=ESP%2DNOW%20is%20a%20wireless,ESP32%2DC%20series%20of%20SoCs.

- Single-Lead, Heart Rate Monitor Front End AD832 Datasheet:
https://cdn.sparkfun.com/datasheets/Sensors/Biometric/AD8232.pdf

- 1 MSPS, 12-Bit Impedance Converter, Network Analyzer AD5933 Datasheet:
https://www.analog.com/media/en/technical-documentation/data-sheets/ad5933.pdf

3.0 CSCI - WIDE DESIGN DECISIONS

Figure 2 shows the CSCI Diagram of the device. In this diagram, we can see that the device revolves around the ESP32-S3 Feather microcontrollers that are attached to each sensor and the user's computer. The team suggested the use of the ESP32 to the sponsor, and the sponsor was convinced and proposed this specific model. Unlike the other ESP32 models, the ESP32-S3 Feather has additional unique features. Here are the specifications that the ESP32-S3 Feather has to offer:

- ESP32-S3 Dual Core 240MHz Tensilica processor
- Power options - USB type C or Lipoly battery
- Built-in battery charging when powered over USB-C
- LiPoly battery monitor - MAX17048 chip actively monitors your battery for voltage and state of charge/percentage reporting over I2C
- Reset and DFU (BOOT0) buttons to get into the ROM bootloader
- STEMMA QT connector for I2C devices
- On/Charge/User LEDs
- Low Power friendly
- Works with ESP-IDF and Arduino: This includes the ESP-NOW Protocol for wireless communication.

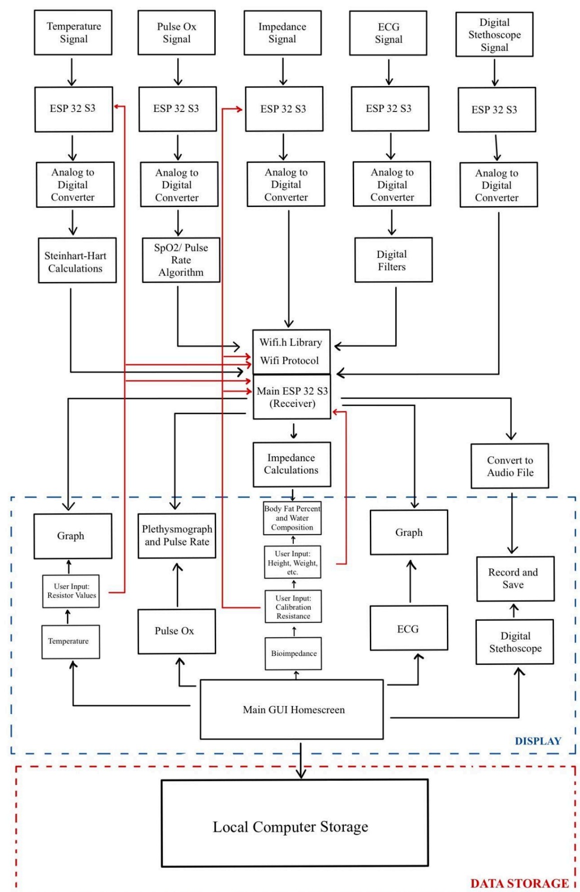

Figure 2. MediBrick 2000 CSCI Diagram

3.1 INPUTS AND OUTPUTS

3.1.1 Electrocardiography (ECG)

<u>Input:</u>

For the ECG sensor, the only GUI user input required is to click the “ECG” option in the main menu. As for the signal input, it is collected from the user using 3-Snap electrodes and disposable electrodes. The disposable electrodes will be attached to the user’s right arm, left arm, and right leg. With the provided 3-Snap electrodes for this device, connect the black cable to the user’s right arm, the blue cable to the left arm, and the red cable to the right leg. It’s best to remember that the cable colors may differ with different brands. It is best to check with the manufacturer if electrodes are not provided with the device.

Other than the GUI and signal inputs, the ECG will also have a button. Users will have the ability to provide input by pressing the button or holding it for 5 seconds. These two inputs will provide different outputs.

<u>Output:</u>

The ECG sensor will output voltage levels corresponding to what the electrodes detect from the user. To avoid inaccurate results, the Arduino code for the ECG's ESP32-S3 Feather includes a lead-off detection which stops the sensor from outputting readings when the electrodes are not connected to a user. Within the GUI, the page that corresponds to the ECG will output a graph of voltage vs. time as well as the voltage itself. Users will have the option to save the results into an Excel file.

Lastly, regarding the ECG button that was previously mentioned, pressing the button will change the OLED display from displaying the battery percentage to the ECG voltage. However, holding it for 5 seconds will put the ESP32-S3 Feather connected to the ECG into deep sleep mode.

3.1.2 Digital Stethoscope

<u>Input:</u>

For the digital stethoscope (sound sensor), the only input in regards to the GUI is for the user to click on the "Digital Stethoscope" option in the main menu. As for the signal input, the sound sensor will have a diaphragm and a bell attached to its module. Users will place this on the chest, near the heart, to listen to the subject's heartbeat, or around different points of the torso to listen to their subject's lungs.

Other than the GUI and signal inputs, the digital stethoscope will also have a button. Users will have the ability to provide input by pressing the button or holding it for 5 seconds. These two inputs will provide different outputs.

<u>Output:</u>

The sound sensor will output audio signals corresponding to where the diaphragm and bell are placed. The GUI will output a graph of the audio signal as well as real-time sound recording. Users will have the option to save this recording into a wav file.

Lastly, regarding the previously mentioned button, pressing the button will change the OLED display from displaying the battery percentage to the audio signal. However, holding it for 5 seconds will put the ESP32-S3 Feather connected to the Sound Sensor into deep sleep mode.

3.1.3 Temperature

<u>Input:</u>

In regards to inputs, the user will also have the option to pick "temperature" in the main menu. However, for the temperature sensor, the GUI will ask the users for three resistance values from the circuit of the temperature sensor. Instructions on how to attain these values will be given in the user manual. Other than the GUI, the temperature sensor uses a probe for the user to use, which provides the signal input into the ESP32-S3 Feather of the temperature sensor.

Other than the GUI and signal inputs, the temperature sensor will also have a button. Users will have the ability to provide input by pressing the button or holding it for 5 seconds. These two inputs will provide different outputs.

Output:

The temperature sensor will output resistance values which the ESP32-S3 Feather will calculate into degrees Celsius. This will then be displayed in the GUI as a graph as well as the actual value. Users will have the option to save this data into an Excel file.

Lastly, regarding the previously mentioned button, pressing the button will change the OLED display from displaying the battery percentage to the temperature in degrees Celsius. However, holding it for 5 seconds will put the ESP32-S3 Feather connected to the Temperature into deep sleep mode.

3.1.4 Impedance

Input:

For the impedance sensor, similar to the other sensors, it will also have the option "BioImpedance" in the main menu of the GUI. When users click on this option, the GUI will ask for their input regarding the use of a calibration resistor. If users would like to use said calibration resistor, they will need to input a resistance value as well as press a button on the GUI that flips the switch from "tissue sample" to the calibration resistor. However, if the user prefers to use a tissue sample, they will be required to press a button on the GUI to make sure that the switch is on the tissue sample circuitry. The GUI will then ask the user's input in regards to their height and weight. As for signal inputs, the impedance sensor uses 2-snap electrodes to measure tissue impedance.

However, other than the software input, the impedance PCB will also have an optional user input that affects its functionality. Users will have the option to use one 2-snap electrode or two 2-snap electrodes. Each option has its advantages and disadvantages depending on the sample. If the user was to choose one or the other, they would need to detach a couple of jumpers within the PCB. Instructions will be provided in the user manual.

Other than the GUI and signal inputs, the impedance sensor will also have a button. Users will have the ability to provide input by pressing the button or holding it for 5 seconds. These two inputs will provide different outputs.

Output:

The impedance sensor will output impedance values. These values will then be calculated by the Python code to output body fat percentage and water composition on the GUI. The GUI will also display a graph of the impedance values.

Lastly, regarding the button that was previously mentioned, pressing the button will change the OLED display from displaying the battery percentage to the impedance values. However, holding it for 5 seconds will put the ESP32-S3 Feather connected to the Impedance into deep sleep mode.

3.1.5 Pulse Oximetry

Input:

For the pulse oximeter sensor, the user will also have the option to pick "Pulse Oximetry" in the main menu of the GUI. The pulse oximeter will use a 9-pin SpO2 sensor to collect user signal input. This probe will be clipped on the subject's index finger.

Other than the GUI and signal inputs, the pulse oximeter sensor will also have a button. Users will have the ability to provide input by pressing the button or holding it for 5 seconds. These two inputs will provide different outputs.

Output:

The pulse oximeter sensor will output red light and near-infrared light absorbances. Using these values, the ESP32-S3 Feather will then calculate the blood oxygen saturation of the subject as well as pulse rate using the library protocentral_afe44xx.h. These four results will then be displayed on the GUI. The red light and near-infrared light absorbances will also be displayed as a live graph. Users will have the option to save all these values into an Excel File.

Lastly, regarding the previously mentioned button, pressing the button will change the OLED display from displaying the battery percentage to the blood oxygen saturation and pulse rate. However, holding it for 5 seconds will put the ESP32-S3 Feather connected to the Pulse Ox into deep sleep mode.

3.1.6 GUI Miscellaneous

Input:

The GUI's home page requires a user input to provide a username.

Output:

Using the entered username, the Python program will create a folder locally in the same location as the program. This folder will be named after the entered username. This folder will have five subfolders named "ECG", "Temperature", "Impedance", "Digital Stethoscope", and "Pulse Oximeter". Whenever the user decides to save any of the data shown in the GUI, these will be saved in their corresponding subfolders.

Input:

The user has the option to pick which port to read data from.

Output:

Depending on the user's input, the Python program will read from that port.

Input:

The user can choose the baud rate they prefer.

Output:

The ESP32-S3 Feather will output data at the rate the user chooses.

3.2 RESPONSE

3.2.1 Electrocardiography (ECG)

When the AD8232 (ECG sensor) detects that the electrode leads are attached, it extracts biopotential signals. The electrical signals picked up by the electrodes are very weak, on the order of microvolts, so the AD8232 amplifies these weak signals to a level that can be more easily processed. The AD8232 also includes an instrumentation amplifier that helps amplify the signal while rejecting common-mode noise. The amplified signal may contain unwanted noise or interference, which the AD8232 will filter to remove noise and isolate the relevant ECG signals. Lastly, the processed and filtered ECG signal is then provided as an output.

After the sensor response, the resulting signal is sent into the ESP32-S3 Feather that is connected to the AD8232. This microcontroller adds a digital filter to remove additional noise from the ECG signal. After filtering, using the ESP-NOW protocol, the result from the ESP32-S3 Feather is sent wirelessly to the ESP32-S3 Feather that is connected to the user's computer. The GUI, which is in Python, will then read the serial port and detect the data coming from the ECG sensor. This data is the resulting output.

3.2.2 Digital Stethoscope

The data collection starts at the stethoscope diaphragm. After the heartbeat is detected and sent through the tubing, the signal is captured electronically by the microphone. The amplitude of the signal is considerably small and almost impossible to hear, so the signal goes through a pre-amplification stage. The signal may contain noise at different high frequencies, so it passes through an analog low pass filter which attenuates frequencies above 1000 Hz. Then, the signal is digitized with the ES8388 and later sent into the ESP32-S3 Feather.

Once the ESP32-S3 receives the digital signal, it executes a IIR digital low pass filter which also attenuates frequencies above 1000 Hz to remove any remaining noise. After filtering, the signal is sent wirelessly to the ESP32-S3 Feather that is connected to the user's computer. This second ESP32-S3 Feather forwards the digital signal to the user's computer, and it is read with the Python-coded GUI. Finally, the program produces a real-time graph and a real-time sound recording with the signal.

3.2.3 Temperature

The temperature sensor response begins once the user chooses the temperature option in the GUI. When the temperature sensor is exposed to heat, its resistance changes. This causes the voltage across the branch of the Wheatstone bridge to change, which is measured by the Arduino code: vdiff=vb-vd. This voltage difference is then converted to resistance by R4 = R3/(R2/(R1+R2)-vdiff/vin)-R3. This resistance is then converted to temperature in Kelvin by Steinhart-Hart equation = 1/A+Blog(R) + Cpow(log(R), 3), which will then be converted to Celsius using the equation temp=temp-273.15. The ESP32-S3 Feather of the temperature sensor will accumulate 100 values together, and it averages these values. The results will then be rounded to the second decimal place, which is sent to the ESP32-S3 Feather connected to the user's computer. This value will then be displayed to the GUI.

3.2.4 Impedance

The impedance sensor starts with the multiple user inputs needed for its functionality. Once the program is ready, the user will attach the electrodes for the impedance sensor. The AD5933 (Impedance sensor) has a frequency generator that allows an external complex impedance to be excited with a known frequency. This is sent through the electrodes. The response signal from the impedance is sampled by the on-board ADC and a discrete Fourier transform (DFT) is processed by an on-board DSP engine. The DFT algorithm returns real (R) and imaginary (I) data at each output frequency. The magnitude of the impedance at each frequency point along the sweep is then calculated using the ESP32-S3 Feather that is attached to the AD5933.

Once an impedance value is calculated, this data is sent from the ESP32-S3 Feather of the impedance sensor to the ESP32-S3 Feather that is attached to the user's computer. This data is then read by the Python program of the GUI. This Python program will then calculate the body fat percentage and water composition using the inputs of the user along with the detected impedance. These will then be displayed on the GUI.

3.2.5 Pulse Oximetry

The pulse oximeter sensor begins the process of detection at the subject's ring finger where the pulse oxi sensor is attached and sends pulses of IR and Red waves. These pulses pass through the finger and are received by a photodetector as an analog signal. The analog signal is sent to the (AFE4490 Integrated Analog Front-End for Pulse Oximeters) IC where the analog signal is converted to a digital signal and transferred to ESP32. The process of sending the pulses through the finger and receiving the information from the pulse oxi sensor is achieved by using SPI communication with the ESP32-S3 Feather.

Once the ESP32 receives the Digital signal that was converted by the AFE4490 IC it goes through an Algorithm that uses the IR and Red values from the pulse oximeter sensor to calculate the heart rate and blood oxygen of the subject. After processing the data by the ESP32 (IR and Red values, BPM, and SPO2), the data is sent wirelessly and received by the master ESP32 attached to the lab computer. This data is then collected by the Python GUI and a graph of real time of the subject is displayed of IR and Red values vs time graph. As well the heart rate and blood oxygen is displayed on the Gui which would be the resulting output.

3.3 APPEARANCE

The GUI appearance was decided by the team along with the sponsor. The main components of the GUI required by the sponsor are the main menu options that list each sensor, live graphical charts, and the ability to save locally. However, the team decided to add additional components and options to improve the user experience.

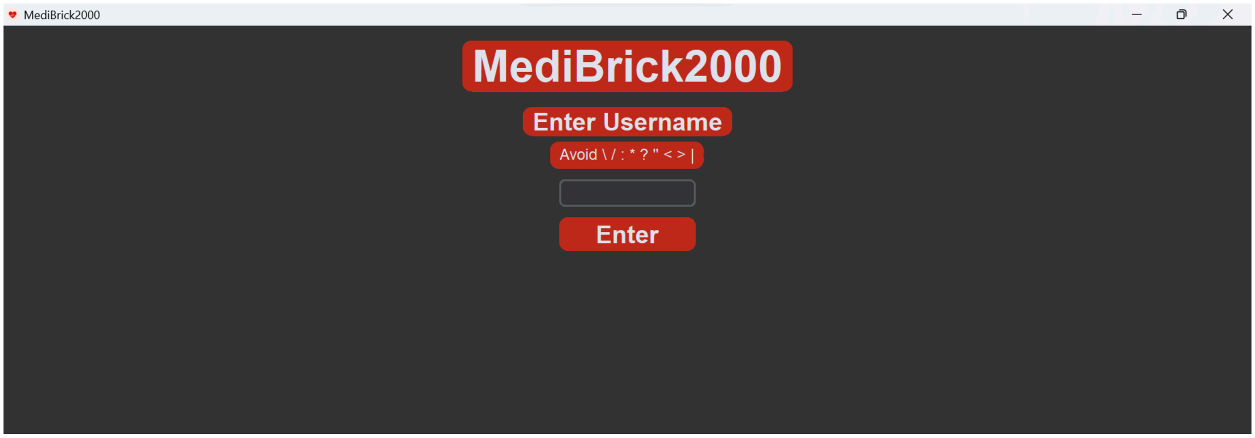

Figure 3. MediBrick 2000 Homepage

As we can see in Figure 3., the GUI starts with the home page, which will ask the user to enter a username. As explained before, this is meant for the saving mechanism of the GUI. This decision was made for users to find their files more easily.

Figure 4. MediBrick2000 Main Menu

Next, the GUI will have a main menu that lists every sensor as shown in Figure 4. From this menu, the user will choose the sensor of their liking.

In the following figures, the pages for each sensor will be shown. All of the sensor pages will have reading and recording options. All the sensor pages will also have the port options as well as the baud rate options.

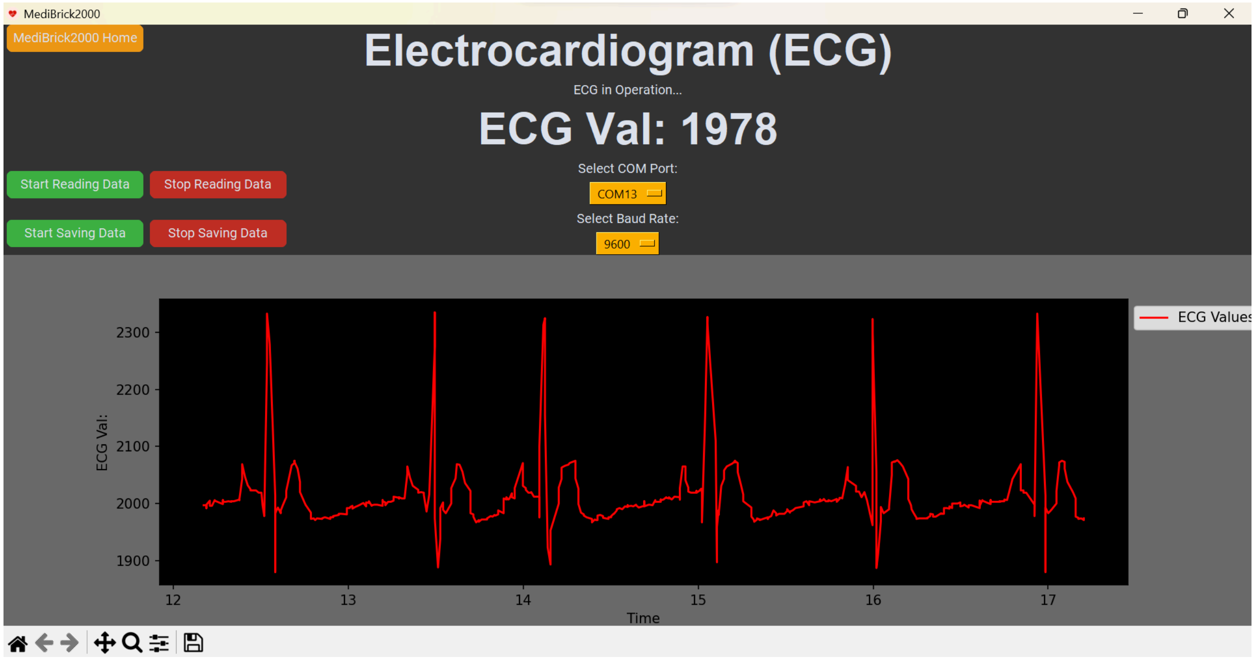

Figure 5. MediBrick 2000 ECG GUI page

Figure 5. displays the ECG page of the GUI. On this page, a live graph is shown corresponding to the measured voltage. This said voltage is also shown.

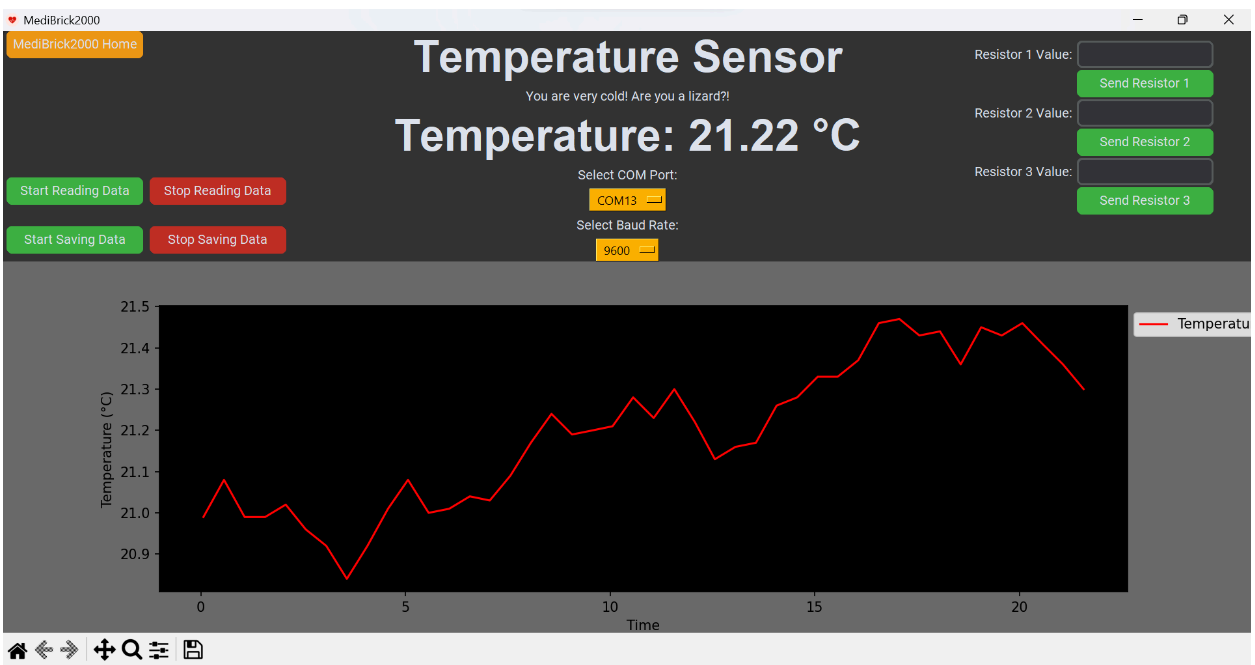

Figure 6. MediBrick 2000 Temperature GUI page

Figure 6. shows the corresponding GUI page for the temperature sensor. This also shows a live graph of the temperature measured by the sensor as well as displays said temperature.

However, the temperature GUI page has inputs for the resistance required for a more accurate measurement.

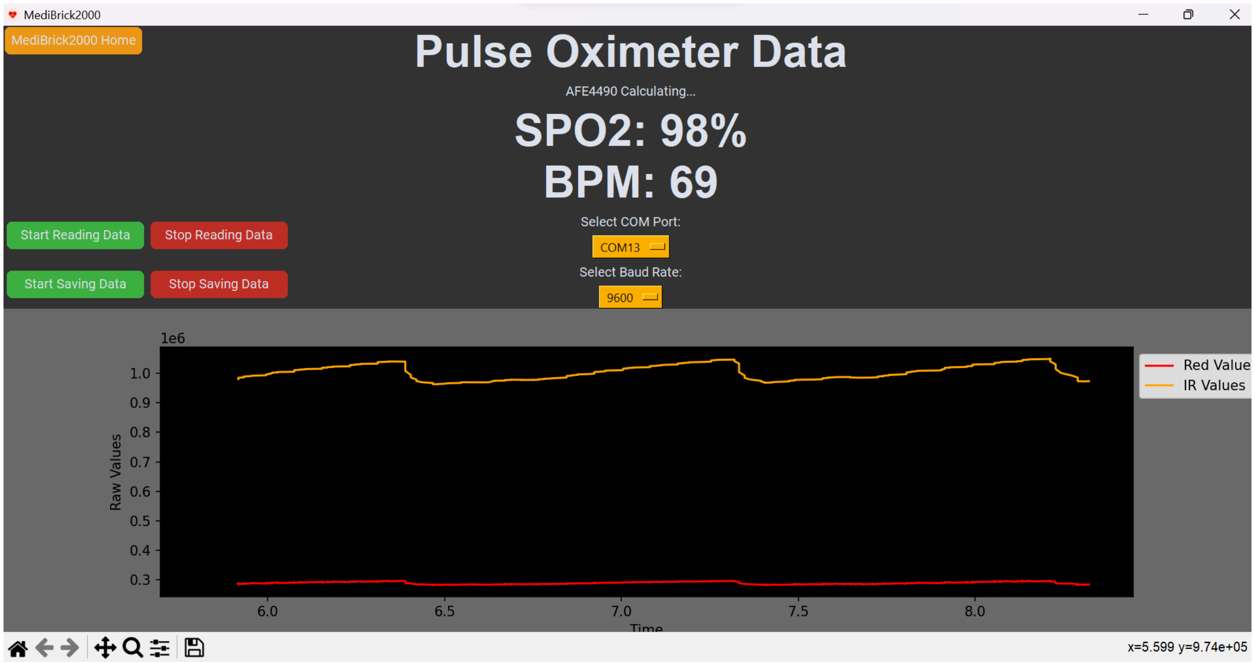

Figure 7. MediBrick 2000 Pulse Oximeter GUI page

Figure 7. displays the GUI page for the Pulse Oximeter. This page has four incoming data with a live graph for the red and IR values. BPM and SpO2 are shown on top.

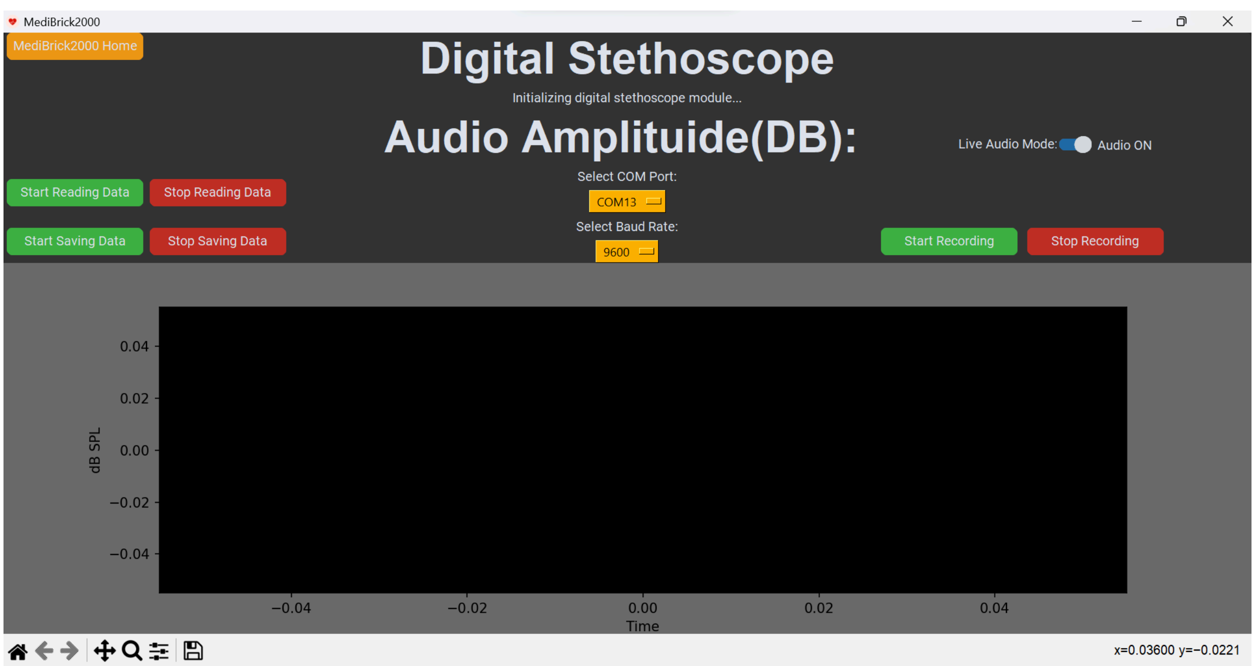

Figure 8. MediBrick 2000 Digital Stethoscope GUI page

Figure 8. shows the dedicated GUI page for the digital stethoscope. This page shows a live graph for the incoming audio amplitude as well as the ability to listen to the recording live.

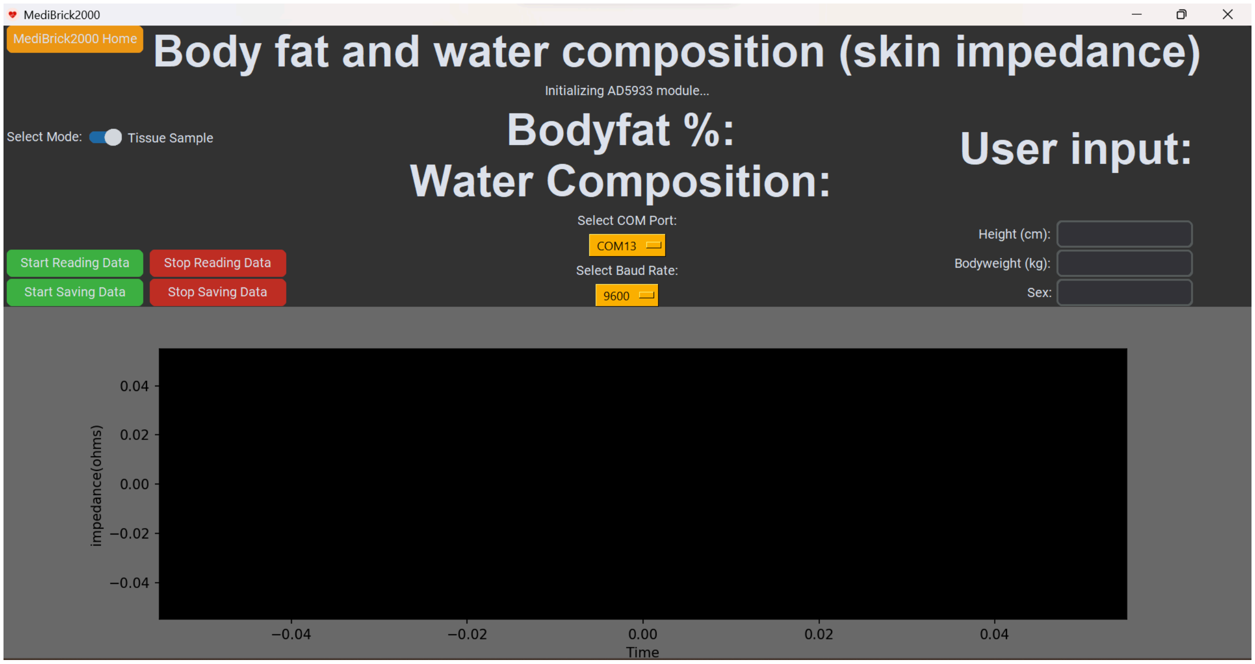

Figure 9. MediBrick 2000 Impedance GUI page

Lastly, Figure 9. is the dedicated GUI page for the impedance sensor. Just like the other sensors, this page will also have a live graph for the incoming impedance values. However, body fat percentage and water composition are displayed as values instead. This GUI page will also ask for user input for proper calculations.

3.4 SAFETY, SECURITY, AND PRIVACY REQUIREMENTS

The biggest concern about the safety of this device is the use of electrodes with electronics. At the beginning of this project, the team took precautions to keep the device as safe as possible from electricity-related accidents by making each sensor wireless. The batteries used to power these sensors are 1200 mAh 3.7V batteries, which is merely a fraction of the power produced by directly connecting to a wall plug. In addition to the wireless communication, the impedance and ECG sensors, which are the ones using electrodes, were thoroughly designed to keep the current as low as possible before entering the leads. These devices were also tested on physical simulators or circuits before live tissue.

As for safety requirements for the users of the device, it will be heavily advised to remove every sensor from the charging station before testing it on anyone. There will also be laboratory-related rules in the user manual such as no liquid or food nearby.

Furthermore, since this project involves the reading and recording of health-related data, the team needed to take precautions and thorough analysis of how to approach the security and privacy of the health records taken by this device. At the beginning of the project's development, the team was advised to seek out IRB approval. However, after further discussions with an IRB representative, it was decided that this project is not considered a research project. This means that IRB approval was not required for the development of this device.

Upon discussing this issue with the sponsor, the team decided to require every member to take the annual HIPPA certification offered in EDGE Learning through the University of Arizona. This can protect the members of the team in regards to the handling of health records, as well as further increase the team's understanding of the importance of health record privacy.

More requirements were placed by the team in regard to collecting data from individuals outside of the group. To protect the confidentiality of the subjects who are volunteering to become part of the device's development, the team will keep the collected data in a separate flash drive. The collected data will also not be saved with the subject's name. A number will be used as an identifier instead. Collected data during this stage will not be saved locally in anyone's computer, and will not be accessed or saved through the internet.

Outside of the device's development, a different set of requirements are placed in regard to the user of this device. If this device were to be used in a classroom setting here at the University of Arizona, the students who are enrolled in this said class would be required to take the same annual HIPPA certification that the team members took. There may be additional certifications needed such as the Human Subjects certification depending on the professor teaching the class. Other than certifications, the GUI made for this device also takes some precautions. By making the GUI an app that works locally, recorded data will only be accessed on the student's personal computer and not online. In case the student does not have a personal computer, the University of Arizona computers are also programmed to erase locally saved data once the user logs out of their account.

3.5 FLEXIBILITY, AVAILABILITY, AND MAINTAINABILITY

As mentioned in the introduction of this document, this device is open-design. Not only that but it is designed to be easily replicated and maintained. All documents related to the development of this project will be released in a GitHub repository online for anyone to replicate. These documents will include the Arduino codes for each sensor, the GUI code, the additional MATLAB codes for analysis, the 3D models for the housing, the schematics and PCB designs, and the user manual of the device.

4.0 CSCI ARCHITECTURAL DESIGN

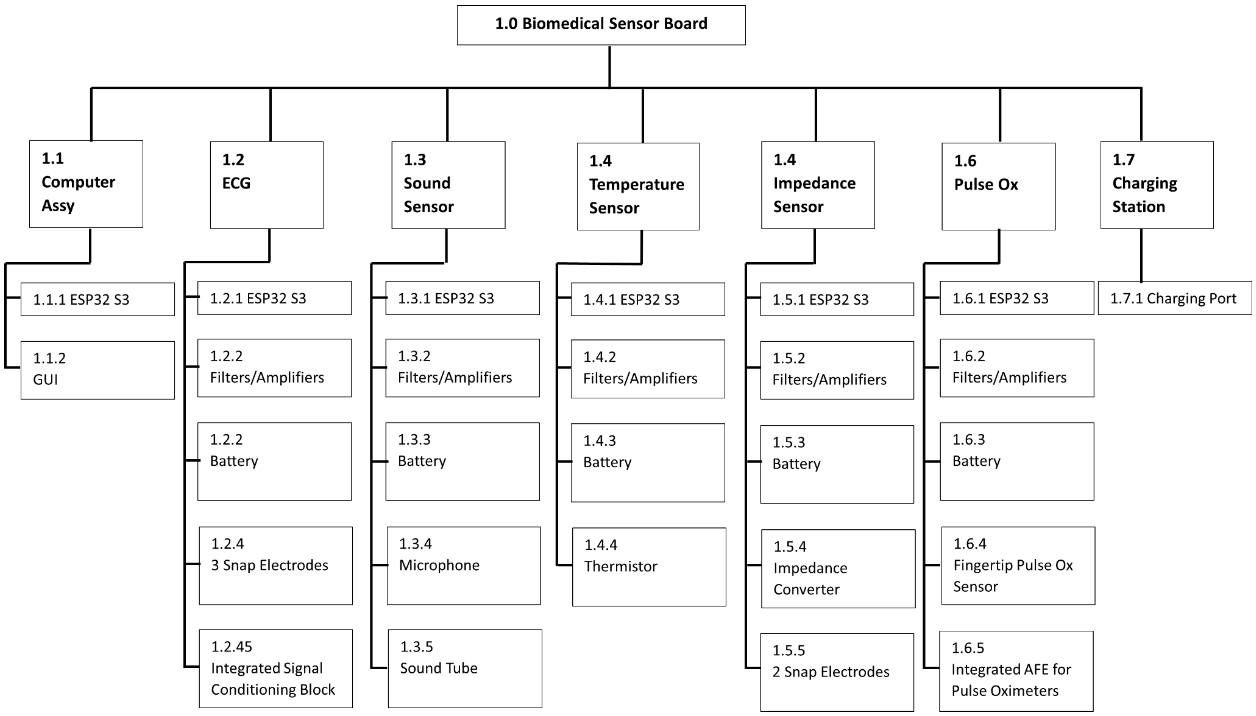

Figure 10. MediBrick 2000 System Architecture

4.1 CSCI COMPONENTS

4.11 IDENTIFICATION

For this device, the two computer languages used for the end product are Python and a variant of C++ (Arduino). However, during the development of each sensor, MATLAB also plays a role in the analysis stage. The core component of this project is the ESP-NOW protocol that is available with the ESP32-S3 Feather microcontroller. This protocol performs ESP-to-ESP communication wirelessly which is one of the most important aspects of this device.

In terms of the GUI, the current design takes advantage of the customtkinter library. This library is a modern twist of the class tkinter library. With this library, the GUI was able to carry out a newer look compared to what tkinter has to offer. Lastly, the live graphs, which is another important aspect of the GUI, are displayed using the matplotlib library.

4.12 STATIC RELATIONSHIP

In terms of the GUI, most components that has a static relationship are typically involved with the aesthetics of the full design. This includes the font sizes, font colors, widget colors, and the overall theme.

As for the individual sensors, the main static component that will remain constant for all the sensors' ESP32-S3 Feathers is the receiver ESP32-S3 Feather's address. This is an important component for the wireless communication of the sensors.

4.13 PURPOSE

The main purpose of the GUI is to display live data from any of the five sensors. The GUI is also enables users to export data for further analysis or share specific views of the GUI with colleagues. This promotes collaboration and facilitates external analysis of the sensor data. Not only that, but the GUI integrates additional details about each sensor, including specifications, calibration status, or any relevant data, which can aid users in better understanding the context of the data being presented.

4.14 STATUS

CSCI Component	**STATUS**
ECG Arduino	Done
Temperature Arduino	Done
Pulse Ox Arduino	Done
Impedance Arduino	Done
Sound Arduino	Done
GUI	Done

4.15 LIBRARIES

- Protocentral_afe44xx.h: https://github.com/Protocentral/protocentral-afe4490-arduino
- Arduino Audio Tools: https://github.com/pschatzmann/arduino-audio-tools
- Arduino Audio Driver: https://github.com/pschatzmann/arduino-audio-driver
- ESP-NOW Library: https://github.com/yoursunny/WifiEspNow
- Customtkinter: https://github.com/TomSchimansky/CustomTkinter
- Matplotlib: https://matplotlib.org/
- Math.h: https://www.tutorialspoint.com/c_standard_library/math_h.htm

4.2 CONCEPT OF EXECUTION

The concept of execution for the GUI involves the step-by-step process of running the application, wirelessly collecting data from the five sensors through ESP32-S3 Feather microcontrollers, and displaying the information in real-time on the graphical user interface. Here is a step-by-step concept of execution for this device:

1. The first step is initialization. Initialize the GUI application by setting up necessary resources, libraries, and configurations.

2. Next is setting up the ESP32. Establish communication between the sensors and the receiver ESP32 using the ESP NOW wireless protocol. Also configure each ESP32 device to read data from its respective sensor and transmit it wirelessly.
3. The next step is widget creation. Create GUI widgets (e.g., labels, charts) to visually represent each sensor's data. Thes includes setting up a layout to organize and display the widgets effectively.
4. This is followed by event binding. Bind events to trigger data retrieval from ESP32 devices. This could include periodic updates or triggered updates based on user interaction.
5. Afterwards, implement a loop or event mechanism to continuously fetch and update sensor data in real-time. Display the live data on the GUI, ensuring that users can monitor the latest readings.
6. After the live data display is the data saving feature. This means the implementation of the functionality to allow users to save the displayed data. This device involves storing data in a local database, The GUI also provide a user interface for initiating and managing the data-saving process.
7. Lastly, there are other concepts of execution is user interaction. This GUI allow users to interact with the GUI, such as selecting specific sensors, adjusting display settings, or triggering actions like data saving.

4.3 INTERFACE DESIGN

4.31 IDENTIFICATION AND DIAGRAMS

The GUI application is identified as a real-time data monitoring and visualization tool for the five sensors connected wirelessly using ESP32-S3 Feather microcontrollers. The diagrams that the GUI will produce will display live data from any of the five sensors.

In Figure 11., this system diagram illustrates the components involved, including the GUI interface, ESP32-S3 Feather microcontrollers, and the wireless communication link. It shows the flow of data from sensors to the GUI.

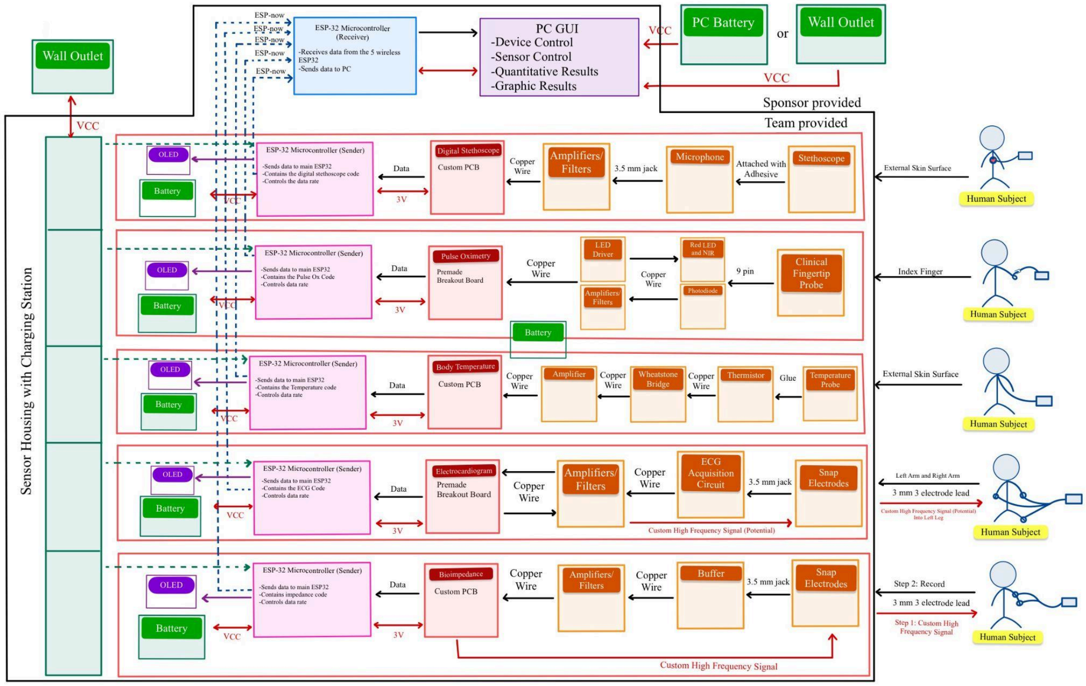

Figure 11. MediBrick 2000 System Block Diagram

4.32 PRIORITY

High priority is given to real-time data display to ensure users have instant access to live sensor readings. It is also a high priority for the data-saving feature to allow users to store and retrieve historical data for analysis. However, some lower priorities are the baud rate and port menu.

4.33 TYPE

- The GUI falls under the category of a monitoring and control application.
- Real-time data display represents a live monitoring type.
- Data-saving functionality falls under a data management and storage type.

4.34 INDIVIDUAL CHARACTERISTICS

Each sensor's data is individually displayed on the GUI, allowing users to monitor specific sensor readings. There are also individual sensor labels that distinguish one sensor from another in the GUI. The GUI also read the port and differentiates the data from each sensor using a symbol or letter that the team will decide later on.

4.35 ASSEMBLY CHARACTERISTICS

The GUI assembles data from multiple ESP32-S3 Feather microcontrollers wirelessly. It then integrates sensor readings in a unified and coherent manner for easy interpretation.

4.36 COMMUNICATION CHARACTERISTICS

Wireless communication between ESP32 devices and the GUI is established using ESP-NOW protocol. Real-time communication ensures quick updates of sensor data on the GUI.

4.37 PROTOCOL CHARACTERISTICS

ESP-NOW is the chosen communication protocol for wireless data transfer. This protocol ensures reliable and low-latency communication between ESP32 microcontrollers. A simple port reading is then performed for the Python program of the GUI to receive data from the receiving ESP32.

5.0 CSCI DETAILED DESIGN

5.1 ELECTROCARDIOGRAPHY (ECG)

ESP32-S3 Feather ECG Arduino Code:

This is the Arduino code for the ECG. The biggest takeaway from this code is the analog read that takes the data from the ECG board into the ESP32.

```
#include <esp_now.h>
#include <WiFi.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include "Adafruit_MAX1704X.h"

const unsigned char Battery_100 [] = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x3F, 0xFF, 0xFF, 0xF0, 0x40, 0x00, 0x00, 0x10, 0x47, 0x3C, 0xE7, 0x18,
0x47, 0xBD, 0xE7, 0x98, 0x45, 0xA5, 0xA5, 0x9E, 0x45, 0xA5, 0xA5, 0x9A, 0x45, 0xA5, 0xA5, 0x9A,
0x45, 0xA5, 0xA5, 0x9A, 0x45, 0xA5, 0xA5, 0x9A, 0x45, 0xA5, 0xA5, 0x9E, 0x47, 0xBD, 0xE7, 0x98,
0x47, 0x3C, 0xE7, 0x18, 0x40, 0x00, 0x00, 0x10, 0x3F, 0xFF, 0xFF, 0xF0, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
const unsigned char ECG_Disp [] = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x80, 0x00, 0x00, 0x00, 0x80, 0x00, 0x00, 0x00, 0x80, 0x00, 0x00, 0x00, 0x80, 0x00,
0x00, 0x00, 0x80, 0x00, 0x00, 0x00, 0x80, 0x00, 0x00, 0x00, 0x80, 0x00, 0x00, 0x00, 0x80, 0x00,
0x00, 0x00, 0xc0, 0x00, 0x00, 0x00, 0xc0, 0x00, 0x00, 0x00, 0xc0, 0x00, 0x00, 0x00, 0xc0, 0x00,
0x00, 0x04, 0xc3, 0x00, 0x01, 0x86, 0xc7, 0x80, 0x03, 0xef, 0xec, 0xff, 0x7e, 0x39, 0x78, 0x00,
0x00, 0x10, 0x60, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x20, 0x00,
0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x20, 0x00,
0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
```

```
static float filteredValue = 0;
// Reset pin not used but needed for library
#define OLED_RESET -1
Adafruit_SSD1306 display(OLED_RESET);

Adafruit_MAX17048 maxlipo;

//debounce of button
// Constants for debounce
const int BUTTON_PIN = 9;
const unsigned long DEBOUNCE_DELAY = 50;
const unsigned long RESET_DURATION = 5000; // 5 seconds
volatile int buttonState = HIGH;
volatile int lastButtonState = HIGH;
volatile unsigned long lastDebounceTime = 0;
volatile unsigned long buttonPressStartTime = 0;
volatile bool resetTriggered = false;
unsigned long lastDisplayUpdateTime = 0;
const unsigned long displayUpdateInterval = 1000;

//define diferent states for oled
typedef enum stateDsiplay {Bat, Data}
stateType;

volatile stateType oledDisplay = Bat;

//interupt func for different states of oled when button is pressed
void handleButtonPress() {
 unsigned long currentMillis = millis();

 // Check if enough time has passed since the last button press
 if (currentMillis - lastDebounceTime > DEBOUNCE_DELAY) {
  int reading = digitalRead(BUTTON_PIN);

  if (reading != lastButtonState) {
   lastDebounceTime = currentMillis;
   lastButtonState = reading;

   if (lastButtonState == HIGH) {
    // Button pressed, start/reset the timer
    buttonPressStartTime = currentMillis;
    resetTriggered = false;
   }
   else {
    // Button released, reset timer and trigger reset if held for 5 seconds
    buttonPressStartTime = 0;
    resetTriggered = false;
   }
```

```
      // Move this part outside the inner if statement
      if (lastButtonState == LOW) {
        // Button pressed, toggle OLED display state
        if (oledDisplay == Bat) {
          oledDisplay = Data;
        }
        else {
          oledDisplay = Bat;
        }
      }
    }
  }
}

// REPLACE WITH YOUR RECEIVER MAC Address
uint8_t broadcastAddress[] = {0xdc, 0x54, 0x75, 0xc3, 0xbe, 0xfc};
// Structure example to send data
// Must match the receiver structure
typedef struct  Ecg {
  int ECG;
} Ecg;

// Create a struct_message called myData
 Ecg ECGData;

esp_now_peer_info_t peerInfo;

// callback when data is sent
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
  Serial.print("\r\nLast Packet Send Status:\t");
  Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Delivery Success" : "Delivery Fail");
}

void updateDisplay() {
  display.clearDisplay();
  // switch case for oled state
  switch (oledDisplay) {
    case Bat:
      display.setTextSize(1.3);
      display.drawBitmap(0, 4, Battery_100, 32, 32, 1);
      display.setCursor(42, 0);
      display.print("ChgR:");
      display.print(maxlipo.chargeRate());
      display.print(" %/hr");
      display.setCursor(40, 15);
      display.print("Bat:");
      display.print(maxlipo.cellPercent(), 1);
      display.print(" %");
```

```
    display.display();
   break;

   //display ecg
   case Data:
    display.setTextSize(1.3);
    display.drawBitmap(0,4,ECG_Disp,32,32,1);
    display.setCursor(35,0);
    display.print("ECG Val:");
    display.print(filteredValue);
    display.display();
   break;
  }
}

void setup() {
  Wire.begin();
  Serial.begin(57600);
  delay(500);

  // Set device as a Wi-Fi Station
  WiFi.mode(WIFI_STA);

  // Init ESP-NOW
  if (esp_now_init() != ESP_OK) {
   Serial.println("Error initializing ESP-NOW");
   return;
  }

  // Once ESPNow is successfully Init, we will register for Send CB to
  // get the status of Trasnmitted packet
  esp_now_register_send_cb(OnDataSent);

  // Register peer
  memcpy(peerInfo.peer_addr, broadcastAddress, 6);
  peerInfo.channel = 0;
  peerInfo.encrypt = false;

  // Add peer
  if (esp_now_add_peer(&peerInfo) != ESP_OK){
   Serial.println("Failed to add peer");
   return;
  }

  pinMode(BUTTON_PIN, INPUT);
  attachInterrupt(digitalPinToInterrupt(BUTTON_PIN), handleButtonPress, CHANGE);
  esp_sleep_enable_ext0_wakeup(GPIO_NUM_9,1); //wake

  pinMode(10, INPUT); // Setup for leads off detection LO +
```

```
 pinMode(11, INPUT); // Setup for leads off detection LO -

 // initialize OLED with I2C addr 0x3C
 display.begin(SSD1306_SWITCHCAPVCC, 0x3C);
 // Clear the display
 display.clearDisplay();
 //Set the color - always use white despite actual display color
 display.setTextColor(WHITE);
 //Set the font size
 display.setTextSize(1.5);
 //Set the cursor coordinates
 display.setCursor(0,0);
 display.print("ECG Config...");
 display.display();
 delay(1000); // pause for a moment

 //check if max module is present if not send error
 if (!maxlipo.begin()) {
  Serial.println(F("Couldnt find Adafruit MAX17048?\nMake sure a battery is plugged in!"));
  while (1) delay(10);
 }
}

void ECG() {
 if((digitalRead(10) == 1)||(digitalRead(11) == 1)) {
  Serial.println('!');
 }
 else {
  int signalValue = analogRead(A1);

  // Apply a low-pass filter
  float alpha = 0.92;  // Filter coefficients (0.9 and 0.1)
  filteredValue = 0;
  filteredValue = alpha * signalValue + (1 - alpha) * filteredValue;
  //myData.ecg = signalValue;
  ECGData.ECG = filteredValue;
   Serial.print("ECG:");
  Serial.println(filteredValue);

  // Send message via ESP-NOW
  esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &ECGData, sizeof(ECGData));

  if (result == ESP_OK) {
   //Serial.println("Sent with success");
  }
  else {
   //Serial.println("Error sending the data");
  }
 }
```

```
}

unsigned long lastECGTime = 0;
const unsigned long ecgInterval = 10;

void loop() {
  unsigned long currentMillis = millis();
  //esp_sleep_enable_ext0_wakeup(GPIO_NUM_9,1);
  // Check if the button is being held down for 5 seconds
  if (buttonPressStartTime != 0 && !resetTriggered && currentMillis - buttonPressStartTime >=
RESET_DURATION) {
    // Button held for 5 seconds, trigger reset
    resetTriggered = true; //deep sleep instead of reset
    display.clearDisplay();
    display.setCursor(0,0);
    display.println("Entering DeepSleep");
    display.println("GOOD NIGHT");
    display.display();
    delay(5000);
    display.clearDisplay();
    display.display();
    esp_deep_sleep_start();
  }
  // Check for ECG update based on time
  if (currentMillis - lastECGTime >= ecgInterval) {
    ECG();
    lastECGTime = currentMillis;
  }

  // Check for display update based on time
  if (currentMillis - lastDisplayUpdateTime >= displayUpdateInterval) {
    updateDisplay();
    lastDisplayUpdateTime = currentMillis;
  }
}
```

<u>ECG MATLAB Code for Analysis:</u>

This is the MATLAB program for the ECG analysis. This program is used to perform fast Fourier transform on the ECG data collected.

```
% Load ECG data from Excel file
[~, ~, data] = xlsread('ecg alone v2.xlsx');

% Assuming the first column is time and the second column is ECG values
time = cell2mat(data(:, 1));
ecg_values = cell2mat(data(:, 2));

% Create smaller figures
```

```
figure('Position', [100, 100, 800, 400]);

% Plot Time Domain
subplot(2, 1, 1);
plot(time, ecg_values);
title('ECG Time Domain');
xlabel('Time (s)');
ylabel('ECG Values');
grid on;

% Plot Frequency Domain (FFT)
subplot(2, 1, 2);
fs = 1 / (time(2) - time(1)); % Calculate sampling frequency
N = length(ecg_values);       % Number of samples

% Ensure N is a power of 2 for FFT efficiency (optional)
N = 2^nextpow2(N);

frequencies = linspace(0, fs/2, N/2 + 1);

% Extend the x-axis range to cover the Nyquist frequency
extendedFrequencies = linspace(0, fs/2, N/2 + 1);

% Compute FFT
fft_values = fft(ecg_values, N);
fft_values = abs(fft_values(1:N/2+1));

% Plot the magnitude spectrum
plot(extendedFrequencies, fft_values);
title('ECG Frequency Domain');
xlabel('Frequency (Hz)');
ylabel('Magnitude');
grid on;

% Identify powerline frequency and harmonics
powerlineFrequency = 60; % Change this to 50 if you are in a 50 Hz powerline region
harmonics = 1:2; % Adjust the range based on the expected harmonics

hold on;

% Plot markers at the powerline frequency and harmonics
for harmonic = harmonics
    line([harmonic * powerlineFrequency, harmonic * powerlineFrequency], [0, max(fft_values)], 'Color', 'r',
'LineStyle', '--');
end

hold off;

% Adjust subplot spacing explicitly
```

```
subplot(2, 1, 1);
ax1 = gca;
ax1.Position = [0.1, 0.55, 0.8, 0.4]; % [left, bottom, width, height]

subplot(2, 1, 2);
ax2 = gca;
ax2.Position = [0.1, 0.1, 0.8, 0.35]; % [left, bottom, width, height]
```

5.2 DIGITAL STETHOSCOPE (SOUND)

ESP32-S3 Feather Sound Sender Arduino Code:

This code is for the sender code of the Sounder Sensor's ESP32-S3 Feather. The sender code for the Sound Sensor module receives the information from the codec ES8388 in I2S format and sends it to the receiver ESP32 using ESPNow. The code initializes the communication of the sender ESP32 with the codec, configures it, filters the incoming signal with a given digital filter, and finally sends the data to the receiver ESP32 using ESPnow.

```
 * @brief We configure the audio codec ES8388 with the help of AudioTools I2SCodecStream.
 * Then, we filtered the stream of bits using a second order IIR with FilteredStream.
 * Finally, we send the stream of bits to a secondary ESP32-S3 using ESP-NOWStream.
 * @author Daniel Campana
 */

#include "AudioTools.h" // install https://github.com/pschatzmann/arduino-audio-tools
#include "AudioLibs/I2SCodecStream.h"
#include "Communication/ESPNowStream.h"

// I2C
#define SDAPIN              3 // I2C Data,  Adafruit ESP32 S3 3, Sparkfun Thing Plus C 23
#define SCLPIN              4 // I2C Clock, Adafruit ESP32 S3 4, Sparkfun Thing Plus C 22
#define I2CSPEED       100000 // Clock Rate
#define ES8388ADDR       0x10 // Address of ES8388 I2C port

// I2S, your configuration for the ES8388 board
#define MCLKPIN            14 // Master Clock
#define BCLKPIN            36 // Bit Clock
#define WSPIN               8 // Word select
#define DOPIN              37 // This is connected to DI on ES8388 (MISO)
#define DIPIN              35 // This is connected to DO on ES8388 (MOSI)

AudioInfo                 audio_info(8000, 1, 16);              // sampling rate, # channels (mono), bit depth

DriverPins                my_pins;                              // board pins
AudioBoard                audio_board(AudioDriverES8388, my_pins); // audio board
I2SCodecStream            i2s_out_stream(audio_board);          // i2s coded
TwoWire                   myWire = TwoWire(0);                  // universal I2C interface
```

```
// Set up filtered stream and copy it to ESP-NOW. The coefficients of the filter may vary (depending on sampling
rate)
uint16_t channels = 1;
FilteredStream<int16_t, float> inFiltered(i2s_out_stream, channels);
// Current specifications of the filter: fc_high = 100, fc_low = 1150, fs = 8000. See MATLAB code for variations of
this filter
const float b_coefficients[] = { 0.1158, 0.0000,-0.2317, 0.0000, 0.1158};
const float a_coefficients[] = { 1.0000,-2.6850, 2.6850,-1.2530, 0.2559};

ESPNowStream now;
StreamCopy copier_now(now, inFiltered);  // copies i2s_out_stream into i2s
const char *peers[] = {"DC:54:75:C3:B8:92"};

// CsvOutput<int16_t> Serial_out(Serial);                    // ASCII output stream
// StreamCopy copier_output(Serial_out, inFiltered);

void setup() {
  // Setup logging
  Serial.begin(115200);
  AudioLogger::instance().begin(Serial, AudioLogger::Warning);
  LOGLEVEL_AUDIODRIVER = AudioDriverWarning;
  delay(2000);

  Serial.println("Setup starting...");

  Serial.println("I2C pin ...");
  my_pins.addI2C(PinFunction::CODEC, SCLPIN, SDAPIN, ES8388ADDR, I2CSPEED, myWire);
  Serial.println("I2S pin ...");
  my_pins.addI2S(PinFunction::CODEC, MCLKPIN, BCLKPIN, WSPIN, DIPIN, DOPIN);
  Serial.println("Pins begin ...");
  my_pins.begin();

  Serial.println("Board begin ...");
  // audio_board.begin();
  CodecConfig cfg;
    //No output to codec
    cfg.output_device = DAC_OUTPUT_LINE1;
    //Mic input from adc channel 1
    cfg.input_device = ADC_INPUT_LINE1;
    //Bits per sample (16 bits)
    cfg.i2s.bits = BIT_LENGTH_16BITS;
    //Sample Rate (44.1 kHz)
    cfg.i2s.rate = RATE_8K;     //cfg.i2s.rate = RATE_44K;
    //Channels 2
    //cfg.i2s.channels = CHANNELS2;
    //Format
    //cfg.i2s.fmt = I2S_NORMAL;
    // codec is slave - microcontroller is master
    //cfg.i2s.mode = MODE_SLAVE;
```

```
  audio_board.begin(cfg);
 //Additional ADC configuration ---> Everything else is in default mode (See datasheet)
     //set to mono mix in ADC channel 1 (lin1/Mic pin) --> 00001000 == 0x08 in ADCControl 3
 es8388_write_reg(ES8388_ADCCONTROL3, 0x08);
     //12 dB gain in ADC channel 1 (lin1/Mic pin) --> 01000100 == 0x44 in ADCControl 1
// es8388_write_reg(ES8388_ADCCONTROL1, 0x44);

 Serial.println("I2S begin ...");
 auto i2s_config = i2s_out_stream.defaultConfig(RXTX_MODE); //RXTX for douplex //RX for sink //TX for
source
 i2s_config.copyFrom(audio_info);
 i2s_out_stream.begin(i2s_config); // this should apply I2C and I2S configuration

 // // Setup CSV
 //   Serial.println("CSV begin...");
 //   Serial_out.begin(audio_info);

 // Setup filter
 inFiltered.setFilter(0, new IIR<float>(b_coefficients, a_coefficients));

 // Setup ESP_NOW
 auto now_config = now.defaultConfig();
 now_config.mac_address = "DC:54:75:C3:B8:0C";
 now.begin(now_config);
 now.addPeers(peers);

 Serial.println("Setup completed ...");
 delay(5000);
}

// Arduino loop - copy sound to out
void loop() {
 //copier_output.copy();
 copier_now.copy();
}
```

ESP32-S3 Feather Sound Receiver Arduino Code:

The receiver code for the Sound Sensor module receives the information from the ESP32 sender (through ESPNow) and displays it to the serial monitor.

```
/**
 * @file example-serial-receive.ino
 * @author Phil Schatzmann
 * @brief Receiving audio via ESPNow
 * @version 0.1
 * @date 2022-03-09
 *
 * @copyright Copyright (c) 2022
```

```
 */

#include "AudioTools.h"
#include "Communication/ESPNowStream.h"

AudioInfo info(32000, 1, 16);
ESPNowStream now;

CsvOutput<int16_t> Serial_out(Serial);                    // ASCII output stream
StreamCopy copier_output(Serial_out, now);
const char *peers[] = {"DC:54:75:C3:B8:0C"};

void setup() {
  Serial.begin(115200);
  AudioLogger::instance().begin(Serial, AudioLogger::Info);

  auto now_cfg = now.defaultConfig();
  now_cfg.mac_address = "DC:54:75:C0:92:28";
  now.begin(now_cfg);
  now.addPeers(peers);

  // Setup CSV
  Serial.println("CSV begin...");
  Serial_out.begin(info);

  Serial.println("Receiver started...");
}

void loop() {
  copier_output.copy();
}
```

MATLAB Sound Sensor Filter Code:

The filter code in MATLAB is a tool that allows the user to create IIR LPF, IIR HPF, and IIR BPF of Butterworth type with different orders of magnitude. It requires the user to include a cutoff frequency for the LPF, a cutoff frequency for the HPF, a sampling frequency, and the order of the filters. The code will return the coefficients of the difference equation that will be included in the sender Arduino Code for the Sound Sensor module.

```
fc_l = 1150;    % Cutoff frequency for LPF
fc_h = 100;     % Cutoff frequency for HPF
fs = 8000;      % Sampling Frequency

% 2nd order LPF
[b_l,a_l] = butter(2,fc_l/(fs/2));
%freqz(b_l,a_l,[],fs);
%subplot(2,1,1);
```

```
% 2nd order HPF
[b_h,a_h] = butter(2,fc_h/(fs/2), 'high');
%freqz(b_h,a_h,[],fs);
%subplot(2,1,1);

% Multiply the transfer functions to create a bandpass filter
   % Create transfer function objects for LPF and HPF
   tf_lpf = tf(b_l, a_l, 1/fs) % LPF transfer function
   tf_hpf = tf(b_h, a_h, 1/fs) % HPF transfer function
   % Multiply the transfer functions to create a bandpass filter
   tf_bpf = tf_lpf * tf_hpf
   % Get the numerator and denominator coefficients of the bandpass filter
   [b_bpf, a_bpf] = tfdata(tf_bpf, 'v');

freqz(b_bpf,a_bpf,[],fs);
subplot(2,1,1);

% DC Gain
dc_gain = sum(b_bpf) / sum(a_bpf);
```

5.3 TEMPERATURE

<u>ESP32-S3 Feather Temperature Arduino Sender Code:</u>

This code is the Arduino code for the Temperature sensor. The biggest aspect of this code is the Steinhart-Hart equation that converts resistance differences into Kelvins.

```
#include <Arduino.h>
#include <math.h>
#include <esp_now.h>
#include <WiFi.h>
// Include Wire Library for I2C
#include <Wire.h>
// Include Adafruit Graphics & OLED libraries
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include "Adafruit_MAX1704X.h"
////////////////////////////////

int ThermVb = A0;
int ThermVd = A1;
float Vb;
float Vd;
float Vin = 3290;
float Vdiff;
float R1 = 9820;
float R2 = 9897;
float R3 = 9860;
```

```
float R4;
float Temps;
float A = 0.001111285538;
float B = 0.0002371215953;
float C = 0.00000007520676806;

int i = 0;
float Temptot = 0;
float Tempave = 0;

////////////////////////////////

uint8_t broadcastAddress[] = {0xDC, 0x54, 0x75, 0xC3, 0xBE, 0xFC};

typedef struct Temp{
  double Temp;
} Temp;
 Temp TempData;

esp_now_peer_info_t peerInfo;

//calibration input from master
typedef struct TempR{
   unsigned int R1;
   unsigned int R2;
   unsigned int R3;
} TempR;

TempR TempCali;
//icons for oled
const unsigned char Battery_100 [] = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x3F, 0xFF, 0xFF, 0xF0, 0x40, 0x00, 0x00, 0x10, 0x47, 0x3C, 0xE7, 0x18,
0x47, 0xBD, 0xE7, 0x98, 0x45, 0xA5, 0xA5, 0x9E, 0x45, 0xA5, 0xA5, 0x9A, 0x45, 0xA5, 0xA5, 0x9A,
0x45, 0xA5, 0xA5, 0x9A, 0x45, 0xA5, 0xA5, 0x9A, 0x45, 0xA5, 0xA5, 0x9E, 0x47, 0xBD, 0xE7, 0x98,
0x47, 0x3C, 0xE7, 0x18, 0x40, 0x00, 0x00, 0x10, 0x3F, 0xFF, 0xFF, 0xF0, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

const unsigned char temprature [] = {
0x00, 0x00, 0x00, 0x00, 0x60, 0x00, 0x01, 0xF8, 0x00, 0x01, 0x08, 0x00, 0x03, 0x0C, 0x00, 0x03,
0x6D, 0xC0, 0x03, 0x6C, 0x00, 0x03, 0x6D, 0xE0, 0x03, 0x69, 0xE0, 0x03, 0x60, 0x00, 0x03, 0x61,
0xC0, 0x03, 0x60, 0x00, 0x03, 0x6D, 0xE0, 0x03, 0x6D, 0xE0, 0x03, 0x6C, 0x00, 0x06, 0xF6, 0x00,
0x05, 0x9A, 0x00, 0x05, 0x9A, 0x00, 0x05, 0x92, 0x00, 0x06, 0xF6, 0x00, 0x03, 0x0C, 0x00, 0x01,
0xF8, 0x00, 0x00, 0x60, 0x00, 0x00, 0x00, 0x00,
};
```

```
// Reset pin not used but needed for library
#define OLED_RESET -1
Adafruit_SSD1306 display(OLED_RESET);


Adafruit_MAX17048 maxlipo;


//debounce of button
// Constants for debounce
const int BUTTON_PIN = 10;
const unsigned long DEBOUNCE_DELAY = 50;
const unsigned long RESET_DURATION = 5000; // 5 seconds
volatile int buttonState = HIGH;
volatile int lastButtonState = HIGH;
volatile unsigned long lastDebounceTime = 0;
volatile unsigned long buttonPressStartTime = 0;
volatile bool resetTriggered = false;
unsigned long lastDisplayUpdateTime = 0;
const unsigned long displayUpdateInterval = 1000; //# of iterations before oled update
unsigned long  lastTempCalcTime = 0 ;
const unsigned long TEMP_CALC_INTERVAL = 15;

//define diferent states for oled
typedef enum stateDsiplay {Bat, Data}
stateType;

volatile stateType oledDisplay = Bat;

//interupt func for different states of oled when button is pressed
void handleButtonPress() {
  unsigned long currentMillis = millis();

  // Check if enough time has passed since the last button press
  if (currentMillis - lastDebounceTime > DEBOUNCE_DELAY) {
    int reading = digitalRead(BUTTON_PIN);

    if (reading != lastButtonState) {
      lastDebounceTime = currentMillis;
      lastButtonState = reading;

      if (lastButtonState == HIGH) {
        // Button pressed, start/reset the timer
        buttonPressStartTime = currentMillis;
        resetTriggered = false;
      } else {
        // Button released, reset timer and trigger reset if held for 5 seconds
        buttonPressStartTime = 0;
```

```
        resetTriggered = false;
      }

      // Move this part outside the inner if statement
      if (lastButtonState == HIGH) {
        // Button pressed, toggle OLED display state
        if (oledDisplay == Bat) {
          oledDisplay = Data;
        } else {
          oledDisplay = Bat;
        }
      }
    }
  }
}

float calctemp(float R4) {
  Temps = 1/(A + B * log(R4) + C * pow(log(R4), 3));
  Temps = (Temps-273.15);
  return Temps;
}

void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
  Serial.print("\r\nLast Packet Send Status:\t");
  Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Delivery Success" : "Delivery Fail");
}

// Callback function executed when data is received
void OnDataRecv(const uint8_t * mac, const uint8_t *incomingData, int len) {
  memcpy(&TempCali, incomingData, sizeof(TempCali));
 R1 = TempCali.R1;
 R2 = TempCali.R2;
 R3 = TempCali.R3;
}
////////////////////////////////
void updateDisplay() {
 display.clearDisplay();
   // switch case for oled state
    switch (oledDisplay)
  {
   //display battery precent and charge discharge rate
  case Bat:

    display.setTextSize(1.3);
  display.drawBitmap(0,4,Battery_100,32,32,1);
  display.setCursor(42,0);
  display.print("ChgR:");display.print(maxlipo.chargeRate()); display.print(" %/hr");
  display.setCursor(40,15);
  display.print("Bat:");display.print(maxlipo.cellPercent(), 1); display.print(" %");
```

```
    break;
  //display temp
  case Data:
  display.setTextSize(1.3);
  display.drawBitmap(0,4,temprature,24,24,1);
  display.setCursor(42,0);
  display.print("Temp:");display.print(Tempave);display.print(" *C");
  display.setCursor(42,7);
  display.setTextSize(0.5);
  display.print("R1:");display.print(R1);
  display.setCursor(42,16);
  display.print("R2:");display.print(R2);
 display.setCursor(42,26);
  display.print("R3:");display.print(R3);
    break;
  }
  display.display();

}
//////////////////////////////////

void setup() {
  Serial.begin(9600);
  delay(500);

//////////////////////////////////

  // Set device as a Wi-Fi Station
  WiFi.mode(WIFI_STA);

  // Init ESP-NOW
  if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
  }

  // Once ESPNow is successfully Init, we will register for Send CB to
  // get the status of Trasnmitted packet
  esp_now_register_send_cb(OnDataSent);
  // Register callback function
  esp_now_register_recv_cb(OnDataRecv);
  // Register peer
  memcpy(peerInfo.peer_addr, broadcastAddress, 6);
  peerInfo.channel = 0;
  peerInfo.encrypt = false;

  // Add peer
  if (esp_now_add_peer(&peerInfo) != ESP_OK){
```

```
    Serial.println("Failed to add peer");
    return;
  }

////////////////////////////////

//INIT  oled button
 pinMode(10,INPUT);
attachInterrupt(digitalPinToInterrupt(BUTTON_PIN), handleButtonPress, CHANGE);
esp_sleep_enable_ext0_wakeup(GPIO_NUM_10,1); //wake up pin if module is in deep sleep
  // Start Wire library for I2C
  Wire.begin();
  // initialize OLED with I2C addr 0x3C
  display.begin(SSD1306_SWITCHCAPVCC, 0x3C);
    // Clear the display
  display.clearDisplay();
  //Set the color - always use white despite actual display color
  display.setTextColor(WHITE);
  //Set the font size
  display.setTextSize(1.5);
  //Set the cursor coordinates
  display.setCursor(0,0);
  display.print("Temp Sensor Config...");
  display.display();
 delay(1000);
  //check if max module is present if not send error
  if (!maxlipo.begin()) {
    Serial.println(F("Couldnt find Adafruit MAX17048?\nMake sure a battery is plugged in!"));
    while (1) delay(10);
  }
}

void loop() {


//reset button
  unsigned long currentMillis = millis();

  // Check if the button is being held down for 5 seconds
  if (buttonPressStartTime != 0 && !resetTriggered && currentMillis - buttonPressStartTime >=
RESET_DURATION) {
    // Button held for 5 seconds, trigger reset
    resetTriggered = true;

    display.clearDisplay();
    display.setCursor(0,0);
    display.println("Entering DeepSleep");
    display.println("GOOD NIGHT");
    display.display();
```

```
    delay(5000);
    display.clearDisplay();
    display.display();
    esp_deep_sleep_start();
  }

// Check if 10 milliseconds have passed since the last temperature calculation
  if (currentMillis - lastTempCalcTime >= TEMP_CALC_INTERVAL) {
    // Perform temperature calculation
    Vb = analogReadMilliVolts(ThermVb);
    Vd = analogReadMilliVolts(ThermVd);

    Vdiff = Vb - Vd;
    R4 = R3 / (R2 / (R1 + R2) - Vdiff / Vin) - R3;

    Temps = calctemp(R4);

    if (i == 100) {
      Tempave = Temptot / i;
      Tempave = Tempave * 100;
      Tempave = round(Tempave);
      Tempave = Tempave / 100;
      i = 0;
      Temptot = 0;
      Serial.println(Tempave);

      TempData.Temp = Tempave;
    }

    i = i + 1;
    Temptot = Temptot + Temps;

    // Update the last temperature calculation time
    lastTempCalcTime = currentMillis;
  }

    // Check for display update based on time
  if (currentMillis - lastDisplayUpdateTime >= displayUpdateInterval) {
    updateDisplay();
    lastDisplayUpdateTime = currentMillis;
  }
      esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &TempData, sizeof(TempData));
    if (result == ESP_OK) {
      Serial.println("Sent with success");
  }
    else {
      Serial.println("Error sending the data");
  }
```

}

5.4 IMPEDANCE

ESP32-S3 Feather Impedance Arduino Code:

This code is the Arduino code for the impedance sensor. It communicates using the I2C protocol.

```
#include "Wire.h"
#include <math.h>
#include <esp_now.h>
#include <WiFi.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include "Adafruit_MAX1704X.h"

// Configure the registers (refer to the AD5933 Datasheet)
#define SLAVE_ADDR 0x0D
#define ADDR_PTR 0xB0

#define START_FREQ_R1 0x82
#define START_FREQ_R2 0x83
#define START_FREQ_R3 0x84

#define FREG_INCRE_R1 0x85
#define FREG_INCRE_R2 0x86
#define FREG_INCRE_R3 0x87

#define NUM_INCRE_R1 0x88
#define NUM_INCRE_R2 0x89

#define NUM_SCYCLES_R1 0x8A
#define NUM_SCYCLES_R2 0x8B

#define RE_DATA_R1 0x94
#define RE_DATA_R2 0x95

#define IMG_DATA_R1 0x96
#define IMG_DATA_R2 0x97

#define TEMP_R1 0x92
#define TEMP_R2 0x93

#define CTRL_REG 0x80
#define CTRL_REG2 0x81

#define STATUS_REG 0x8F
```

```
const float MCLK = 16.776*pow(10,6); // AD5933 Internal Clock Speed 16.776 MHz
const float start_freq = 50*pow(10,3); // Set start freq, < 100Khz
const float incre_freq = 1*pow(10,2); // Set freq increment
const int incre_num = 49; // Set number of increments; < 511

// Initialized the receive data from the receiver
bool cal = 0;
bool measure = 1;

char state;
double gainFactor = 0.0;

// Receiver MAC address
uint8_t broadcastAddress[] = {0xDC, 0x54, 0x75, 0xC3, 0xBE, 0xFC};

typedef struct Imp{ //Send to receiver
  double Impavg;
  double Imps;
} Imp;
Imp ImpData;

esp_now_peer_info_t peerInfo;

//calibration input from master
typedef struct ImpFlag{
   bool cal;
   bool measure;
} ImpFlag;
ImpFlag Flag;

//icons for oled
const unsigned char Battery_100 [] = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x3F, 0xFF, 0xFF, 0xF0, 0x40, 0x00, 0x00, 0x10, 0x47, 0x3C, 0xE7, 0x18,
0x47, 0xBD, 0xE7, 0x98, 0x45, 0xA5, 0xA5, 0x9E, 0x45, 0xA5, 0xA5, 0x9A, 0x45, 0xA5, 0xA5, 0x9A,
0x45, 0xA5, 0xA5, 0x9A, 0x45, 0xA5, 0xA5, 0x9A, 0x45, 0xA5, 0xA5, 0x9E, 0x47, 0xBD, 0xE7, 0x98,
0x47, 0x3C, 0xE7, 0x18, 0x40, 0x00, 0x00, 0x10, 0x3F, 0xFF, 0xFF, 0xF0, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

const unsigned char Imp_screen [] = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x07, 0xe0, 0x00, 0x00, 0x1f, 0xf8, 0x00, 0x00, 0x3c, 0x3c, 0x00, 0x00, 0x73, 0xce, 0x00,
0x00, 0x67, 0xe6, 0x00, 0x1e, 0xee, 0x77, 0x78, 0x3f, 0xcc, 0x33, 0xfc, 0x33, 0xcc, 0x33, 0xcc,
0x33, 0xce, 0x73, 0xcc, 0x3f, 0xde, 0x7b, 0xfc, 0x1e, 0xfe, 0x7f, 0x78, 0x00, 0x60, 0x06, 0x00,
0x00, 0x70, 0x0e, 0x00, 0x00, 0x3c, 0x3c, 0x00, 0x00, 0x1f, 0xf8, 0x00, 0x00, 0x07, 0xe0, 0x00,
```

```
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

// Reset pin not used but needed for library
#define OLED_RESET -1
Adafruit_SSD1306 display(OLED_RESET);

Adafruit_MAX17048 maxlipo;

//debounce of button
//Constants for debounce
const int BUTTON_PIN = 9;
const unsigned long DEBOUNCE_DELAY = 50;
const unsigned long RESET_DURATION = 5000; // 5 seconds
volatile int buttonState = HIGH;
volatile int lastButtonState = HIGH;
volatile unsigned long lastDebounceTime = 0;
volatile unsigned long buttonPressStartTime = 0;
volatile bool resetTriggered = false;
unsigned long lastDisplayUpdateTime = 0;
const unsigned long displayUpdateInterval = 1000; //# of iterations before oled update

//define diferent states for oled
typedef enum stateDsiplay {Bat, Data}
stateType;

volatile stateType oledDisplay = Bat;

//interupt func for different states of oled when button is pressed
void handleButtonPress() {
 unsigned long currentMillis = millis();

 // Check if enough time has passed since the last button press
 if (currentMillis - lastDebounceTime > DEBOUNCE_DELAY) {
  int reading = digitalRead(BUTTON_PIN);

  if (reading != lastButtonState) {
   lastDebounceTime = currentMillis;
   lastButtonState = reading;

   if (lastButtonState == HIGH) {
    // Button pressed, start/reset the timer
    buttonPressStartTime = currentMillis;
    resetTriggered = false;
   }
   else {
    // Button released, reset timer and trigger reset if held for 5 seconds
    buttonPressStartTime = 0;
```

```
        resetTriggered = false;
      }

      // Move this part outside the inner if statement
      if (lastButtonState == LOW) {
        // Button pressed, toggle OLED display state
        if (oledDisplay == Bat) {
          oledDisplay = Data;
        }
        else {
          oledDisplay = Bat;
        }
      }
    }
  }
}

void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
  Serial.print("\r\nLast Packet Send Status:\t");
  Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Delivery Success" : "Delivery Fail");
}

// Callback function executed when data is received
void OnDataRecv(const uint8_t * mac, const uint8_t *incomingData, int len) {
  memcpy(&Flag, incomingData, sizeof(Flag));
  cal = Flag.cal;
  measure = Flag.measure;
}

/////////////////////////////////
void updateDisplay() {
  display.clearDisplay();

  // switch case for oled state
  switch (oledDisplay) {

  //display battery precent and charge discharge rate
  case Bat:
    display.setTextSize(1.3);
    display.drawBitmap(0,4, Battery_100 ,32,32,1);
    display.setCursor(42,0);
    display.print("ChgR:");
    display.print(maxlipo.chargeRate());
    display.print(" %/hr");
    display.setCursor(40,15);
    display.print("Bat:");
    display.print(maxlipo.cellPercent(), 1);
    display.print(" %");
    display.display();
```

```
  break;

  //display imp
  case Data:
    display.setTextSize(1.3);
    display.drawBitmap(0,4, Imp_screen ,32,32,1);
    display.setCursor(42,0);
    display.print("Impedance:");
    display.setCursor(40,15);
    display.print(ImpData.Impavg);
    display.print(" ohm");
    display.display();
  break;
  }
}
/////////////////////////////
void setup() {
        Wire.begin();
        Serial.begin(115200);
  delay(500);
  pinMode(12, OUTPUT);
  ////////////////////////////////

  // Set device as a Wi-Fi Station
  WiFi.mode(WIFI_STA);

  // Init ESP-NOW
  if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
  }

  // Once ESPNow is successfully Init, we will register for Send CB to

  // Register callback function
  esp_now_register_recv_cb(OnDataRecv);
  // Register peer
  memcpy(peerInfo.peer_addr, broadcastAddress, 6);
  peerInfo.channel = 0;
  peerInfo.encrypt = false;

  // Add peer
  if (esp_now_add_peer(&peerInfo) != ESP_OK){
    Serial.println("Failed to add peer");
    return;
  }


  //INIT  oled button
```

```
  pinMode(BUTTON_PIN, INPUT);
  attachInterrupt(digitalPinToInterrupt(BUTTON_PIN), handleButtonPress, CHANGE);
  esp_sleep_enable_ext0_wakeup(GPIO_NUM_9,1); //wake
  // Start Wire library for I2C
  //Wire.begin();
  // initialize OLED with I2C addr 0x3C
  display.begin(SSD1306_SWITCHCAPVCC, 0x3C);
    // Clear the display
  display.clearDisplay();
  //Set the color - always use white despite actual display color
  display.setTextColor(WHITE);
  //Set the font size
  display.setTextSize(1.5);
  //Set the cursor coordinates
  display.setCursor(0,0);
  display.print("Imp Sensor Config...");
  display.display();
  delay(1000);
  //check if max module is present if not send error
  if (!maxlipo.begin()) {
    Serial.println(F("Couldnt find Adafruit MAX17048?\nMake sure a battery is plugged in!"));
    while (1) delay(10);
  }

		//nop - clear ctrl-reg
		writeData(CTRL_REG,0x0);

		//reset ctrl register
		writeData(CTRL_REG2,0x10);

		programReg();
  calibrateGainFactor();
}

void loop(){
  //reset button
  unsigned long currentMillis = millis();

  // Check if the button is being held down for 5 seconds
  if (buttonPressStartTime != 0 && !resetTriggered && currentMillis - buttonPressStartTime >=
RESET_DURATION) {
    // Button held for 5 seconds, trigger reset
    resetTriggered = true;
    display.clearDisplay();
    display.setCursor(0,0);
    display.println("Entering DeepSleep");
    display.println("GOOD NIGHT");
    display.display();
    delay(5000);
```

```
    display.clearDisplay();
    display.display();
    esp_deep_sleep_start();
  }
  if (cal == 1) {
    ESP.restart();
    cal = 0;
  }
  if (measure == 1) {
    runSweep();
    delay(1000);
    measure = 0;
  }
  Serial.flush();

  if (currentMillis - lastDisplayUpdateTime >= displayUpdateInterval) {
    updateDisplay();
    lastDisplayUpdateTime = currentMillis;
  }
}


void calibrateGainFactor() {
	short re;
	short img;
	double freq;
	float mag;
  float magtot = 0;
  float magavg;
  int i=0;

	programReg();
  digitalWrite(12, HIGH);
  delay(100);

	// 1. Standby '10110000' Mask D8-10 of avoid tampering with gains
	writeData(CTRL_REG,(readData(CTRL_REG) & 0x07) | 0xB0);

	// 2. Initialize sweep
	writeData(CTRL_REG,(readData(CTRL_REG) & 0x07) | 0x10);

	// 3. Start sweep
	writeData(CTRL_REG,(readData(CTRL_REG) & 0x07) | 0x20);


	while((readData(STATUS_REG) & 0x07) < 4 ) {  // Check that status reg != 4, sweep not complete
		delay(100); // delay between measurements

		int flag = readData(STATUS_REG)& 2;
```

```
                if (flag==2) {

                        byte R1 = readData(RE_DATA_R1);
                        byte R2 = readData(RE_DATA_R2);
                        re = (R1 << 8) | R2;

                        R1  = readData(IMG_DATA_R1);
                        R2  = readData(IMG_DATA_R2);
                        img = (R1 << 8) | R2;


                        freq = start_freq + i*incre_freq;
                        mag = sqrt(pow(double(re),2)+pow(double(img),2));
   magtot = magtot + mag;
   Serial.print("mag:");
   Serial.println(mag);
                        if((readData(STATUS_REG) & 0x07) < 4 ){
                                writeData(CTRL_REG,(readData(CTRL_REG) & 0x07) | 0x30);
                                i++;
                        }
  }
        }
 int count = 50;
 magavg = magtot / count;
 gainFactor = (1.0 /2000.0) / magavg;
 Serial.print("Mag Avg: ");
 Serial.println(magavg);
 Serial.print("GF: ");
 Serial.println(gainFactor, 16);
 digitalWrite(12, LOW);
 delay(100);
        writeData(CTRL_REG,(readData(CTRL_REG) & 0x07) | 0xA0);
}

void programReg(){

        // Set Range 1, PGA gain 1
        writeData(CTRL_REG,0x01);

        // Set settling cycles
        writeData(NUM_SCYCLES_R1, 0x07);
        writeData(NUM_SCYCLES_R2, 0xFF);

        // Start frequency of 1kHz
        writeData(START_FREQ_R1, getFrequency(start_freq,1));
        writeData(START_FREQ_R2, getFrequency(start_freq,2));
        writeData(START_FREQ_R3, getFrequency(start_freq,3));
```

```
        // Increment by 1 kHz
        writeData(FREG_INCRE_R1, getFrequency(incre_freq,1));
        writeData(FREG_INCRE_R2, getFrequency(incre_freq,2));
        writeData(FREG_INCRE_R3, getFrequency(incre_freq,3));

        // Points in frequency sweep (100), max 511
        writeData(NUM_INCRE_R1, (incre_num & 0x001F00)>>0x08 );
        writeData(NUM_INCRE_R2, (incre_num & 0x0000FF));
}

void runSweep() {
        short re;
        short img;
        double freq;
        double mag;
        double phase;
        double gain;
        double Impedance;
  double GF;
  double FFW;
  double wt;
  double ht;
  double BF;
  double tot = 0;
  double magcount = 0;
  double impcount = 0;
  double avgmag;
  double totimp = 0;
  double avgimp;
  double totreac = 0;
  double avgreac;
  double totre;
  double avgre;
        int i=0;
  int A;
  int G;

        programReg();
  delay(100);

        // 1. Standby '10110000' Mask D8-10 of avoid tampering with gains
        writeData(CTRL_REG,(readData(CTRL_REG) & 0x07) | 0xB0);

        // 2. Initialize sweep
        writeData(CTRL_REG,(readData(CTRL_REG) & 0x07) | 0x10);

        // 3. Start sweep
        writeData(CTRL_REG,(readData(CTRL_REG) & 0x07) | 0x20);
```

```
        while((readData(STATUS_REG) & 0x07) < 4 ) {  // Check that status reg != 4, sweep not complete
                delay(100); // delay between measurements

                int flag = readData(STATUS_REG)& 2;


                if (flag==2) {

                        byte R1 = readData(RE_DATA_R1);
                        byte R2 = readData(RE_DATA_R2);
                        re = (R1 << 8) | R2;
    totre = totre + re;

                        R1  = readData(IMG_DATA_R1);
                        R2  = readData(IMG_DATA_R2);
                        img = (R1 << 8) | R2;
    totreac = totreac + img;

    // debug prints
    Serial.print("real:"); Serial.println(re);
    Serial.print("img:"); Serial.println(img);
    phase = atan(double(img)/double(re));
                        phase = (180.0/3.1415926)*phase;  //convert phase angle to degrees

                        freq = start_freq + i*incre_freq;
                        mag = sqrt(pow(double(re),2)+pow(double(img),2));

    // debug print
    Serial.print("phase:"); Serial.println(phase);
    Serial.print("mag:"); Serial.println(mag);

    tot = tot+mag;
    Impedance = (1/(gainFactor*mag));
    ImpData.Imps = Impedance;
    esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &ImpData, sizeof(ImpData));
    if (result == ESP_OK) {
      Serial.println("Sent with success");
    }
    else {
      Serial.println("Error sending the data");
    }

    // Update impedance count and total impedance for all impedance values
    //impcount = 50;
    totimp = totimp + Impedance;

                        Serial.print("Frequency: ");
                        Serial.print(freq/1000);
```

```
                    Serial.print(",kHz;");

    Serial.print(" Impedance Magnitude: ");
                    Serial.print(Impedance);
                    Serial.println(";");

                    // break;  //TODO: for single run, remove after debugging

                    //Increment frequency
                    if((readData(STATUS_REG) & 0x07) < 4 ){
                        writeData(CTRL_REG,(readData(CTRL_REG) & 0x07) | 0x30);
                        i++;
                    }
                }
            }

  int count = 50;
  avgmag = tot/count;
  avgimp = totimp/count;
  avgreac = totreac/count;
  avgre = totre/count;

  // Uncomment below if want to measure body fat from the impedance sensor. Body fat calculations are made in the
GUI.
  // wt = 82; //lbsC
  // ht = 167; //cm, m

  //FFW = (0.396*((pow(ht,2))/(avgimp/200))+ (0.143 * wt) + 8.399)*1.37*2; //instruc
  // Serial.print ("FFM: ");
  // Serial.println (FFW);
  // BF = ((wt-FFW)/wt)*100;


  Serial.print(" Avg Mag: ");
  Serial.print(avgmag);
  Serial.print(",");

  Serial.print(" Avg Impedance: ");
  Serial.print(avgimp);
  Serial.print(",");
  ImpData.Impavg = avgimp;
  esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &ImpData, sizeof(ImpData));
  if (result == ESP_OK) {
    Serial.println("Sent with success");
  }
   else {
    Serial.println("Error sending the data");
  }
  // Serial.print(" % Body Fat: ");
```

```
  // Serial.print(BF);
  // Serial.print(",");

        //Power down
  writeData(CTRL_REG,0xA0);
  //digitalWrite(12, LOW);
  delay(100);
        writeData(CTRL_REG,(readData(CTRL_REG) & 0x07) | 0xA0);
}

void writeData(int addr, int data) {

 Wire.beginTransmission(SLAVE_ADDR);
 Wire.write(addr);
 Wire.write(data);
 Wire.endTransmission();
 delay(1);
}

int readData(int addr){
        int data;

        Wire.beginTransmission(SLAVE_ADDR);
        Wire.write(ADDR_PTR);
        Wire.write(addr);
        Wire.endTransmission();

        delay(1);

        Wire.requestFrom(SLAVE_ADDR,1);

        if (Wire.available() >= 1){
                data = Wire.read();
        }
        else {
                data = -1;
        }

        delay(1);
        return data;
}

byte getFrequency(float freq, int n){
        long val = long((freq/(MCLK/4)) * pow(2,27));
        byte code;

          switch (n) {
            case 1:
              code = (val & 0xFF0000) >> 0x10;
```

```
        break;

      case 2:
        code = (val & 0x00FF00) >> 0x08;
        break;

      case 3:
        code = (val & 0x0000FF);
        break;

      default:
        code = 0;
    }

    return code;
}
```

5.5 PULSE OXIMETER

ESP32-S3 Feather Pulse Ox Arduino Sender Code:

This code is the Arduino code for the Pulse ox sensor. It takes advantage of the library Protocentral_afe44xx in order to calculate the SpO2 and Heart Rate of the users.

```
#include <Arduino.h>
#include <SPI.h>
#include <esp_now.h>
#include <WiFi.h>
#include "protocentral_afe44xx.h"
// Include Wire Library for I2C
#include <Wire.h>
// Include Adafruit Graphics & OLED libraries
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include "Adafruit_MAX1704X.h"
// MAC Address of responder - edit as required
uint8_t broadcastAddress[] = {0xDC, 0x54, 0x75, 0xC3, 0xBE, 0xFC};
// Must match the receiver structure
typedef struct OXI {
  uint16_t IR;  // Use uint16_t for IR to save space
  uint16_t RED; // Use uint16_t for RED to save space
  uint8_t BPM;
  uint8_t SPO2;
} OXI;

// Create a structured object
OXI MediData;
esp_now_peer_info_t peerInfo;
//icons for oled
```

```
const unsigned char Battery_100 [] = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x3F, 0xFF, 0xFF, 0xF0, 0x40, 0x00, 0x00, 0x10, 0x47, 0x3C, 0xE7, 0x18,
0x47, 0xBD, 0xE7, 0x98, 0x45, 0xA5, 0xA5, 0x9E, 0x45, 0xA5, 0xA5, 0x9A, 0x45, 0xA5, 0xA5, 0x9A,
0x45, 0xA5, 0xA5, 0x9A, 0x45, 0xA5, 0xA5, 0x9A, 0x45, 0xA5, 0xA5, 0x9E, 0x47, 0xBD, 0xE7, 0x98,
0x47, 0x3C, 0xE7, 0x18, 0x40, 0x00, 0x00, 0x10, 0x3F, 0xFF, 0xFF, 0xF0, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
const unsigned char Heart_Pulse [] = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x07, 0xF8, 0x1F, 0xE0,
0x1F, 0xFC, 0x3F, 0xF8, 0x3F, 0xFE, 0x7F, 0xFC, 0x3F, 0xFF, 0xFF, 0xFC, 0x7F, 0xFF, 0xFF, 0xFE,
0x7F, 0xFF, 0xFF, 0xFE, 0xFF, 0xC7, 0xFF, 0xFF, 0xFF, 0xC7, 0xFF, 0xFF, 0xFF, 0x83, 0xE7, 0xFF,
0xFF, 0x83, 0xC3, 0xFF, 0xFF, 0x83, 0xC3, 0xFF, 0xFF, 0x01, 0x81, 0xFF, 0x00, 0x11, 0x80, 0x00,
0x00, 0x18, 0x00, 0x00, 0x00, 0x78, 0x18, 0x00, 0x1F, 0xF8, 0x1F, 0xF8, 0x0F, 0xFC, 0x3F, 0xF0,
0x07, 0xFC, 0x3F, 0xE0, 0x03, 0xFE, 0x7F, 0xC0, 0x01, 0xFF, 0xFF, 0x80, 0x00, 0xFF, 0xFF, 0x00,
0x00, 0x7F, 0xFE, 0x00, 0x00, 0x3F, 0xFC, 0x00, 0x00, 0x1F, 0xF8, 0x00, 0x00, 0x0F, 0xF0, 0x00,
0x00, 0x07, 0xE0, 0x00, 0x00, 0x01, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};


#define AFE44XX_CS_PIN   6
#define AFE44XX_DRDY_PIN 13
#define AFE44XX_PWDN_PIN 5

// Reset pin not used but needed for library
#define OLED_RESET -1
Adafruit_SSD1306 display(OLED_RESET);


Adafruit_MAX17048 maxlipo;
bool shouldEnterDeepSleep = false;
const int wakeupInterval = 15 * 1000000;

AFE44XX afe44xx(AFE44XX_CS_PIN, AFE44XX_PWDN_PIN);

afe44xx_data afe44xx_raw_data;
int32_t heart_rate_prev=0;
int32_t spo2_prev=0;

//debounce of button
// Constants for debounce
const int BUTTON_PIN = 10;
const int Deepsleeppin=9;
const unsigned long DEBOUNCE_DELAY = 50;
const unsigned long RESET_DURATION = 5000; // 5 seconds
volatile int buttonState = HIGH;
volatile int lastButtonState = HIGH;
```

```
volatile unsigned long lastDebounceTime = 0;
volatile unsigned long buttonPressStartTime = 0;
volatile bool resetTriggered = false;
unsigned long lastAfeReadTime = 0;
const unsigned long afeReadInterval = 8;  // Adjust the interval as needed
unsigned long lastDisplayUpdateTime = 0;
const unsigned long displayUpdateInterval = 1000; //# of iterations before oled update
int voltageState;

//define diferent states for oled
typedef enum stateDsiplay {Bat, Data}
stateType;

volatile stateType oledDisplay = Bat;

//interupt func for different states of oled when button is pressed
void handleButtonPress() {
 unsigned long currentMillis = millis();

 // Check if enough time has passed since the last button press
 if (currentMillis - lastDebounceTime > DEBOUNCE_DELAY) {
  int reading = digitalRead(BUTTON_PIN);

  if (reading != lastButtonState) {
   lastDebounceTime = currentMillis;
   lastButtonState = reading;

   if (lastButtonState == HIGH) {
    // Button pressed, start/reset the timer
    buttonPressStartTime = currentMillis;
    resetTriggered = false;
   } else {
    // Button released, reset timer and trigger reset if held for 5 seconds
    buttonPressStartTime = 0;
    resetTriggered = false;
   }

   // Move this part outside the inner if statement
   if (lastButtonState == HIGH) {
    // Button pressed, toggle OLED display state
    if (oledDisplay == Bat) {
     oledDisplay = Data;
    } else {
     oledDisplay = Bat;
    }
   }
  }
 }
}
```

```
// Callback function called when data is sent
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
  //Serial.print("\r\nLast Packet Send Status:\t");
 // Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Delivery Success" : "Delivery Fail");
}
void updateDisplay() {
  // switch case for oled state
    switch (oledDisplay)
  {
   //display battery precent and charge discharge rate
  case Bat:
    display.clearDisplay();
  display.drawBitmap(0,4,Battery_100,32,32,1);
  display.setCursor(42,0);
  display.print("ChgR:");display.print(maxlipo.chargeRate()); display.print(" %/hr");
  display.setCursor(40,15);
  display.print("Bat:");display.print(maxlipo.cellPercent(), 1); display.print(" %");
  display.display();
   break;
  //display spo2 and heart rate
  case Data:
  display.clearDisplay();
  display.drawBitmap(0,4,Heart_Pulse,32,32,1);
  display.setCursor(42,0);
  if (afe44xx_raw_data.spo2 == -999){
  display.print("Probe error!!!");

  }
  else{
  if(afe44xx_raw_data.spo2 <= 80){
   display.print("SPO2:");display.print("Calc...");
  }
  else{
  display.print("SPO2:");display.print(afe44xx_raw_data.spo2);display.print(" %");
  }
  display.setCursor(40,15);
  display.print("BPM:");display.print(afe44xx_raw_data.heart_rate);display.print(" bpm");
  display.display();
   break;
  }
  }
     }


void setup()
{
 /* pinMode(Deepsleeppin,INPUT_PULLUP);
```

```
  voltageState = digitalRead(Deepsleeppin);
  if (voltageState == HIGH) {
    Serial.println("Entering deep sleep");
   // Configure deep sleep
    esp_sleep_enable_timer_wakeup(wakeupInterval); // Wake up every 60 seconds
    // Enter deep sleep mode
    esp_deep_sleep_start();
  }
*/
  Serial.begin(57600);
  // Set ESP32 as a Wi-Fi Station
  WiFi.mode(WIFI_STA);
  // Initilize ESP-NOW
  if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
  }
   // Register the send callback
  esp_now_register_send_cb(OnDataSent);
    // Register peer
  memcpy(peerInfo.peer_addr, broadcastAddress, 6);
  peerInfo.channel = 0;
  peerInfo.encrypt = false;

  // Add peer
  if (esp_now_add_peer(&peerInfo) != ESP_OK){
    Serial.println("Failed to add peer");
    return;
  }

//INIT  oled button
 pinMode(10,INPUT);
attachInterrupt(digitalPinToInterrupt(BUTTON_PIN), handleButtonPress, CHANGE);
esp_sleep_enable_ext0_wakeup(GPIO_NUM_10,1); //wake up pin if module is in deep sleep



  // Start Wire library for I2C
  Wire.begin();
  // initialize OLED with I2C addr 0x3C
  display.begin(SSD1306_SWITCHCAPVCC, 0x3C);
    // Clear the display
  display.clearDisplay();
  //Set the color - always use white despite actual display color
  display.setTextColor(WHITE);
  //Set the font size
  display.setTextSize(1.3);
  //Set the cursor coordinates
  display.setCursor(0,0);
```

```
  display.print("Pulse Oxi Config...");
  display.display();
  SPI.begin();
  //Serial.println("Intilaziting AFE44xx.. ");
  delay(2000) ;   // pause for a moment
  afe44xx.afe44xx_init();
   display.clearDisplay();
  display.setCursor(0,0);
  display.print("Inited...");
  display.display();
  //Serial.println("Inited...");
  delay(1000);
  display.clearDisplay();
  display.setCursor(0,0);
  display.print("Calculating...");
  //Serial.println("Calculating...");
  display.display();

  //check if max module is present if not send error
  if (!maxlipo.begin()) {
    Serial.println(F("Couldnt find Adafruit MAX17048?\nMake sure a battery is plugged in!"));
    while (1) delay(10);
  }

}


void loop()
{

 /*voltageState = digitalRead(Deepsleeppin); // Read digital voltage
 switch (shouldEnterDeepSleep)
{
case true:
    Serial.println("Entering deep sleep");
   // Configure deep sleep
    esp_sleep_enable_timer_wakeup(wakeupInterval); // Wake up every 60 seconds
    // Enter deep sleep mode
    esp_deep_sleep_start();
    Serial.println("this will never be printed");
  break;

default:
  break;
}
  if (voltageState == HIGH) {
    shouldEnterDeepSleep = true;
  } */
 unsigned long currentMillis = millis();
```

```
 // Check if the button is being held down for 5 seconds
 if (buttonPressStartTime != 0 && !resetTriggered && currentMillis - buttonPressStartTime >=
RESET_DURATION) {
   // Button held for 5 seconds, trigger reset
   resetTriggered = true;
   display.clearDisplay();
   display.setCursor(0,0);
   display.println("Entering DeepSleep");
   display.println("GOOD NIGHT");
   display.display();
   delay(5000);
   display.clearDisplay();
   display.display();
   esp_deep_sleep_start();
 }
   delay(8);
   afe44xx.get_AFE44XX_Data(&afe44xx_raw_data);
 //displaying raw Red Data
 //Serial.println(afe44xx_raw_data.RED_data);
 //Serial.println(afe44xx_raw_data.IR_data);
 MediData.RED = afe44xx_raw_data.RED_data;
 MediData.IR = afe44xx_raw_data.IR_data;
 //buffer for spo2 and heart rate
 if (afe44xx_raw_data.buffer_count_overflow)
   {
     if(afe44xx_raw_data.spo2 == -999)
     {
       Serial.println("Probe error!!!!");
     }
     //displays new spo2 and Bpm if theres change
     else if ((heart_rate_prev != afe44xx_raw_data.heart_rate) || (spo2_prev != afe44xx_raw_data.spo2))
     {
       heart_rate_prev = afe44xx_raw_data.heart_rate;
       spo2_prev = afe44xx_raw_data.spo2;

      // Serial.print("calculating sp02...");
       //Serial.print(" Sp02 : ");
       Serial.print(afe44xx_raw_data.spo2);
       MediData.SPO2 = afe44xx_raw_data.spo2;
      // Serial.print("% ,");
      // Serial.print("Pulse rate:");
      Serial.print(afe44xx_raw_data.heart_rate);
      MediData.BPM =afe44xx_raw_data.heart_rate;
    //  Serial.println(" bpm");
```

```
    }



}
// Check for display update based on time
  if (currentMillis - lastDisplayUpdateTime >= displayUpdateInterval) {
    updateDisplay();
    lastDisplayUpdateTime = currentMillis;
  }
// Send message via ESP-NOW
  esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &MediData, sizeof(MediData));
}
```

5.6 GRAPHICAL USER INTERFACE (GUI) AND RECEIVER

Master Receiver Code:

This code is the Arduino code of the ESP32-S3 Feather that is connected to the user's computer. The most important part of this code are the structs that separated each sensor's data. This code also differentiates the received data using their corresponding name as a way to aid the Python GUI program later on when it comes to reading individual sensors.

```
#include <esp_now.h>
#include <WiFi.h>
//replace with temp sensor mac address
uint8_t tempAddress[] = {0xDC, 0x54, 0x75, 0xC3, 0xBF, 0x18};
uint8_t impad[] = {0xDC,0x54,0x75,0xC3,0x07,0x98};

// Define peerInfo structures for each peer
esp_now_peer_info_t peer1Info;
esp_now_peer_info_t peer2Info;


typedef struct TempR{
   unsigned int R1 = 9820;
   unsigned int R2 = 9897;
   unsigned int R3 = 9860;
} TempR;
 TempR TempCali;

//calibration input from master
typedef struct ImpFlag{
   bool cal;
   bool measure;
} ImpFlag;
ImpFlag Flag;
esp_now_peer_info_t peerInfo;
```

```
// Must match the receiver structure
typedef struct OXI {
  uint16_t IR;  // Use uint16_t for IR to save space
  uint16_t RED; // Use uint16_t for RED to save space
  uint8_t BPM;
  uint8_t SPO2;
} OXI;

typedef struct Ecg {
  int ECG;
} Ecg;

typedef struct Temp{
  double Temp;
} Temp;

typedef struct Imp{ //Send to receiver
  double Impavg;
  double Imps;
} Imp;




OXI MediData;
Ecg ECGData;
Temp TempData;
Imp ImpData;

// Union to hold different types of data
typedef union {
  OXI mediData;
  Ecg ecgData;
  Temp tempData;
} DataUnion;

DataUnion receivedData;
void OnDataRecv
(const uint8_t * mac, const uint8_t *incomingData, int len) {

  if (len == sizeof(OXI)) {
    memcpy(&MediData, incomingData, sizeof(MediData));
  Serial.print("Red Values:");
  Serial.println(MediData.RED);
  Serial.print("IR Values:");
  Serial.println(MediData.IR);
```

```
  Serial.print(" Sp02 : ");
  Serial.print(MediData.SPO2);
  Serial.print("% ,");
  Serial.print("Pulse rate:");
  Serial.println(MediData.BPM);
  Serial.println(" bpm");
  }
  else if (len == sizeof(Ecg)) {
  memcpy(&ECGData, incomingData, sizeof(ECGData));
  Serial.print("ECG:");
  Serial.println(ECGData.ECG);
  }
  else if (len == sizeof(Temp)) {
  memcpy(&TempData, incomingData, sizeof(TempData));
  Serial.print("Temperature:");
  Serial.println(TempData.Temp);
  }
  else if (len == sizeof(Imp)) {
  memcpy(&ImpData, incomingData, sizeof(ImpData));
  Serial.print("ImpAvg:");
  Serial.println(ImpData.Impavg);
  Serial.print("Imp:");
  Serial.println(ImpData.Imps);
  }
}

void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
  Serial.print("\r\nLast Packet Send Status:\t");
  Serial.println(status == ESP_NOW_SEND_SUCCESS ? "Delivery Success" : "Delivery Fail");
}

void handleResistorData(String R) {
  // display r values if inputed
  if (R.startsWith("Resistor1:")) {
    // Extract and use the value, e.g., convert to integer
    int resistor1Value = R.substring(10).toInt();
    TempCali.R1 = resistor1Value;
    // Implement your logic with resistor1Value
  }
  else if (R.startsWith("Resistor2:")) {
    int resistor2Value = R.substring(10).toInt();
    TempCali.R2 = resistor2Value;
    // Handle Resistor2 value
  }
  else if (R.startsWith("Resistor3:")) {
    int resistor3Value = R.substring(10).toInt();
    TempCali.R3 = resistor3Value;
    // Handle Resistor3 value
  }
```

```
  esp_err_t result = esp_now_send(tempAddress, (uint8_t *) &TempCali, sizeof(TempCali));
}

void handleImpeadanceData(String IN){
  if (IN.startsWith("Cali:")) {
    // Extract the calibration flag
    bool cali = 1;
    Flag.cal = cali;
    esp_err_t result = esp_now_send(impad, (uint8_t *) &ImpData, sizeof(ImpData));
    Flag.cal = 0;
  }
  else if (IN.startsWith("measure:1")) {
    bool meas = 1;

    Serial.println("Performing impedance measurement...");
    Flag.measure = meas;
    esp_err_t result = esp_now_send(impad, (uint8_t *) &ImpData, sizeof(ImpData));
    Flag.measure= 0;
  }
}

void setup() {
  // Set up Serial Monitor
  Serial.begin(500000);
  // Define the total number of data points
  pinMode(13,OUTPUT);
  // Set ESP32 as a Wi-Fi Station
  WiFi.mode(WIFI_STA);

  // Initilize ESP-NOW
  if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
  }

  // Register callback function
  esp_now_register_recv_cb(OnDataRecv);

  // Once ESPNow is successfully Init, we will register for Send CB to
  // get the status of Trasnmitted packet
  esp_now_register_send_cb(OnDataSent);

  // Register and add peer1
  memcpy(peer1Info.peer_addr, tempAddress, 6);
  peer1Info.channel = 0;
  peer1Info.encrypt = false;
  if (esp_now_add_peer(&peer1Info) != ESP_OK) {
    Serial.println("Failed to add peer1");
    return;
```

```
  }

  // Register and add peer2
  memcpy(peer2Info.peer_addr, impad, 6);
  peer2Info.channel = 0;
  peer2Info.encrypt = false;
  if (esp_now_add_peer(&peer2Info) != ESP_OK) {
    Serial.println("Failed to add peer2");
    return;
  }

}

void loop() {
  if (Serial.available() > 0) {
    String receivedData = Serial.readStringUntil('\n'); // Read data until newline character
    //Serial.print("Received from Python: ");
    //Serial.println(receivedData);
    handleResistorData(receivedData);
    handleImpeadanceData(receivedData);
  }
}
```

Python Program GUI Code:

This code revolves around the customtkinter library for appearances. As for the live graphs, the library matplotlib is used. The most important aspects of the code are the pages that represent each sensor as well as the choosing mechanism of the main menu. In terms of the saving aspect, the home page plays an important role since this initializes the folders needed for data to be stored.

```
# -*- coding: utf-8 -*-
"""
Created on Tue Oct  3 13:02:50 2023

@author: Moath Alsayar
@edited by: Carmella Ocaya
    Fri Dec  29 23:59:00 2023

"""
import serial
import tkinter as tk
from tkinter import ttk
import customtkinter as ctk
import matplotlib as plt
import serial.tools.list_ports as port_list
from matplotlib.figure import Figure
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.backend_bases import MouseButton
```

```
import time
import datetime
from datetime import timedelta
from matplotlib.backends.backend_tkagg import NavigationToolbar2Tk
import psutil
import os
import threading
import logging
import numpy as np
LARGEFONT = ("Comic Sans", 45, "bold")
MidFONT = ("Comic Sans", 25, "bold")
regFont = ("Comic Sans",15,"bold" )
smallFont =("Comic Sans",13,"bold" )
ctk.set_appearance_mode("dark")

class loginPage(ctk.CTkFrame):
    def __init__(self, parent, controller):
        ctk.CTkFrame.__init__(self, parent)

        label = ctk.CTkLabel(self, text="MediBrick2000", font=LARGEFONT, fg_color="#bf2c19",
corner_radius=10)
        label.pack(side="top", pady=15)

        userlab=ctk.CTkLabel(self, text="Enter Username", font=MidFONT, fg_color="#bf2c19", corner_radius=10)
        userlab.pack()

        self.warning_label = ctk.CTkLabel(self, text="Avoid \\ / : * ? \" < > |", fg_color="#bf2c19", font=("Comic
Sans", 16), corner_radius=10)
        self.warning_label.pack(pady=5)

        self.text_input = ctk.CTkEntry(self)
        self.text_input.pack(pady=5)

        enter = ctk.CTkButton(self, text="Enter", font=MidFONT, command=lambda: self.on_enter_click(controller),
fg_color="#bf2c19", corner_radius=10)
        enter.pack(pady=5)

    def show_warning_popup(self, message):
        popup = tk.Toplevel()
        popup.title("Warning")

        label = tk.Label(popup, text=message, fg="#bf2c19")  # Set the text color with fg
        label.pack(padx=10, pady=10)

        button = tk.Button(popup, text="OK", command=popup.destroy)
        button.pack(pady=10)

    def on_enter_click(self, controller):
        # Get the user input from the text box
```

```
        user_input = self.text_input.get()

        # Check if the user input is empty or contains invalid characters
        if not user_input or not self.is_valid_folder_name(user_input):
            self.show_warning_popup("Invalid folder name. Please enter a valid name")
            return

        # Get the directory of the current Python script
        current_directory = os.path.dirname(__file__)

        # Create the main user folder in the current directory
        user_folder_path = os.path.join(current_directory, user_input)
        os.makedirs(user_folder_path, exist_ok=True)

        # Create subfolders inside the user folder
        subfolders = ["Pulse Oximetry Data", "ECG Data", "Temperature Data", "Skin Impedance Data", "Digital
Stethoscope Data"]
        for subfolder in subfolders:
            os.makedirs(os.path.join(user_folder_path, subfolder), exist_ok=True)

        # Continue with navigating to the next frame (e.g., StartPage)
        controller.show_frame(StartPage)

    def is_valid_folder_name(self, folder_name):
        # Check if the folder name contains invalid characters
        invalid_characters = set(r'\/:*?"<>|.')
        return not any(char in invalid_characters for char in folder_name)

class tkinterApp(tk.Tk):
    def __init__(self, *args, **kwargs):
        tk.Tk.__init__(self, *args, **kwargs)
        self.title("MediBrick2000")
        container = ctk.CTkFrame(self)
        container.pack(side="top", fill="both", expand=True)
        container.grid_rowconfigure(0, weight=1)
        container.grid_columnconfigure(0, weight=1)
        self.frames = {}
        for F in (loginPage, StartPage, Page1, Page2, Page3, Page4, Page5):
            frame = F(container, self)
            self.frames[F] = frame
            frame.grid(row=0, column=0, sticky="nsew")
        self.show_frame(loginPage)

    def show_frame(self, cont):
        frame = self.frames[cont]
        frame.tkraise()

class StartPage(ctk.CTkFrame):
    def __init__(self, parent, controller):
```

```
        ctk.CTkFrame.__init__(self, parent)

        label = ctk.CTkLabel(self, text="MediBrick2000", font=LARGEFONT, fg_color="#bf2c19",
corner_radius=10)
        label.pack(pady = 5)

        button1 = ctk.CTkButton(self, text="Pulse Oximeter", command=lambda: controller.show_frame(Page1),
fg_color="#f2a138",font = regFont)
        button1.pack(pady = 1)

        button2 = ctk.CTkButton(self, text="Electrocardiogram (ECG)", command=lambda:
controller.show_frame(Page2), fg_color="#f2a138",font = regFont)
        button2.pack(pady = 1)

        button3 = ctk.CTkButton(self, text="Temperature Sensor", command=lambda: controller.show_frame(Page3),
fg_color="#f2a138",font = regFont)
        button3.pack(pady = 1)

        button4 = ctk.CTkButton(self, text="Body Fat and Water Composition (Skin Impedance)", command=lambda:
controller.show_frame(Page4), fg_color="#f2a138",font = regFont)
        button4.pack(pady = 1)

        button5 = ctk.CTkButton(self, text="Digital Stethoscope", command=lambda: controller.show_frame(Page5),
fg_color="#f2a138",font = regFont)
        button5.pack(pady = 1)

        port_label = ctk.CTkLabel(self, text="Select COM Port:",font = smallFont)
        port_label.pack()

        # Add a dropdown to select the port
        ports = [port.device for port in port_list.comports()]
        port_var = ctk.StringVar(self)
        port_menu = ctk.CTkComboBox(self, values=ports ,variable=port_var,corner_radius=10 )
        port_menu.pack()
        port_var.set(ports[0] if ports else "No ports available")

        baud_label = ctk.CTkLabel(self, text="Select Baud Rate:",font = smallFont)
        baud_label.pack()

        baud_var = ctk.IntVar(self)
        baud_var.set(500000)
        # Convert baud rates to strings
        baud_values = ["9600", "19200", "38400", "57600", "115200",'500000']
        baud_menu = ctk.CTkComboBox(self, values = baud_values ,variable = baud_var,corner_radius= 10)
        baud_menu.pack()

        connect_button = ctk.CTkButton(self, text="Connect", command=lambda: self.connect_to_serial(port_var,
baud_var))
        connect_button.place(x=570,y=330)
```

```
        self.themeMode = "dark"
        switch_button = ctk.CTkSwitch(self, command=self.toggle_theme,text="Light/Dark Mode")
        switch_button.place(x=3,y=3)
    def toggle_theme(self):

        # Add logic here to toggle the theme between light and dark mode
        if self.themeMode == "dark":
         ctk.set_appearance_mode("light")
         self.themeMode = "light"
        else:
            ctk.set_appearance_mode("dark")
            self.themeMode = "dark"

    def connect_to_serial(self, port_var, baud_var):
       selected_port = port_var.get()
       selected_baud = baud_var.get()

       try:
           # Establish the serial connection
           serial.Serial(selected_port, selected_baud, timeout=1)
           print(f"Connected to {selected_port} at {selected_baud} baud")
           # Continue with other actions (e.g., navigate to the data acquisition page)
       except serial.SerialException as e:
           print(f"Error: {e}")
           # Handle the connection error (e.g., display an error message)

class Page1(ctk.CTkFrame):
    def __init__(self, parent, controller):
        self.start_time = 0  # Start the time at 0
        ctk.CTkFrame.__init__(self, parent)
        self.exit_flag = True
        self.data_counter = 0
        self.x_data = []
        self.y_data = []
        self.x_data_ir = []
        self.y_data_ir = []
        self.save_data = False  # Make save_data a class attribute
        self.line_ir = None

        button1 = ctk.CTkButton(self, text="MediBrick2000 Home", command=lambda:
stop_reading_and_show_home(controller), fg_color="#ed9818")
        button1.place(x=1,y=1)

        # Create a label to display the received data
        data_label = ctk.CTkLabel(self, text="Pulse Oximeter Data", font=LARGEFONT)
        data_label.pack()

        # Create labels to display the initialization status, SPO2, and BPM
```

```
status_label = ctk.CTkLabel(self, text="Initializing AFE44xx...")
status_label.pack()

spo2_label = ctk.CTkLabel(self, text="SPO2: ", font=LARGEFONT)
spo2_label.pack()

bpm_label = ctk.CTkLabel(self, text="BPM: ", font=LARGEFONT)
bpm_label.pack()

# Get available serial ports
ports = [port.device for port in port_list.comports()]
if not ports:
    print("No serial ports available. Please check your connections.")
    exit(1)

# Create a combo box for selecting the serial port
port_label = ctk.CTkLabel(self, text="Select COM Port:")

port_label.pack()

port_var = ctk.StringVar(self)
port_menu = ctk.CTkComboBox(self, values=ports ,variable=port_var,corner_radius=10 )
port_menu.pack()
port_var.set(ports[0] if ports else "No ports available")

baud_label = ctk.CTkLabel(self, text="Select Baud Rate:")
baud_label.pack()

baud_var = ctk.IntVar(self)
baud_var.set(500000)
# Convert baud rates to strings
baud_values = ["9600", "19200", "38400", "57600", "115200",'500000']
baud_menu = ctk.CTkComboBox(self, values = baud_values ,variable = baud_var,corner_radius= 10)
baud_menu.pack()

# Create a subframe for the Matplotlib graph
graph_frame = ttk.Frame(self)
graph_frame.pack()

# Create a Matplotlib figure and subplot
# Create a Matplotlib figure and subplot
fig = Figure(figsize=(13, 3.3), dpi=100, facecolor='dimgrey')
self.ax = fig.add_subplot(111)  # Define self.ax here
self.ax.set_facecolor('black')
self.ax.set_xlabel('Time')
self.ax.set_ylabel('Raw Values')
self.line, = self.ax.plot([], [], 'b-')

# Create a canvas for the Matplotlib figure
```

```
canvas = FigureCanvasTkAgg(fig, master=graph_frame)
canvas.draw()
canvas.get_tk_widget().pack(side=tk.TOP, fill=tk.BOTH, expand=True)

# Create an instance of the toolbar
toolbar = NavigationToolbar2Tk(canvas, self)
toolbar.pack()
toolbar.update()

#x_data, y_data = [], []
# Create buttons for starting and stopping data saving
start_button = ctk.CTkButton(self, text="Start Saving Data",fg_color = "#3cb043")
start_button.place(x=3,y=250)

stop_button = ctk.CTkButton(self, text="Stop Saving Data", fg_color= "#bf2f24")
stop_button.place(x=150,y=250)

save_data = False  # Initialize the save flag

def stop_reading_and_show_home(controller):
   stop_reading()
   controller.show_frame(StartPage)

def start_saving_data():

   self.save_data = True

def stop_saving_data():
   self.save_data = False
   if len(self.x_data) >= 500:  # Save data only if there are at least 1000 points
      save_to_csv()  # Save data to CSV when stop is pressed

def save_to_csv():
   def find_folder_path(folder_name):
      for root, dirs, files in os.walk(os.path.dirname(os.path.abspath(__file__))):  # Starting from GUI directory
         if folder_name in dirs:
            return os.path.join(root, folder_name)

   folder_name_to_find = "Pulse Oximetry Data" #Replace with folder name
   folder_path = find_folder_path(folder_name_to_find)

   file_name = "PulseOxi.csv"
   file_path = os.path.join(folder_path, file_name)

   print("Saving to CSV...")

   with open(file_path, "a") as file:
    if os.path.getsize(file_path) == 0:
     file.write("Time, Red Values, IR Values\n")
```

```
            start_index = max(0, len(self.x_data) - 500)

            for i in range(start_index, len(self.x_data)):
              file.write(f"{self.x_data[i]}, {self.y_data[i]}, {self.y_data_ir[i]}\n")

          print(f"File saved to: {os.path.abspath(file_name)}")  # Print the absolute path of the file

        # Configure button commands
        start_button.configure(command=start_saving_data)
        stop_button.configure(command=stop_saving_data)

        # Define global variables
        exit_flag = False

        ser = None  # Initialize serial connection object

        def start_reading():
          self.exit_flag = False  # Access exit_flag using self

          ser_thread = threading.Thread(target=read_serial_data)
          ser_thread.start()

        # Function to stop reading serial data
        def stop_reading():
            self.exit_flag = True  # Access exit_flag using self

        # Create buttons for starting and stopping reading
        start_reading_button = ctk.CTkButton(self, text="Start Reading Data", fg_color="#3cb043",
command=start_reading)
        start_reading_button.place(x=3,y=200)

        stop_reading_button = ctk.CTkButton(self, text="Stop Reading Data", fg_color="#bf2f24",
command=stop_reading)
        stop_reading_button.place(x=150,y=200)

        def read_serial_data():
            global exit_flag, ser,x_data,y_data, data_counter
            ser = None  # Initialize ser here
            data_buffer_red = []  # Buffer to accumulate Red data points
            data_buffer_ir = []  # Buffer to accumulate IR data points
            batch_size = 20  # Set the size for batch processing
            start_time = time.time()  # Variable to store the start time

            while not self.exit_flag:  # Access exit_flag using self
                try:
                    # Open serial connection if not open
                    if ser is None or not ser.is_open:
                        selected_port = port_var.get()
```

```
        selected_baud = baud_var.get()
        ser = serial.Serial(selected_port, selected_baud, timeout=1)
        print(f"Connected to {selected_port} at {selected_baud} baud")

    # Read incoming serial data
    if ser.in_waiting > 0:
        data = ser.readline(50).decode('utf-8').strip()

        # Check if data contains "Red Values:" or "IR Values:"
        if "Red Values:" in data:

            # Extract Red value
             # Extract Red value
          try:
           red_value = int(data.split(":")[1].strip())
           current_time = time.time() - start_time
           data_buffer_red.append((current_time, red_value))
          # print(red_value)

          except ValueError:
            print(f"Invalid Red Value: {data}")

        elif "IR Values:" in data:
            # Extract IR value
            ir_value = int(data.split(":")[1].strip())
            current_time = time.time() - start_time
            data_buffer_ir.append((current_time, ir_value))  # Append IR data to the buffer
            #print(ir_value)

        # Process batched data when buffer size reaches the specified batch size
        if len(data_buffer_red) >= batch_size and len(data_buffer_ir) >= batch_size:
            process_batch_data(data_buffer_red, data_buffer_ir)
            data_buffer_red = []  # Clear the Red buffer after processing
            data_buffer_ir = []  # Clear the IR buffer after processing
            time.sleep(0.005)  # Introduce a small delay to throttle the data processing

        # Handle initialization and other messages
        elif "Init" in data:
            status_label.configure(text=data)
        elif "Sp02" in data and "Pulse rate" in data:
            spo2 = data.split(",")[0].split(":")[1].strip()
            bpm = int(data.split(",")[1].split(":")[1].strip().split(" ")[0])  # Convert to integer for comparison
            if self.data_counter >= 500:  # Check after 500 data points
                if bpm > 205:
                    status_label.configure(text="Probe error please place oximeter on finger")
                else:
                    spo2_label.configure(text=f"SPO2: {spo2}")
                    bpm_label.configure(text=f"BPM: {bpm}")
```

```
        except serial.SerialException as e:
            print(f"Serial Exception: {e}")
            if ser is not None:
                ser.close()
            time.sleep(2)


line_ir = None  # Define line_ir globally

# Function to create the plot for IR values
def create_ir_plot():
    global line_ir
    self.line_ir, = self.ax.plot([], [], 'orange', label='IR Values')  # Create an initial line for IR values
    self.ax.legend(loc='upper left', bbox_to_anchor=(1, 1))  # Display legend outside the graph

# Call the function to create the plot for IR values
create_ir_plot()

# Inside the process_batch_data function
def process_batch_data(data_buffer_red, data_buffer_ir):

    x_batch_red, y_batch_red = zip(*data_buffer_red)  # Unzip Red data batch
    x_batch_ir, y_batch_ir = zip(*data_buffer_ir)  # Unzip IR data batch

    self.x_data.extend(x_batch_red)  # Update x_data with Red values' time
    self.y_data.extend(y_batch_red)  # Update y_data with Red values

    # Display the last 1000 points for Red values
    max_points = 1000
    if len(self.x_data) > max_points:
        self.x_data = self.x_data[-max_points:]
        self.y_data = self.y_data[-max_points:]

    # Plot Red values
    self.line.set_data(self.x_data, self.y_data)
    self.line.set_color('red')  # Change the color of the Red values line to red
    self.ax.relim()
    self.ax.autoscale_view()
    canvas.draw_idle()

    # Store and display the last 1000 points for IR values
    self.x_data_ir.extend(list(x_batch_ir))
    self.y_data_ir.extend(list(y_batch_ir))

    if len(self.x_data_ir) > max_points:
        self.x_data_ir = self.x_data_ir[-max_points:]
        self.y_data_ir = self.y_data_ir[-max_points:]

    # Plot IR values in the same plot
```

```
            self.line_ir.set_data(self.x_data_ir, self.y_data_ir)  # Set IR data to the plot
            self.line_ir.set_color('orange')  # Set color of IR values line to orange

            self.data_counter += len(data_buffer_red)  # Increment data counter

            # Update the legend with both Red and IR values
            self.ax.legend(['Red Values', 'IR Values'], loc='upper left', bbox_to_anchor=(1, 1))

            if self.data_counter >= 500:
                status_label.configure(text="AFE4490 Calculating...")
        # Start reading the serial data in a separate thread
        import threading
        thread = threading.Thread(target=read_serial_data)
        thread.start()

class Page2(ctk.CTkFrame):
    def __init__(self, parent, controller):
        ctk.CTkFrame.__init__(self, parent)

        self.exit_flag = True
        self.data_counter = 0
        self.x_data = []
        self.y_data = []
        self.save_data = False  # Make save_data a class attribute
        self.line_ir = None
        self.zero_ecg_time = 0

        label = ctk.CTkLabel(self, text="Electrocardiogram (ECG)", font=LARGEFONT)
        label.pack()
        button2 = ctk.CTkButton(self, text="MediBrick2000 Home", command=lambda:
stop_reading_and_show_home(controller), fg_color="#ed9818")
        button2.place(x=3,y=0)

        # Create labels to display the initialization status, SPO2, and BPM
        status_label = ctk.CTkLabel(self, text="Initializing ECG module...")
        status_label.pack()

        ECG_label = ctk.CTkLabel(self, text="ECG Val: ", font=LARGEFONT)
        ECG_label.pack()

        # Get available serial ports
        ports = [port.device for port in port_list.comports()]
        if not ports:
            print("No serial ports available. Please check your connections.")
            exit(1)
            #warning_label = ctk.CTkLabel(self, text="No ports connected. Check your connection.",
fg_color="#bf2c19")
            #warning_label.pack()
```

```
# Custom styling for the option menu
style = ttk.Style()
style.theme_use('default')
style.configure('Custom.TMenubutton', background='#fcb103')

# Create a combo box for selecting the serial port
port_label = ctk.CTkLabel(self, text="Select COM Port:")
port_label.pack()

port_var = ctk.StringVar(self)
port_menu = ctk.CTkComboBox(self, values=ports ,variable=port_var,corner_radius=10 )
port_menu.pack()
port_var.set(ports[0] if ports else "No ports available")

baud_label = ctk.CTkLabel(self, text="Select Baud Rate:")
baud_label.pack()

baud_var = ctk.IntVar(self)
# Convert baud rates to strings
baud_values = ["9600", "19200", "38400", "57600", "115200"]
baud_menu = ctk.CTkComboBox(self, values = baud_values ,variable = baud_var,corner_radius= 10)
baud_menu.pack()

# Create a subframe for the Matplotlib graph
graph_frame = ttk.Frame(self)
graph_frame.pack()

# Create a Matplotlib figure and subplot
fig = Figure(figsize=(14, 3.8), dpi=100, facecolor='dimgrey')
self.ax = fig.add_subplot(111)  # Define self.ax here
self.ax.set_facecolor('black')
self.ax.set_xlabel('Time')
self.ax.set_ylabel('ECG Val:')
self.line, = self.ax.plot([], [], 'b-')

# Create a canvas for the Matplotlib figure
canvas = FigureCanvasTkAgg(fig, master=graph_frame)
canvas.draw()
canvas.get_tk_widget().pack(side=tk.TOP, fill=tk.BOTH, expand=True)

# Create an instance of the toolbar
toolbar = NavigationToolbar2Tk(canvas, self)
toolbar.pack()
toolbar.update()

#x_data, y_data = [], []
# Create buttons for starting and stopping data saving
start_button = ctk.CTkButton(self, text="Start Saving Data",fg_color = "#3cb043")
start_button.place(x=3,y=200)
```

```
stop_button = ctk.CTkButton(self, text="Stop Saving Data", fg_color= "#bf2f24")
stop_button.place(x=150,y=200)

save_data = False  # Initialize the save flag

def stop_reading_and_show_home(controller):
    stop_reading()
    controller.show_frame(StartPage)


def start_saving_data():
    self.save_data = True

def stop_saving_data():
    self.save_data = False
    if len(self.x_data) >= 300:  # Save data only if there are at least 1000 points
        save_to_csv()  # Save data to CSV when stop is pressed

def save_to_csv():
    def find_folder_path(folder_name):
        for root, dirs, files in os.walk(os.path.dirname(os.path.abspath(__file__))):  # Starting from GUI directory
            if folder_name in dirs:
                return os.path.join(root, folder_name)

    folder_name_to_find = "ECG Data" #Replace with folder name
    folder_path = find_folder_path(folder_name_to_find)

    #folder_name = "ECG Data"
    #folder_path = os.path.dirname(folder_name)
    file_name = "ECG.csv"
    file_path = os.path.join(folder_path, file_name)

    print("Saving to CSV...")

    with open(file_path, "a") as file:
        if os.path.getsize(file_path) == 0:
            file.write("Time, ECG VAL\n")

        start_index = max(0, len(self.x_data) - 1000)

        for i in range(start_index, len(self.x_data)):
            file.write(f"{self.x_data[i]}, {self.y_data[i]}\n")

        print(f"File saved to: {os.path.abspath(file_name)}")

# Configure button commands
start_button.configure(command=start_saving_data)
stop_button.configure(command=stop_saving_data)
```

```
# Define global variables
exit_flag = False

ser = None  # Initialize serial connection object

def start_reading():
  self.exit_flag = False  # Access exit_flag using self

  ser_thread = threading.Thread(target=read_serial_data)
  ser_thread.start()

# Function to stop reading serial data
def stop_reading():
    self.exit_flag = True  # Access exit_flag using self


# Create buttons for starting and stopping reading
start_reading_button = ctk.CTkButton(self, text="Start Reading Data", fg_color="#3cb043",
command=start_reading)
start_reading_button.place(x=3,y=150)

stop_reading_button = ctk.CTkButton(self, text="Stop Reading Data", fg_color="#bf2f24",
command=stop_reading)
stop_reading_button.place(x=150,y=150)
#function to receive the ECG data
def update_ecg_label(ecg_value):
 ECG_label.configure(text=f"ECG Val: {ecg_value}")  # Update label text with ECG value
# Inside the read_serial_data function
def read_serial_data():
 global exit_flag, ser, x_data, y_data, data_counter
 ser = None  # Initialize ser here
 data_buffer_ecg = []  # Buffer to accumulate ECG data points
 batch_size = 10  # Set the size for batch processing
 start_time = time.time()  # Variable to store the start time

 while not self.exit_flag:  # Access exit_flag directly
  try:
    # Open serial connection if not open
    if ser is None or not ser.is_open:
        selected_port = port_var.get()
        selected_baud = baud_var.get()
        ser = serial.Serial(selected_port, selected_baud, timeout=1)
        print(f"Connected to {selected_port} at {selected_baud} baud")

    # Read incoming serial data
    if ser.in_waiting > 0:
        data = ser.readline().decode('utf-8').strip()
```

```
            # Check if data contains ECG values
            if data.startswith("ECG:"):
                # Extract ECG value
                ecg_value = float(data.split(":")[1].strip())
                current_time = time.time() - start_time
                data_buffer_ecg.append((current_time, ecg_value))  # Append ECG data to the buffer
                # Update ECG label with the received value
                update_ecg_label(ecg_value)
                # Check if ECG value is zero
                if ecg_value == 0:
                   if self.zero_ecg_time == 0:
                       self.zero_ecg_time = current_time
                   elif current_time - self.zero_ecg_time >= 3:  # Check if zero for 3 seconds
                       status_label.configure(text="Check for Pulse!!")  # Update status label
                else:
                    self.zero_ecg_time = 0  # Reset zero ECG time if value is not zero
            # Process batched data when buffer size reaches the specified batch size
            if len(data_buffer_ecg) >= batch_size:
                process_batch_data(data_buffer_ecg)  # Process the ECG batch data
                data_buffer_ecg = []  # Clear the ECG buffer after processing
            # Handle other messages or initialization here
      except serial.SerialException as e:
        print(f"Serial Exception: {e}")
        if ser is not None:
            ser.close()
        time.sleep(2)

# Inside the process_batch_data function
    def process_batch_data(data_buffer_ecg):
      global x_data, y_data

      x_batch_ecg, y_batch_ecg = zip(*data_buffer_ecg)  # Unzip ECG data batch

      self.x_data.extend(x_batch_ecg)  # Update x_data with ECG values' time
      self.y_data.extend(y_batch_ecg)  # Update y_data with ECG values

# Plot ECG values or perform further processing as needed

# Display the last 1000 points for ECG values
      max_points = 500
      if len(self.x_data) > max_points:
       self.x_data = self.x_data[-max_points:]
       self.y_data = self.y_data[-max_points:]

# Plot ECG values
      self.line.set_data(self.x_data, self.y_data)
      self.line.set_color('Red')  # Change the color of the ECG values line to blue
      self.ax.relim()
      self.ax.autoscale_view()
```

```
          canvas.draw_idle()

          self.data_counter += len(data_buffer_ecg)  # Increment data counter

          status_label.configure(text="ECG in Operation...")


    # Update the legend with ECG values
          self.ax.legend(['ECG Values'], loc='upper left', bbox_to_anchor=(1, 1))
          # Start reading the serial data in a separate thread
        import threading
        thread = threading.Thread(target=read_serial_data)
        thread.start()

class Page3(ctk.CTkFrame):
    def __init__(self, parent, controller):
        self.exit_flag = True
        self.data_counter = 0
        self.x_data = []
        self.y_data = []
        self.save_data = False  # Make save_data a class attribute
        self.line_ir = None
        ctk.CTkFrame.__init__(self, parent)
        self.label = ctk.CTkLabel(self, text="Temperature Sensor", font=LARGEFONT)
        self.label.pack()

        self.button3 = ctk.CTkButton(self, text="MediBrick2000 Home", command=lambda:
stop_reading_and_show_home(controller), fg_color="#ed9818")
        self.button3.place(x=3,y=0)
        # Configure logging to a file
      #  logging.basicConfig(filename='uart_log.txt', level=logging.INFO, format='%(asctime)s - %(message)s')
        # Create labels to display the initialization status, temp
        status_label = ctk.CTkLabel(self, text="Initializing Temperature Sensor...")
        status_label.pack()

        temp_label = ctk.CTkLabel(self, text="Temperature: ", font=LARGEFONT)
        temp_label.pack()
        # Create entry fields for resistor values
        resistor1_label = ctk.CTkLabel(self, text="Resistor 1 Value:")
        resistor1_label.place(x=995, y=10)

        self.resistor1_entry = ctk.CTkEntry(self)
        self.resistor1_entry.place(x=1100, y=10)

        resistor2_label = ctk.CTkLabel(self, text="Resistor 2 Value:")
        resistor2_label.place(x=995, y=70)

        self.resistor2_entry = ctk.CTkEntry(self)
        self.resistor2_entry.place(x=1100, y=70)
```

```
        resistor3_label = ctk.CTkLabel(self, text="Resistor 3 Value:")
        resistor3_label.place(x=995, y=130)

        self.resistor3_entry = ctk.CTkEntry(self)
        self.resistor3_entry.place(x=1100, y=130)

       # Create buttons for sending resistor values
        send_resistor1_button = ctk.CTkButton(self, text="Send Resistor 1", fg_color="#3cb043",
command=self.send_resistor1)
        send_resistor1_button.place(x=1100, y=40)

        send_resistor2_button = ctk.CTkButton(self, text="Send Resistor 2", fg_color="#3cb043",
command=self.send_resistor2)
        send_resistor2_button.place(x=1100, y=100)

        send_resistor3_button = ctk.CTkButton(self, text="Send Resistor 3", fg_color="#3cb043",
command=self.send_resistor3)
        send_resistor3_button.place(x=1100, y=160)


        # Get available serial ports
        ports = [port.device for port in port_list.comports()]
        if not ports:
            print("No serial ports available. Please check your connections.")
            exit(1)

        # Custom styling for the option menu
        style = ttk.Style()
        style.theme_use('default')
        style.configure('Custom.TMenubutton', background='#fcb103')

        # Create a combo box for selecting the serial port
        port_label = ctk.CTkLabel(self, text="Select COM Port:")
        port_label.pack()

        port_var = ctk.StringVar(self)
        port_menu = ctk.CTkComboBox(self, values=ports ,variable=port_var,corner_radius=10 )
        port_menu.pack()
        port_var.set(ports[0] if ports else "No ports available")

        baud_label = ctk.CTkLabel(self, text="Select Baud Rate:")
        baud_label.pack()

        baud_var = ctk.IntVar(self)
        # Convert baud rates to strings
        baud_values = ["9600", "19200", "38400", "57600", "115200"]
        baud_menu = ctk.CTkComboBox(self, values = baud_values ,variable = baud_var,corner_radius= 10)
        baud_menu.pack()
```

```
# Create a subframe for the Matplotlib graph
graph_frame = ttk.Frame(self)
graph_frame.pack()

# Create a Matplotlib figure and subplot
fig = Figure(figsize=(14, 3.8), dpi=100, facecolor='dimgrey')
self.ax = fig.add_subplot(111)  # Define self.ax here
self.ax.set_facecolor('black')
self.ax.set_xlabel('Time')
self.ax.set_ylabel('Temperature (°C)')
self.line, = self.ax.plot([], [], 'b-')

# Create a canvas for the Matplotlib figure
canvas = FigureCanvasTkAgg(fig, master=graph_frame)
canvas.draw()
canvas.get_tk_widget().pack(side=tk.TOP, fill=tk.BOTH, expand=True)

# Create an instance of the toolbar
toolbar = NavigationToolbar2Tk(canvas, self)
toolbar.pack()
toolbar.update()

#x_data, y_data = [], []
# Create buttons for starting and stopping data saving
start_button = ctk.CTkButton(self, text="Start Saving Data",fg_color = "#3cb043")
start_button.place(x=3,y=200)

stop_button = ctk.CTkButton(self, text="Stop Saving Data", fg_color= "#bf2f24")
stop_button.place(x=150,y=200)

save_data = False  # Initialize the save flag

def stop_reading_and_show_home(controller):
    stop_reading()
    controller.show_frame(StartPage)


def start_saving_data():
    self.save_data = True

def stop_saving_data():
    self.save_data = False
    if len(self.x_data) >= 10000:  # Save data only if there are at least 1000 points
        save_to_csv()  # Save data to CSV when stop is pressed

def save_to_csv():
    def find_folder_path(folder_name):
        for root, dirs, files in os.walk(os.path.dirname(os.path.abspath(__file__))):  # Starting from GUI directory
```

```
                if folder_name in dirs:
                    return os.path.join(root, folder_name)

            folder_name_to_find = "Temperature Data"
            folder_path = find_folder_path(folder_name_to_find)

            file_name = "Temp.csv"
            file_path = os.path.join(folder_path, file_name)

            print("Saving to CSV...")

            with open(file_path, "a") as file:
                if os.path.getsize(file_path) == 0:
                    file.write("Time, Temperature °C\n")

                start_index = max(0, len(self.x_data) - 10000)

                for i in range(start_index, len(self.x_data)):
                    file.write(f"{self.x_data[i]}, {self.y_data[i]}\n")

                print(f"File saved to: {os.path.abspath(file_name)}")

        # Configure button commands
        start_button.configure(command=start_saving_data)
        stop_button.configure(command=stop_saving_data)

        # Define global variables
        exit_flag = False

        ser = None  # Initialize serial connection object

        def start_reading():
          self.exit_flag = False  # Access exit_flag using self

          ser_thread = threading.Thread(target=read_serial_data)
          ser_thread.start()

        # Function to stop reading serial data
        def stop_reading():
            self.exit_flag = True  # Access exit_flag using self

        # Create buttons for starting and stopping reading
        start_reading_button = ctk.CTkButton(self, text="Start Reading Data", fg_color="#3cb043",
command=start_reading)
        start_reading_button.place(x=3,y=150)

        stop_reading_button = ctk.CTkButton(self, text="Stop Reading Data", fg_color="#bf2f24",
command=stop_reading)
        stop_reading_button.place(x=150,y=150)
```

```
#function to receive the ECG data
def update_ecg_label(temp_value):
 temp_label.configure(text=f"Temperature: {temp_value} °C")  # Update label text with ECG value
# Inside the read_serial_data function
def read_serial_data():
 global exit_flag, ser, x_data, y_data, data_counter
 ser = None  # Initialize ser here
 data_buffer_temp = []  # Buffer to accumulate temp data points
 batch_size = 150  # Set the size for batch processing
 start_time = time.time()  # Variable to store the start time

 while not self.exit_flag:  # Access exit_flag directly
  try:
    # Open serial connection if not open
    if ser is None or not ser.is_open:
        selected_port = port_var.get()
        selected_baud = baud_var.get()
        ser = serial.Serial(selected_port, selected_baud, timeout=0.1)
      # print(f"Connected to {selected_port} at {selected_baud} baud")

    # Read incoming serial data
    if ser.in_waiting > 0:
        data = ser.readline(32).decode('utf-8').strip()
        #print(data)

        # Check if data contains ECG values
        if data.startswith("Temperature:"):
            # Extract temp value
            temp_value = float(data.split(":")[1].strip())
            current_time = time.time() - start_time
            data_buffer_temp.append((current_time, temp_value))  # Append ECG data to the buffer
            # Update temp label with the received value
            update_ecg_label(temp_value)
            # Check if ECG value is zero
            if temp_value < 36 :
                  status_label.configure(text="You are very cold! Are you a lizard?!")  # Update status label
            if temp_value > 37.5:
               status_label.configure(text="You are very hot! check for fever!")  # Update status label
            if (temp_value > 36 and temp_value < 37.5):
                status_label.configure(text="Perfect Temprature! you are chilling and healthy :)")  # Update status
label
        # Process batched data when buffer size reaches the specified batch size
        if len(data_buffer_temp) >= batch_size:
            process_batch_data(data_buffer_temp)  # Process the ECG batch data
            data_buffer_temp = []  # Clear the ECG buffer after processing
            time.sleep(0.01)  # Adjust the sleep duration
        # Handle other messages or initialization here
  except serial.SerialException as e:
    print(f"Serial Exception: {e}")
```

```
            if ser is not None:
                ser.close()
            time.sleep(2)

# Inside the process_batch_data function
        def process_batch_data(data_buffer_temp):
          global x_data, y_data

          x_batch_temp, y_batch_temp = zip(*data_buffer_temp)  # Unzip temp data batch

          self.x_data.extend(x_batch_temp)  # Update x_data with time
          self.y_data.extend(y_batch_temp)  # Update y_data with temp

    # Plot temp values or perform further processing as needed

    # Display the last temp points for temp values
          max_points = 10000
          if len(self.x_data) > max_points:
           self.x_data = self.x_data[-max_points:]
           self.y_data = self.y_data[-max_points:]

    # Plot ECG values
          self.line.set_data(self.x_data, self.y_data)
          self.line.set_color('Red')  # Change the color of the temp values line to blue
          self.ax.relim()
          self.ax.autoscale_view()
          canvas.draw_idle()

          self.data_counter += len(data_buffer_temp)  # Increment data counter

    # Update the legend with ECG values
          self.ax.legend(['Temperature °C'], loc='upper left', bbox_to_anchor=(1, 1))
          # Start reading the serial data in a separate thread
        import threading
        thread = threading.Thread(target=read_serial_data)
        thread.start()

    def send_resistor1(self):
           resistor1_value = self.resistor1_entry.get()
           self.send_to_esp32(f"Resistor1:{resistor1_value}")

    def send_resistor2(self):
           resistor2_value = self.resistor2_entry.get()
           self.send_to_esp32(f"Resistor2:{resistor2_value}")

    def send_resistor3(self):
           resistor3_value = self.resistor3_entry.get()
           self.send_to_esp32(f"Resistor3:{resistor3_value}")
```

```
    def send_to_esp32(self, message):


        ser.write(message.encode('utf-8'))  # Send the message as bytes

       # logging.info(f"Sent to ESP32: {message}")

class Page4(ctk.CTkFrame):
    def __init__(self, parent, controller):
        ctk.CTkFrame.__init__(self, parent)
        self.button3 = ctk.CTkButton(self, text="MediBrick2000 Home", command=lambda:
stop_reading_and_show_home(controller), fg_color="#ed9818")
        self.button3.place(x=3,y=0)
        self.exit_flag = True
        self.data_counter = 0
        self.x_data = []
        self.y_data = []
        self.save_data = False  # Make save_data a class attribute
        self.line_ir = None

        # Configure logging to a file
        logging.basicConfig(filename='uart_log.txt', level=logging.INFO, format='%(asctime)s - %(message)s')

        label = ctk.CTkLabel(self, text="        Body Fat and Water Composition (Skin Impedance)",
font=LARGEFONT)
        label.pack()
        button2 = ctk.CTkButton(self, text="MediBrick2000 Home", command=lambda:
controller.show_frame(StartPage), fg_color="#ed9818")
        button2.place(x=3,y=0)

        # Create labels to display the initialization status, SPO2, and BPM
        status_label = ctk.CTkLabel(self, text="Initializing AD5933 module...")
        status_label.pack()

        bodyFat_label = ctk.CTkLabel(self, text="Bodyfat %: ", font=LARGEFONT)
        bodyFat_label.pack()

        waterComp_label = ctk.CTkLabel(self, text="Water Composition: ", font=LARGEFONT)
        waterComp_label.pack()

        Impeadance_label = ctk.CTkLabel(self, text="Avg Impedance: ", font=MidFONT)
        Impeadance_label.pack()

        input_label = ctk.CTkLabel(self, text="User input: ", font=LARGEFONT)
        input_label.place(x=980,y=100)

        # Create a button to trigger the sending of the 'measure:1' signal
        measure_button = ctk.CTkButton(self, text="Start Measure", command=self.send_measure)
        measure_button.place(x=3, y=150)
```

```
        logging.warning("Serial connection is not open.")

        switch_button = ctk.CTkButton(self, text="Calibrate Module",command= self.send_cali)
        switch_button.place(x=3, y=103)
        # Configure button commands

        # Create input boxes for Height, Bodyweight, and Sex
        self.height_entry = ctk.CTkEntry(self)
        self.height_label = ctk.CTkLabel(self, text="Height (cm): ")
        self.height_label.place(x=1000,y=150)
        self.height_entry.place(x=1080,y=150)

        self.bodyweight_entry = ctk.CTkEntry(self)
        self.bodyweight_label = ctk.CTkLabel(self, text="Bodyweight (kg): ")
        self.bodyweight_label.place(x=975,y=180)
        self.bodyweight_entry.place(x=1080,y=180)

        self.sex_label = ctk.CTkLabel(self, text="Sex: ")
        self.sex_label.place(x=1050,y=210)
        self.sex_var = ctk.StringVar(self)

        sex_options = ["Male", "Female"]  # Add more options if needed
        sex_combobox = ctk.CTkComboBox(self, values=sex_options, variable=self.sex_var)
        sex_combobox.place(x=1080, y=210)

        input_button = ctk.CTkButton(self, text='Send Input')
        input_button.place(x=1080,y=290)

        # Get available serial ports
        ports = [port.device for port in port_list.comports()]
        if not ports:
            print("No serial ports available. Please check your connections.")
            exit(1)
            #warning_label = ctk.CTkLabel(self, text="No ports connected. Check your connection.",
fg_color="#bf2c19")
            #warning_label.pack()

        # Custom styling for the option menu
        style = ttk.Style()
        style.theme_use('default')
        style.configure('Custom.TMenubutton', background='#fcb103')

        # Create a combo box for selecting the serial port
        port_label = ctk.CTkLabel(self, text="Select COM Port:")
        port_label.pack()

        port_var = ctk.StringVar(self)
        port_menu = ctk.CTkComboBox(self, values=ports ,variable=port_var,corner_radius=10 )
```

```
port_menu.pack()
port_var.set(ports[0] if ports else "No ports available")

baud_label = ctk.CTkLabel(self, text="Select Baud Rate:")
baud_label.pack()

baud_var = ctk.IntVar(self)
# Convert baud rates to strings
baud_values = ["9600", "19200", "38400", "57600", "115200"]
baud_menu = ctk.CTkComboBox(self, values = baud_values ,variable = baud_var,corner_radius= 10)
baud_menu.pack()

# Create a subframe for the Matplotlib graph
graph_frame = ttk.Frame(self)
graph_frame.pack()

# Create a Matplotlib figure and subplot
fig = Figure(figsize=(14, 3.8), dpi=100, facecolor='dimgrey')
self.ax = fig.add_subplot(111)  # Define self.ax here
self.ax.set_facecolor('black')
self.ax.set_xlabel('Time')
self.ax.set_ylabel('Impedance(ohms)')
self.line, = self.ax.plot([], [], 'b-')

# Create a canvas for the Matplotlib figure
canvas = FigureCanvasTkAgg(fig, master=graph_frame)
canvas.draw()
canvas.get_tk_widget().pack(side=tk.TOP, fill=tk.BOTH, expand=True)

#x_data, y_data = [], []
# Create buttons for starting and stopping data saving
start_button = ctk.CTkButton(self, text="Start Saving Data",fg_color = "#3cb043")
start_button.place(x=3,y=260)

stop_button = ctk.CTkButton(self, text="Stop Saving Data", fg_color= "#bf2f24")
stop_button.place(x=150,y=260)

save_data = False  # Initialize the save flag

def stop_reading_and_show_home(controller):
    stop_reading()
    controller.show_frame(StartPage)

def start_saving_data():
    self.save_data = True

def stop_saving_data():
    self.save_data = False
    if len(self.x_data) >= 300:  # Save data only if there are at least 1000 points
```

```
        save_to_csv()  # Save data to CSV when stop is pressed

def save_to_csv():
    def find_folder_path(folder_name):
        for root, dirs, files in os.walk(os.path.dirname(os.path.abspath(__file__))):  # Starting from GUI directory
            if folder_name in dirs:
                return os.path.join(root, folder_name)

    folder_name_to_find = "Skin Impedance Data" #Replace with folder name
    folder_path = find_folder_path(folder_name_to_find)

    #folder_name = "ECG Data"
    #folder_path = os.path.dirname(folder_name)
    file_name = "Skin Impedance Data.csv"
    file_path = os.path.join(folder_path, file_name)

    print("Saving to CSV...")

    with open(file_path, "a") as file:
        if os.path.getsize(file_path) == 0:
            file.write("Time, ECG VAL\n")

        start_index = max(0, len(self.x_data) - 1000)

        for i in range(start_index, len(self.x_data)):
            file.write(f"{self.x_data[i]}, {self.y_data[i]}\n")

        print(f"File saved to: {os.path.abspath(file_name)}")

# Configure button commands
start_button.configure(command=start_saving_data)
stop_button.configure(command=stop_saving_data)

# Define global variables
exit_flag = False

ser = None  # Initialize serial connection object

def start_reading():
  self.exit_flag = False  # Access exit_flag using self

  ser_thread = threading.Thread(target=read_serial_data)
  ser_thread.start()

# Function to stop reading serial data
def stop_reading():
    self.exit_flag = True  # Access exit_flag using self
  # Create buttons for starting and stopping reading
```

```
        start_reading_button = ctk.CTkButton(self, text="Start Reading Data", fg_color="#3cb043",
command=start_reading)
        start_reading_button.place(x=3,y=230)

        stop_reading_button = ctk.CTkButton(self, text="Stop Reading Data", fg_color="#bf2f24",
command=stop_reading)
        stop_reading_button.place(x=150,y=230)


        # Inside the read_serial_data function
        def read_serial_data():
          global exit_flag, ser, x_data, y_data, data_counter
          ser = None  # Initialize ser here
          data_buffer_imp = []  # Buffer to accumulate impedance data points
          batch_size = 2  # Set the size for batch processing

          while not self.exit_flag:  # Access exit_flag directly
            try:
              # Open serial connection if not open
              if ser is None or not ser.is_open:
                selected_port = port_var.get()
                selected_baud = baud_var.get()
                ser = serial.Serial(selected_port, selected_baud, timeout=1)
                print(f"Connected to {selected_port} at {selected_baud} baud")

              # Read incoming serial data
              if ser.in_waiting > 0:
                data = ser.readline().decode('utf-8').strip()
                print(data)

                # Check if data contains impedance values
                if data.startswith("ImpAvg:"):
                  # Extract impedance average value
                  impedance_avg = float(data.split(":")[1])
                  # Update GUI label with the received average impedance value
                  Impeadance_label.configure(text=f"Average Impedance: {impedance_avg} ohms")
                  if impedance_avg > 0:
                  # Calculate body fat percentage
                  height_cm = float(self.height_entry.get())
                  weight_kg = float(self.bodyweight_entry.get())
                  sex = 1 if self.sex_var.get() == "Male" else 0  # Male=1, Female=0
                  #age = int(self.age_entry.get())
                  #print(age)
                  print(sex)
                  print(weight_kg)
                  print(height_cm)

                  # If a better equation model was discovered for Fat-Free Mass, update below
                  FFM = (0.396*((height_cm ** 2)/(impedance_avg/200)))+(0.143*weight_kg+8.399)*1.37*2
```

```
            BFM = weight_kg - FFM
            body_fat_percentage = (BFM / weight_kg) * 100

        # Update bodyFat_label with the calculated body fat percentage
            bodyFat_label.configure(text=f"Bodyfat % {body_fat_percentage:.2f}")
            TBW = FFM * 0.73

          # Update waterComp_label with the calculated total body water
            waterComp_label.configure(text=f"Total Body Water: {TBW:.2f} liters")

          if data.startswith("Imp:"):
            # Extract impedance value
            impedance_val = float(data.split(":")[1])
            current_time = time.time()  # Get current time
            data_buffer_imp.append((current_time, impedance_val))  # Append impedance data to the buffer
            print(impedance_val)
          # Process batched data when buffer size reaches the specified batch size
          if len(data_buffer_imp) >= batch_size:
            process_batch_data(data_buffer_imp)  # Process the impedance batch data
            data_buffer_imp = []  # Clear the impedance buffer after processing

      except serial.SerialException as e:
        print(f"Serial Exception: {e}")
        if ser is not None:
          ser.close()
        time.sleep(2)

# Inside the process_batch_data function
  def process_batch_data(data_buffer_imp):
    global x_data, y_data

    x_batch_imp, y_batch_imp = zip(*data_buffer_imp)  # Unzip impedance data batch

    self.x_data.extend(x_batch_imp)  # Update x_data with impedance values' time
    self.y_data.extend(y_batch_imp)  # Update y_data with impedance values

    # Plot impedance values or perform further processing as needed

    # Display the last 1000 points for impedance values
    max_points = 500
    if len(self.x_data) > max_points:
      self.x_data = self.x_data[-max_points:]
      self.y_data = self.y_data[-max_points:]

    # Plot impedance values
    self.line.set_data(self.x_data, self.y_data)
    self.line.set_color('Green')  # Change the color of the impedance values line to green
    self.ax.relim()
    self.ax.autoscale_view()
```

```
            canvas.draw_idle()

            self.data_counter += len(data_buffer_imp)  # Increment data counter

            status_label.configure(text="Impedance Reading in Progress...")

        # Update the legend with impedance values
        self.ax.legend(['Impedance Values'], loc='upper left', bbox_to_anchor=(1, 1))

        # Start reading the serial data in a separate thread
        import threading
        thread = threading.Thread(target=read_serial_data)
        thread.start()

    def send_cali(self):
            print("cali")
            self.send_to_esp32("Cali:1")

    def send_measure(self):
        # Call the method to send the signal to ESP32
        self.send_to_esp32("measure:1")

    def send_to_esp32(self, message):


        ser.write(message.encode('utf-8'))
        print(message)
        logging.info(f"Sent to ESP32: {message}")

class Page5(ctk.CTkFrame):
    def __init__(self, parent, controller):
        ctk.CTkFrame.__init__(self, parent)

        self.exit_flag = True
        self.data_counter = 0
        self.x_data = np.array([])
        self.y_data = np.array([])
        self.save_data = False  # Make save_data a class attribute
        self.line_ir = None
        self.zero_ecg_time = 0

        label = ctk.CTkLabel(self, text="Digital Stethoscope", font=LARGEFONT)
        label.pack()
        button2 = ctk.CTkButton(self, text="MediBrick2000 Home", command=lambda:
stop_reading_and_show_home(controller), fg_color="#ed9818")
        button2.place(x=3,y=0)

        # Create labels to display the initialization status, SPO2, and BPM
        status_label = ctk.CTkLabel(self, text="Initializing digital stethoscope module...")
```

```
        status_label.pack()

        Audio_label = ctk.CTkLabel(self, text="Audio Amplituide(DB): ", font=LARGEFONT)
        Audio_label.pack()

        # Get available serial ports
        ports = [port.device for port in port_list.comports()]
        if not ports:
            print("No serial ports available. Please check your connections.")
            exit(1)
            #warning_label = ctk.CTkLabel(self, text="No ports connected. Check your connection.",
fg_color="#bf2c19")
            #warning_label.pack()

        # Custom styling for the option menu
        style = ttk.Style()
        style.theme_use('default')
        style.configure('Custom.TMenubutton', background='#fcb103')

        # Create a combo box for selecting the serial port
        port_label = ctk.CTkLabel(self, text="Select COM Port:")
        port_label.pack()

        port_var = ctk.StringVar(self)
        port_menu = ctk.CTkComboBox(self, values=ports ,variable=port_var,corner_radius=10 )
        port_menu.pack()
        port_var.set(ports[0] if ports else "No ports available")

        baud_label = ctk.CTkLabel(self, text="Select Baud Rate:")
        baud_label.pack()

        baud_var = ctk.IntVar(self)
        # Convert baud rates to strings
        baud_values = ["9600", "19200", "38400", "57600", "115200"]
        baud_menu = ctk.CTkComboBox(self, values = baud_values ,variable = baud_var,corner_radius= 10)
        baud_menu.pack()

        # Create a subframe for the Matplotlib graph
        graph_frame = ttk.Frame(self)
        graph_frame.pack()

        self.switch_var = tk.IntVar(self)
        self.switch_var.set(0)  # Set the initial state to 0 (tissue sample)

        # Create the switch (Checkbutton)
        switch_label = ctk.CTkLabel(self, text="Live Audio Mode:")
        switch_label.place(x=979,y=100)

        self.switch_text_var = tk.StringVar(self)
```

```
        self.switch_text_var.set("Audio ON")

        switch = ctk.CTkSwitch(self, textvariable=self.switch_text_var, variable=self.switch_var, onvalue=0,
offvalue=1,
                        command=self.update_switch_text)
        switch.place(x=1080,y=103)

        # Create a Matplotlib figure and subplot
        fig = Figure(figsize=(14, 3.8), dpi=100, facecolor='dimgrey')
        self.ax = fig.add_subplot(111)  # Define self.ax here
        self.ax.set_facecolor('black')
        self.ax.set_xlabel('Time')
        self.ax.set_ylabel('dB SPL')
        self.line, = self.ax.plot([], [], 'b-')

        # Create a canvas for the Matplotlib figure
        canvas = FigureCanvasTkAgg(fig, master=graph_frame)
        canvas.draw()
        canvas.get_tk_widget().pack(side=tk.TOP, fill=tk.BOTH, expand=True)

        # Create an instance of the toolbar
        toolbar = NavigationToolbar2Tk(canvas, self)
        toolbar.pack()
        toolbar.update()

        # Create buttons for starting and stopping data saving
        start_rec = ctk.CTkButton(self, text="Start Recording",fg_color = "#3cb043")
        start_rec.place(x=900,y=200)

        stop_rec = ctk.CTkButton(self, text="Stop Recording", fg_color= "#bf2f24")
        stop_rec.place(x=1050,y=200)


        #x_data, y_data = [], []
        # Create buttons for starting and stopping data saving
        start_button = ctk.CTkButton(self, text="Start Saving Data",fg_color = "#3cb043")
        start_button.place(x=3,y=200)

        stop_button = ctk.CTkButton(self, text="Stop Saving Data", fg_color= "#bf2f24")
        stop_button.place(x=150,y=200)

        save_data = False  # Initialize the save flag

        def stop_reading_and_show_home(controller):
            stop_reading()
            controller.show_frame(StartPage)

        def start_saving_data():
            self.save_data = True
```

```
def stop_saving_data():
    self.save_data = False
    if len(self.x_data) >= 300:  # Save data only if there are at least 1000 points
        save_to_csv()  # Save data to CSV when stop is pressed

def save_to_csv():
    def find_folder_path(folder_name):
        for root, dirs, files in os.walk(os.path.dirname(os.path.abspath(__file__))):  # Starting from GUI directory
            if folder_name in dirs:
                return os.path.join(root, folder_name)

    folder_name_to_find = "Digital Stethoscope Data" #Replace with folder name
    folder_path = find_folder_path(folder_name_to_find)

    file_name = "Audio.csv"
    file_path = os.path.join(folder_path, file_name)

    print("Saving to CSV...")

    with open(file_path, "a") as file:
        if os.path.getsize(file_path) == 0:
            file.write("Time, ECG VAL\n")

        start_index = max(0, len(self.x_data) - 1000)

        for i in range(start_index, len(self.x_data)):
            file.write(f"{self.x_data[i]}, {self.y_data[i]}\n")

        print(f"File saved to: {os.path.abspath(file_name)}")

# Configure button commands
start_button.configure(command=start_saving_data)
stop_button.configure(command=stop_saving_data)

# Define global variables
exit_flag = False

ser = None  # Initialize serial connection object

def start_reading():
  self.exit_flag = False  # Access exit_flag using self

  ser_thread = threading.Thread(target=read_serial_data)
  ser_thread.start()

# Function to stop reading serial data
def stop_reading():
    self.exit_flag = True  # Access exit_flag using self
```

```
        # Create buttons for starting and stopping reading
        start_reading_button = ctk.CTkButton(self, text="Start Reading Data", fg_color="#3cb043",
command=start_reading)
        start_reading_button.place(x=3,y=150)

        stop_reading_button = ctk.CTkButton(self, text="Stop Reading Data", fg_color="#bf2f24",
command=stop_reading)
        stop_reading_button.place(x=150,y=150)
        #function to receive the ECG data
        def update_audio_label(audio_value):
         Audio_label.configure(text=f"Audio: {audio_value}")  # Update label text with ECG value
       # Inside the read_serial_data function
        def read_serial_data():
         global exit_flag, ser, x_data, y_data, data_counter
         ser = None  # Initialize ser here
         data_buffer_audio = []  # Buffer to accumulate ECG data points
         batch_size = 100  # Set the size for batch processing
         start_time = time.time()  # Variable to store the start time

         while not self.exit_flag:  # Access exit_flag directly
          try:
            # Open serial connection if not open
            if ser is None or not ser.is_open:
                selected_port = port_var.get()
                selected_baud = baud_var.get()
                ser = serial.Serial(selected_port, selected_baud, timeout=1)
                print(f"Connected to {selected_port} at {selected_baud} baud")

                # Read incoming serial data
            if ser.in_waiting > 0:
                    data = ser.readline().decode('utf-8').strip()
                    #print(data)
                    # Check if data contains audio values
                    if data:
                     try:
                        # Parse audio value (assuming one value per line)
                        audio_value = int(data)
                        current_time = time.time() - start_time
                        data_buffer_audio.append((current_time, audio_value))  # Append audio data to the buffer
                        # Update label with the received value
                        update_audio_label(audio_value)
                     except ValueError:
                        # Handle non-numeric data gracefully
                        print("Skipping non-numeric data:", data)
                    # Process batched data when buffer size reaches the specified batch size
                    if len(data_buffer_audio) >= batch_size:
                        process_batch_data(data_buffer_audio)  # Process the audio batch data
                        data_buffer_audio = []  # Clear the audio buffer after processing
```

```
            # Handle other messages or initialization here
        except serial.SerialException as e:
            print(f"Serial Exception: {e}")
            if ser is not None:
                ser.close()
            time.sleep(2)

# Inside the process_batch_data function
      def process_batch_data(data_buffer_audio):
        global x_data, y_data
# Update x_data and y_data using NumPy array concatenation
        self.x_data = np.concatenate([self.x_data, np.array(data_buffer_audio)[:, 0]])
        self.y_data = np.concatenate([self.y_data, np.array(data_buffer_audio)[:, 1]])

    # Plot ECG values or perform further processing as needed

    # Display the last 1000 points for ECG values
        max_points = 20000
        if len(self.x_data) > max_points:
         self.x_data = self.x_data[-max_points:]
         self.y_data = self.y_data[-max_points:]

          # Plot audio values using NumPy arrays
        self.line.set_data(self.x_data, self.y_data)

    # Plot audio values
        self.line.set_data(self.x_data, self.y_data)
        self.line.set_color('blue')  # Change the color of the audio values line to blue
        self.ax.relim()
        self.ax.autoscale_view()
        canvas.draw_idle()

        self.data_counter += len(data_buffer_audio)  # Increment data counter

        status_label.configure(text="audio in Operation...")

    # Update the legend with ECG values
        self.ax.legend(['Audio amp'], loc='upper left', bbox_to_anchor=(1, 1))

        # Start reading the serial data in a separate thread
      import threading
      thread = threading.Thread(target=read_serial_data)
      thread.start()


    def update_switch_text(self):
          if self.switch_var.get() == 0:
            self.switch_text_var.set("Audio ON")
```

```
        else:
            self.switch_text_var.set("MUTE")

    def get_switch_state(self):
        return self.switch_var.get()

# Driver Code
app = tkinterApp()
app.mainloop()
```

6.0 REQUIREMENT TRACEABILITY

*the highlighted ones are affected by CSCI components

System Requirement
4.2.1.1: Temperature Sensor Range
4.2.1.2: Temperature Sensor Accuracy
4.2.1.3: Stethoscope Sensitivity
4.2.1.4: Stethoscope Audio
4.2.1.5: Pulse Oximeter Signal Noise
4.2.1.6: Pulse Oximeter Signal Count
4.2.1.7: Pulse Oximeter Heart Rate
4.2.1.8: Skin Impedance Range
4.2.1.9: Skin Impedance Accuracy
4.2.1.10: Electrocardiogram Accuracy
4.2.2.1: IP Rating
4.2.2.2: Operational Temperature (D) System operates from 10 to 40 degrees Celsius
4.2.2.3: Operational Humidity (D) System operates under 50% humidity
4.2.3.1: Weight (A/I) System no more than 15 pounds
4.2.3.2: Dimensions (A/I) System 35.5 by 25.4 by 12.7 cm
4.2.3.3: Inexpensive (A) System costs less than $300
4.2.3.4: Saved Data
4.2.3.5: Probe Chemical Resistance
4.2.4.1: Electrical Safety (I) System does not have exposed electrical components
4.2.4.2: Privacy Protection
4.2.4.3: Maximum Bioimpedance Current
4.2.4.4: Maximum ECG Current
4.2.4.5: Medical Device Standard (I) System meets IEC 60601-1 standards
4.2.4.6: Chemical Safety (D) System meets ACGIH TLV-SL guidelines
4.2.5.1: Internal Power Supply

4.2.5.2: Computer Communication
4.2.5.3: GUI
4.2.5.4: External Power Supply (A) System functions with 15 amp, 120 V AC power

7.0 NOTES

As of April 30, 2024, the software design document for the MediBrick2000 is complete.