PTSD Detection Device

FINAL REPORT DOCUMENT

Team #15
BAE Systems / America's VetDogs
Rachel Shannon (Faculty)
Casey Halbmaier, Caden Backen, Coby Konkel, Ben Gardner,
Nihaal Zaheer, Andres Ceballos
sdmay24-15@iastate.edu
sdmay24-15.sd.ece.iastate.edu

Revised: Version #3

1 Introduction / Background

1.1 Team Members

1) Casey Halbmaier	**Software Engineer**
2) Caden Backen	**Software Engineer**
3) Coby Konkol	**Software Engineer**
4) Ben Gardner	**Electrical Engineer**
5) Andres Ceballos	**Electrical Engineer**
6) Nihaal Koyakunju Zaheer	**Computer Engineer**

1.2 Problem Statement

If a discrete and wearable device existed that is capable of detecting PTSD episodes, support for veterans with PTSD would become much cheaper, accessible, and faster than it currently is. Veterans could either wear the device or have an early warning system for PTSD episodes. The device can also help a service animal learn when to act. These devices are also good tools for trainers to train service animals.

American VetDogs proposed a potential solution to this problem. We are challenged with designing and building a prototype for a device that monitors the veteran's physiological data, detects PTSD symptoms in advance, and alerts a service animal that an episode is imminent.

1.3 Intended Uses / Users

Use cases:

- Veterans and non-veterans with a history of PTSD episodes need to be able to prevent these episodes before they happen. (Currently, veterans are the primary testing group for this project.)
- Service dog trainers need to be able to determine when a PTSD episode is happening so they can train service animals more effectively.

Who benefits from the project:

People with PTSD episodes benefit from the prevention of PTSD episodes before they happen, reducing their stress and pain and thus improving their quality of life.

The families of people with PTSD indirectly benefit from the improvement of their family member's quality of life. Seeing a loved one routinely suffer and relive unimaginable trauma significantly affects the psyche of everyone witnessing the episode.

Service dogs of people with PTSD episodes benefit from the alert of a PTSD episode, which makes it easier for them to know when they need to assist their owner.

1.4 Work in Context

There have been significant advances in wearable technology in recent years; Sports teams are constructing wearable devices to monitor performance and health, corporations are selling fitness bands to help track fitness progress, and wearable devices are regularly assembled to aid in academic research. Advances in ambulatory measurements of electrodermal activity, wrist-worn electrocardiograms, and many other devices have been constructed for specific purposes.

Our work aids in the introduction of wearables in researching episode-based disorders (ours is focused explicitly on PTSD). It can help capture data points for academic research for identifying physiological changes during PTSD episodes, responding to changes, and overall, allowing research in a field that is very difficult to begin performing research on.

The first and arguably most prominent challenge we faced in our project was a lack of available research data on PTSD episodes. Given data, a project could be done to predict PTSD episodes in advance or gain an understanding of critical factors to observe. Introducing a microcontroller-based wearable with built-in Wifi and Bluetooth capabilities and local data storage permits future teams to evolve how data can be collected, add more sensors, and develop the device over time.

2 Revised Design

2.1 Requirements

Functional requirements:

- Reliably monitors the user's physiological data.
- The accelerometer provides information about the user's movement.
- Detect any abnormal behavior (spikes) in blood pressure/heart rate consistent with a PTSD episode.
- Communicate with the device on the service animal that a PTSD episode is imminent.
- The user should be able to dismiss the device before the dog is notified.
- The device on the service dog should alert them that a PTSD episode is imminent.
- The user should be able to power off and on the device.

Qualitative aesthetics:

- It must be discrete. It was emphasized in the project proposal that the device should not disturb anybody in the vicinity.
- Dog notification is quiet (or at least subtle/non-disruptive) **(constraint)**
- The device worn by the user should be comfortable and non-invasive

Economic/market requirements:

- We are given a budget of $5000 for designing a prototype **(constraint)**
- The end product should be cheaper than smart watches and other biological monitoring devices currently available.
- Our end-users are commonly retired and may have a fixed income. This may limit their ability to purchase expensive products. Cheaper manufacturing techniques will be a consideration after our initial design.

UI requirements:

- The interface with the user should be accessible for various disabilities. We must take into account motor function, vision, hearing, and any other disabilities we discover common among our primary users, veterans.
- The interface should be simple to use so notifications can be easily dismissed before alerting the service animal. Our primary clients are also commonly members of an older generation, less skilled at using technology.

Performance requirements:

Algorithm performance: The PTSD detection device should be more tolerant of false positives than true negatives (we would rather have the device alert the service animal when there are no PTSD symptoms than not provide a notification when there is an episode). We will look at the False positive rate and actual negative rate (portion of episodes missed).

- The device must have a minimal number of missed PTSD episodes. Upon further study, we should determine a goal metric for this requirement. This means we have a low tolerance for the actual negative rate. We should shoot for a rate of under 0.2 (20%).
- We should have a reasonably low number of false-positive notifications for the dog. The dog being alerted that they need to respond is a low-damage situation, while the dog not being alerted during a PTSD episode is potentially embarrassing to the veteran and disruptive to the problem. This can be measured with the false-positive rate.

Combined algorithm performance::

These first two performance requirements can be quantified using the area under the curve (AUC) of the receiving operating characteristic (ROC) graph. An AUC value of over 0.65 would assert that most positive outputs are correct.

Durability: The device should be durable since it is intended to be worn everywhere the user goes.

- Should be drop/impact resistant
- Should be water-resistant
- The battery/device for the monitoring device should last at least 24 hours.

Legal and Ethical requirements:

- We are storing, processing, and moving medical vitals from our clients. We would need to follow local laws for computer-based records to allow this prototype to be on the market. These laws vary by state and country. This legal consideration of medical information is outside the scope of our prototype, and we will **not** be putting significant effort into following these standards and requirements.
- The materials used in the feedback device and the device worn by the user should be made of materials with no potential to harm the users.
- The feedback device should use an ethical mode of communication to notify the service animal

Maintainability requirements:

- We want the devices to be modular so that each component can be easily updated and replaced
- Backward compatible, so updating different components doesn't require a replacement of the entire system.

Security Requirements:

- End-to-end encryption of vitals
- In-place encryption of stored vitals and information (Not necessary for prototype).
- Authentication/identity verification for accessing information
    - (Not necessary for the prototype.)

Testing requirements:

- The system should have a minimal false positivity rate.
- The device should be tested with our partner company, VetDogs, to ensure the alerts' subtleness and the devices' comfortability for both the user and the dog.

2.2 Engineering Standards

- IEEE 802.15.1: WPAN / Bluetooth - We plan on using Bluetooth to connect separate devices (phone and wearable) in our design.

- IEEE 802.11: WiFi - Future iterations may use WiFi.

- ISO/IEEE 11073: Medical / Health Device Communication Standards - We will be designing/using a wearable device that collects health data and communicates it to another device.

- IEEE 360-2022: IEEE Standard for Wearable Consumer Electronic Devices - We plan to design a wearable device to collect heart rate/blood pressure.

- IEEE 370-2020: IEEE Standard for Electrical Characterization of Printed Circuit Board and Related Interconnects at Frequencies up to 50 GHz - We will be doing some PCB design and testing

- Revised 508 Standards and 255 Guidelines https://www.access-board.gov/ict/ - accessibility standards are used on government and official user interfaces to ensure access for people with physical, sensory, or cognitive disabilities. Section 255 covers telecommunications and customer-premises equipment, While 508 covers information and communication technology.

- Web Content Accessibility Guidelines https://www.w3.org/WAI/standards-guidelines/wcag/ WCAG documents explain how to make web content accessible to people with disabilities. If we design any interfaces on screens, this standard can ensure usability for our clients.

2.3 Security Concerns and Countermeasures

While we did not pursue the mobile app portion of our original design, we have given some thought to the security concerns it would present. Due to the nature of phone applications, security testing is essential so as to not allow any breaches in user data. As such, the tests we would put into place are as follows:

- Testing the encryption of data transmission via our communication protocols (Bluetooth).
- Ensuring that the firmware for the microcontroller and sensors are up to date.

- End-to-end encryption of vitals can be ensured via digital signatures. Man-in-the-middle attacks can test this.

2.4 Evolution Since CPRE/SE/CYBE/EE 491

Due to time constraints, we shifted our focus away from an additional phone application. This application was planned to allow users to utilize and visualize their own data. This, unfortunately, was deemed out of scope for the time frame we had available this semester. Still, we would love to see this implemented if future senior design teams choose to continue our project, as we discuss in the plans section.

On the hardware side, we designed and tested a PCB similar to our vision during 491, with everything on one board. After testing, we had multiple errors that warranted some redesign. Among these were some that we could not quickly isolate with the one-board design. We split the PCB design into two boards to isolate circuits and make testing and isolating issues easier. We also added a ground plane, which we had omitted before, which solved a signal integrity issue and allowed us to follow better practices with our trace layout. The split boards did make our testing significantly easier. The idea is still that an ideal final consumer design would utilize one PCB for a smaller overall consumer device for better ergonomics and aesthetics - the device needs to be something that someone can wear all day.

Another evolution is our decision to upgrade our sensor. We decided to utilize the MAX30101 sensor over our MAX86150 sensor from the original design. Our decision to upgrade from the 86150 to the 30101 was due to the 30101 having more LEDs for sensing and because the 86150 had extraneous ECG sensing that we had no use for and added extra complexity through extra pins and functions.

2.5 Implementation Details

2.5.0 Sensor Choice

We attempt to obtain sensor readings for cardiovascular and respiratory vitals in our requirements. For these metrics, we identified Photoplethysmography (PPG), saturation of peripheral oxygen (SPO2), and electrodermal-activity (EDA) readings as potentially useful for PTSD episode prediction.

PPG is a measurement of how blood vessels reflect various frequencies of light. As a person's heart beats, blood vessels expand and contract, changing how much light skin reflects by a small amount. This can measure heart rate, blood pressure, blood volume/flow, and other metrics.

SPO2 is a measurement of blood oxygen content. As oxygen saturation changes, the red/IR light reflected changes proportionally. More oxygenated blood reflects more IR light, allowing more red light to pass through. Using this information, the amount of red/IR light reflected can be used to calculate SPO2 amounts.

We identified the MAX30101 sensor as a candidate for PPG and SPO2 measurements. The sensor is a small IC with red, green, and IR LEDs and a photodiode (to measure reflected light). The sensor is commonly used in devices to measure heart/respiratory activity on wrists. We decided in our first iteration to use this device as a starting point for getting vitals.

We did not include an EDA sensor in our initial design, saving it for a later rework since that type of measurement would prove complex and we wanted to establish a good framework to build from first.

2.5.1 Detailed Design

At a high level, our design is split into hardware and software. The hardware consists of two PCB designs and a design for how the wireless communication and dog device will work. The software design has modules for different functions the device needs.

Hardware:

The two PCB designs are modular, with one housing the battery and generating the different voltages that we need (Figure 2.5.1.1) and the other containing the ESP32-Pico microcontroller, MAX30101 sensor, and external connector pins for power and our microSD card breakout board (Figure 2.5.1.2). We call these the "power pcb" and "main pcb" respectively.

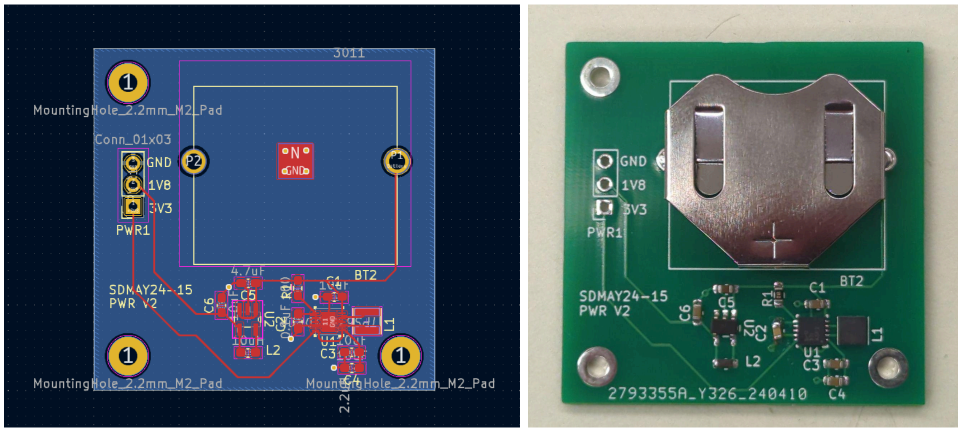

Figure 2.5.1.1 - Power PCB

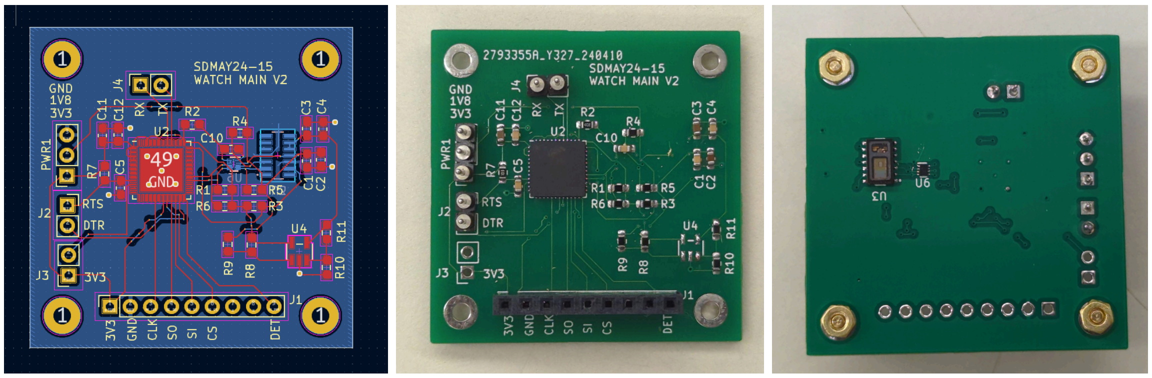

Figure 2.5.1.2 - Main PCB

The power PCB holds a cr2032 watch battery and converts its voltage output to stable 3.3V and 1.8V rails using a buck-boost converter and a buck converter, which can then be accessed through connection pins.

The main PCB is the "brain" of the device. It has pins for receiving power, transmitting and receiving data, and connecting a microSD card breakout board. Both PCBs are designed in two layers with a ground plane, which was not present in previous interactions of the design, in order to help with signal integrity and heat dissipation. The MAX30101 sensor is on the back of the main PCB so that it can contact the user's skin.

We created a simple one-way RF communication circuit for communication between the wearable and the dog device. We aimed to keep the design simple with a transmitter, receiver, and vibration motor. The transmitter will be integrated with the PCB's 3.3 V power rail that will send a signal to the receiver circuit on a breadboard, making a motor vibrate. Our prototype circuit is shown in Figure 2.5.1.3.

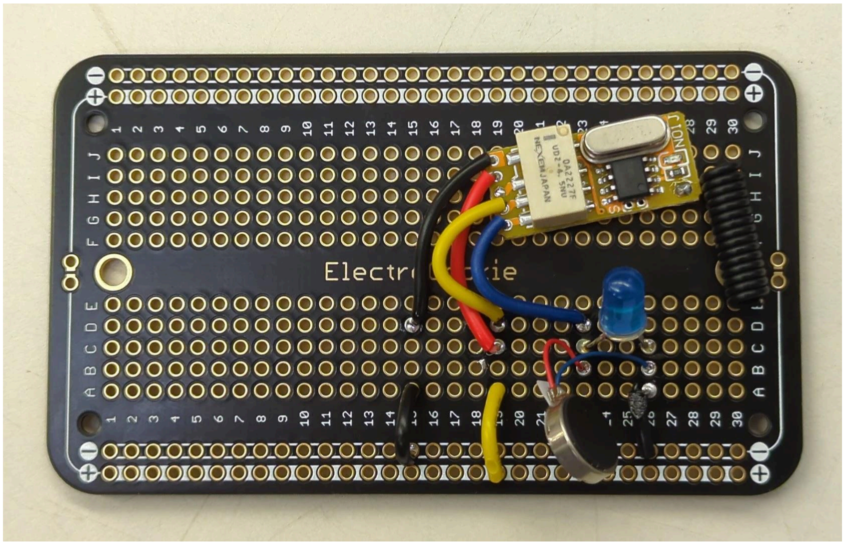
Figure 2.5.1.3 - Feedback Device

Software:

The software runs on the ESP32-Pico-D4 microcontroller using ESP-IDF as a framework/toolchain for building and flashing. The software is organized in a layered-modular architecture with various peripheral components isolated as standalone "components." This architecture is typical among low-level code as it allows hardware to be replaced, added, and removed without causing significant problems.

The software functions by polling and using callbacks for button events. When the button is pressed once, the program begins writing to the microSD Card. The microSD card is formatted as a FAT32 file system and mounted with software where we can store data. When the button is held for a few seconds, the program begins to send pings to the radio receiver on the feedback device.

Adding new hardware involves either importing the firmware or developing custom firmware, then using the firmware within the "main" component. Figure 2.5.1.4 shows an overview of the software implementation.

ESP IDF has a built-in tool called "menuconfig" based on KConfig files for configuring various variables before compilation. We can use this to test various ESP32 development board modules and quickly change the configuration for different build configurations. This requires the program to be rebuilt before it can be flashed onto the ESP32 microcontroller.

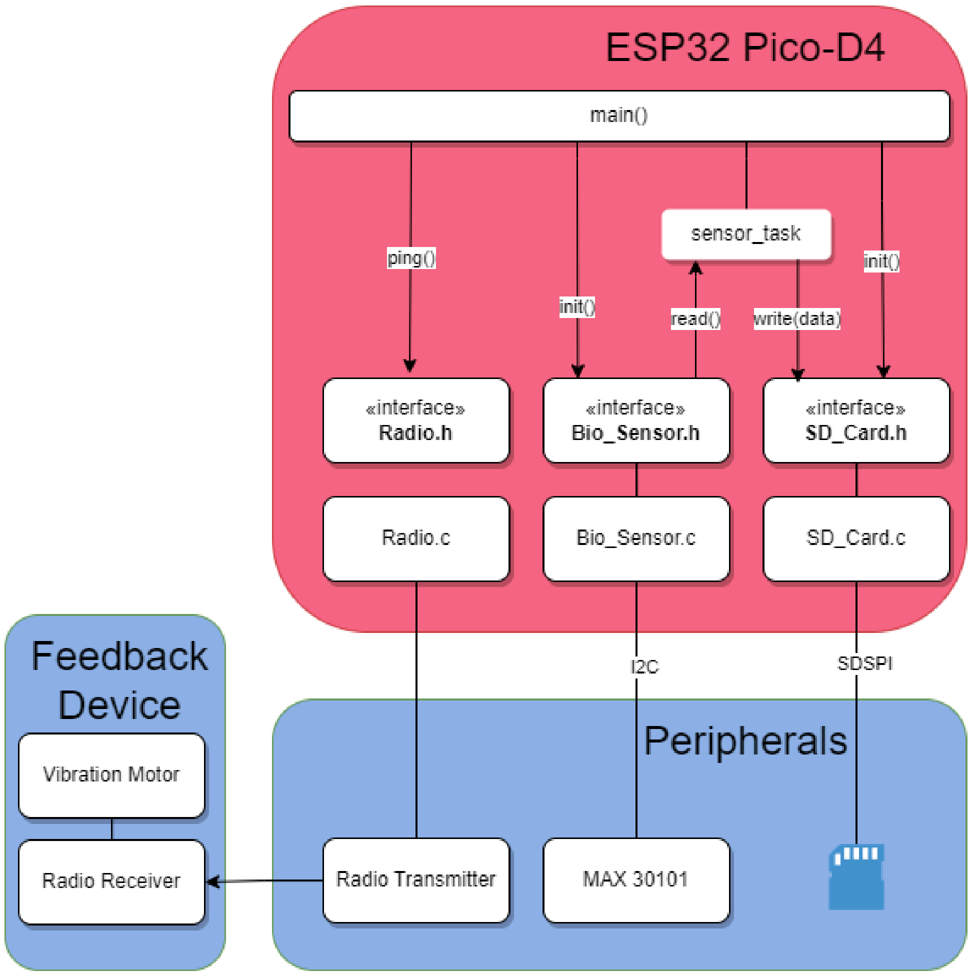

Figure 2.5.1.4 - Software Implementation Diagram

2.6 Testing

2.6.1 Process

Hardware Testing Process:

We had two hardware releases this semester. A portion of that testing was the same for both. Our hardware testing mostly consisted of verifying that our circuits worked as expected. For both, the first step was running rule checks on our schematics and PCB design in the design software. For these checks, we used the default design rules. We made sure that our designs passed these checks before ordering them. After assembling the hardware, we tested it to ensure it was operating as expected. This consisted of measuring points on the PCB with a voltmeter in the lab and trying to flash the microcontroller using an off-the-shelf programmer. ETG, an Iowa State University internal technology resource group, assisted in inspection of our team's PCBs. BAE Systems also assisted in reviewing our PCB schematics and providing feedback. When testing the first hardware release, we noticed that the power circuits were not functioning as expected, so we had to bypass those and provide power to the rest of the PCB from the bench to continue testing.

To test our RF communication circuit, we first laid out our circuit with a transmitter, receiver, vibration motor, and an LED into a breadboard. The LED was to aid in identifying whether the circuit was working. We used a bench power supply to simulate a high and low signal to verify whether the circuit operated as expected, with the transmitter transmitting when receiving a high input signal and the receiver receiving the signal properly and activating the motor and LED.

Software Testing Process:

Software testing consists of two stages for our project: building/compiling focused testing, followed by physical testing on the team's devices. For building, we utilized Docker to simplify the build process. Using an image and the same command for running the program allowed each team member to have the same build process regardless of their personal IDE or operating system. Using the toolchain for the software platform we chose (ESP IDF), we can connect a terminal to standard output, and use normal GDB tools for debugging code. We used a modular approach keeping each team member's code decoupled so testing can be done for one sensor or part at a time. Finally, we constructed a CI pipeline on Gitlab to ensure code builds before introducing any code to our main branch. This helps us always maintain a working version of the code.

To simplify the process of testing software, we used the ESP32-Wrover-E microcontroller provided to each team member by BAE Systems, as well as breakout boards for various components. The toolchain was configured to detect the type of device being used. We connected our sensors directly to the development board and, after flashing to the microcontroller, were able to run our code. When we flash code onto the PCB, the ESP IDF automatically uses the same build process and transfers the code through a serial USB/UART adapter. This makes it so that from a software standpoint, nothing changes from building/flashing on the dev kits from our actual target device. We also implemented some feature flags for specific components inside our software using the KConfig settings. Using this, we enabled/disabled test code and isolated specific components in the software for testing.

Our software used two main components: the MAX30101 sensor and the microSD card reader. For storage testing, we attached the microSD Card reader to our microcontroller, flashed our code to the microcontroller, and then ran the program. To ensure the program worked, we removed the microSD card attached to the reader and inserted it into our PC. From there, we checked the microSD card to ensure the correct file was created and the correct data was written. For our MAX30101 sensor testing, we used an open-source ESP-IDF I2C tool project (provided by Espressif on GitHub) to write and read from registers using a command-based interface. Afterward, we adapt those actions into our actual code.

Most of our testing was done by introducing print statements and connecting them to a breadboard. Building is automated and scripted using Docker, and we test with breadboards and the actual PCB created by the team. In future project evolutions, PyTest can emulate code running, as shown in the ESP IDF 'README' files and example projects.

Integration Testing Process:

Our integration testing consisted of testing our software with our different peripherals. We modified our existing code for each device to isolate its functionality. We then flashed the code to the development kit so that it would run the part of the code relevant to the test. Finally, we re-ran that test on the PCB.

When testing the microSD card, we isolated its init function and tested that the code could be written to the device. Then, we read the microSD card back to verify it was written correctly. For the transmitter, we isolated initializing and sending a signal. Then, when we flashed the device to verify that the receiver vibrated. After testing whether the microSD card and transmitter worked independently, we also tested them with a button to control them manually. For the microSD card, we tested whether we could trigger writing to the microSD card by pressing the button. For the transmitter, we tested whether we could manually control the signal by pressing the button.

2.6.2 Results

Hardware Testing Results:

In our first hardware release, the board had failures in the power circuit - it output only one of the two voltages that we needed access to, then when we bypassed the power circuit to test flashing to the microcontroller, we experienced both complete failures to communicate and signal integrity errors when we got past those. With the microcontroller not functioning, all we could verify about the sensor and microSD card circuits and connections were that things were electronically connected as desired. Our troubleshooting led us to decide that we needed to do some redesigns and make another hardware release. Precisely, we needed to introduce grounding planes in our PCB design and move some parts away from each other to help with signal integrity and make future testing easier.

In the second release, we had two PCBs rather than one. The power supply PCB worked as expected, outputting the two required voltages with a range of input voltages to simulate a battery discharging. The PCB with the microcontroller also passed initial tests with no signal integrity issues, with complete round-trip communication between the microcontroller and the connected computer.

For the RF circuit, the breadboard testing all went as expected. The transmitter and receiver in their default configuration operated as we desired, with the transmitter transmitting continuously as long as it received a high input and the receiver properly receiving the signal and turning on the motor for the duration of the signal.

Software Testing Results:

Software testing was split between the testing of two different components of our project: the microSD card reader and the MAX30101 sensor code. There were many challenges for the microSD card reader at the beginning of testing. Mainly, finding which connection pins would allow the microSD card reader to function was a challenge. In some cases, the program would work but fail to initialize the microSD card reader, while other times, the program would enter an infinite loop due to incorrect pins being set. Once the pin configuration was taken care of, the next issue was determining why the microSD card reader was still unable to initialize. The team believed the issue stemmed from the pin configuration when, in reality, there was another factor.

Most of the microSD card's code was taken from an Espressif open-source project with a 'storage' example using the ESP32-Wrover-E's built-in microSD card slot. As such, code components needed to be configured to match what our project needed, which in this case was to transfer data over SPI. By default, the program connected using 'sdmmc,' when, in our case, we needed to use SPI. Once we had made the switch, the microSD card reader initialized and produced our first 'Hello World' text file.

The final stretch in testing was to remove any 'unneeded' components of the code. Since this was an open-source project we were modifying, several components were unnecessary for our project to function. Many of these components were functions for different data transfer methods and microSD card formatting options. Once the clutter was resolved, the team created new functions that could be called from our 'main. c' file. The original Espressif open-source code had all its functions within the primary function. It was initializing the microSD card reader, creating the file, reading it, writing to it, and then closing the connection to the reader in one function. Splitting these functions proved to be easier than anticipated. It became an initialization function, a write function, a read function, an extra function to format any incoming data that needs to be written to the microSD card to a global variable for ease of use, and a deactivation function to close the connection between the microSD card reader and the ESP32-Pico microcontroller. The program was flashed to the ESP32-Pico microcontroller after any changes were made to ensure the program was still functional. Occasionally, an infinite loop would be created due to improper string handling or pointer use, but these errors were quickly stamped out when they occurred.

The final results for the microSD card reader gave us functions that can initialize the microSD card reader, read from the file created on the microSD card, write to a file on the microSD card, format a string of data to prepare to be written onto the microSD card and terminate the bus connection between the microSD card reader and the ESP32-Pico. For this final prototype, the program can only initialize one file at a time and delete any previous files already created with the same name. This means that any previous data is wiped whenever the program initializes the microSD card reader.

Integration Testing Results

All integration testing on the breadboard with the development kit worked mostly expected. We were able to write to the microSD card and transmit a signal both with and without the button control. We did have to implement a simple transistor circuit as the logic level of our microcontroller was too low to control the transmitter directly. However, this broke down when we tried the same setup using our PCB. The only part that we were able to get working there was writing to the microSD card without button control. We identified that we had multiple design flaws in the button circuit on the PCB that prevented it from working and the microcontroller from detecting a button press. Due to this, any code that requires manual input from the user cannot run properly. For the transmitter, our PCB cannot provide enough power to the pins that it's connected to in order for it to function. This is not unexpected, as we were not designing around connecting the transmitter on this version of the board and were just trying to see if we could make it work with what we currently have.

2.7 Broader Context

Area	**Description**
Public health, safety, and welfare	The wearable cannot be harmful to the user. The dog's wearable device should not disrupt the dog or anybody nearby. The veteran's quality of life would be drastically improved if PTSD episodes could be preemptively handled.
Global, cultural, and social	We have chosen to use a vibration motor to signal the dog as opposed to something like a shock or a noise, as those are not considered very ethical options in our culture and the culture that will use it. Our device, upon success, will add to the modern research and development in utilizing wearable devices for continuous improvement of users' daily lives by aiding in the prevention of potentially harmful/painful situations and uplifting members of the community in need of assistance for improving their daily quality of life.
Environmental	The final product will likely be made from plastic. Ideally, this would be utilizing recycled material.
Economic	Our product should be relatively affordable. We plan to use relatively cheap sensors and other components. It needs to be affordable because our main audience, veterans, generally are not very wealthy and the device is meeting a medical need. Also, a charity

	may fund these devices for some people, so we would like to not put undue financial strain on them.

2.8 Conclusions

2.8.1 Progress Review

We finished the semester partially finished with the forest iteration of our project. We successfully flashed code onto a microcontroller on a printed circuit board and designed a power supply. On this PCB, you could write a file onto a MicroSD card and utilize a transmitter to send a signal to a feedback device. We hoped to get much further along than this in the process but had unexpected roadblocks halt progress on several fronts.

Our current design should prove a solid foundation for further development iterations for the device. A baseline hardware device is established, and software infrastructure is set up for building and flashing to a device.

2.8.2 Value Provided to Users

This device serves as a foundation for future development by teams established by American Vetdogs. Vetdogs should be able to involve electrical, software, and computer engineers in developing a suitable device to collect data, learn about PTSD, and further research. The other wearable biometric devices that could measure this data that we found were, if even usable outside of a clinical setting, expensive, with limited sensing capabilities set by whoever manufactures the device, and had much hidden proprietary information on how they work. Using an in-house designed device allows America's VetDogs to dictate how the project will continue without being limited by the capabilities of expensive hardware or commercial smart watches but by the creative minds of the students they engage with. For example, students could interface with the integrated Bluetooth capabilities to communicate with an app or the cloud, expand the PCBs to communicate with other types of sensors, revise the hardware to have a smaller form factor, and develop algorithms to predict PTSD in advance. The possibilities are boundless, and the power to continue this work is in America's VetDogs' hands.

2.8.3 Potential Future Steps

We have already discussed a succession plan for our project with our end client, America's VetDogs. They would like us to provide them with all hardware and software so they can have a future senior design group start from where we left off. BAE Systems has agreed to help with continuity within the ISU SE/EE/CPRE/CYB senior design system.

In addition to passing it off, we are also aware of multiple changes that need to be made to our design to improve it. We have identified a better sensor for our application that is an almost perfect drop-in replacement. It needs access to a 5V power rail, which requires adding another voltage regulation circuit on the power board and a connection pin. The PCBs and hardware assembly also need to be made smaller and possibly re-combined into one board for practical use by an end user.

On the software side, improvements can be made to the microSD card reader and the MAX30101 sensor programs. The code can be modified for the microSD card reader to allow for more than one file to exist at a time on the microSD card. This was originally planned to be a feature but fell through due to time constraints and programming issues. The MAX30101 Sensor can be corrected to properly function and output the biological data required to determine when a PTSD episode is occurring.

1 Appendix A - Operation Manual

To attach radio transmitter, hook up a N channel MOSFET source to Rx, gate to Tx, drain to the negative end of the transmitter connector, and the positive to a 5V source.

Readmes copied from Git Repo:

Wearable device software project

To Run the Project:

In GitBash:

1. **Start esp rfc2217 server:** `./esp_rfc2217_server.exe -p 4000 COM13`
2. **Source aliases file** `source toolchain-aliases.sh`
3. **Build, flash, or monitor code:**
- **To build: dtc-build**
- **To flash: dtc-flash**
- **To monitor: dtc-monitor**
- **Other commands: dtc (idf.py options)**

****To run any other idf.py stuff run ****

Troubleshooting

Device or serial port busy: Try to kill all esp-idf running containers from Docker Desktop.

To Program Device using Programmer (CP2102N):

1. **Connect RTS, DTR, TX, and RX pins on the programmer to PCB.**
2. **Connect 3v3 on PCB to 3v on programmer and gnd on programmer to gnd on PCB**
3. **Connect (cp2102N) device to breadboard, plug into computer via. USB-C cable.**
4. **Identify which serial port (COMX) the programmer attaches to (use device manager for Windows)**
5. **Start esp rfc tool with `./esp_rfc2217_server.exe -p 4000**
6. **Open terminal/bash window in software cmake project directory. Source the toolchain tools with** `source toolchain-aliases.sh`
7. **Run commands for project according to aliases**

Hardware team workflow:

Only work in a hardware folder

Only First time accessing the project, Clone

- Pull (Copy changes from remote repository directly into working directory)
- Make changes ()
- Stage changes
- commit changes
- Push changes

git fetch only copies changes into local repo

Software team:

Only work in a software folder

Submodules:

After Cloning project run `git submodule init` to fetch all data for esp-idf project.

When current dir is in the submodule, all of the normal git commands/actions work as normal, but for that specific project's repo.

For more details on submodules, check here

Workflow

First time accessing project: Before running any code, you need to Setup the ESP toolchain.

ESP-IDF is cloned as a submodule in this project.

- Clone project
- Initialize submodules: `git submodule init`
- Then run `git submodule update --init --recursive`
- Create environment variable `IDF_PATH` as path to esp-idf folder in this project.

Starting work:

- Check out master
- pull
- Create a branch
- push and set upstream. (`push --set-upstream origin <BRANCH>`) Each time doing some work:
- Checkout the branch
- pull
- Make some changes

- stage changes
- commit changes
- push

After potentially done with work:

- In gitlab create merge request
- Get at least 1 approval
- merge or revise and repeat. For reviewing:
- Fetch and checkout branch
- Test it yourself on your local
- Approve or request changes.

2 Appendix B - Alternative/Initial Version of Design

As alluded to in the report, we had some pretty lofty goals for our initial design. That design is discussed in detail in our fall semester final design document, which is also linked on our team website.

As shown in Figures B.1 through B.3, our original design was split into three design and release stages, with features incrementally added in each stage. The first design stage would solely collect data, which could then be used to develop an algorithm. Stage two would integrate the dog feedback device. Stage three would add a Bluetooth connection and a mobile app.

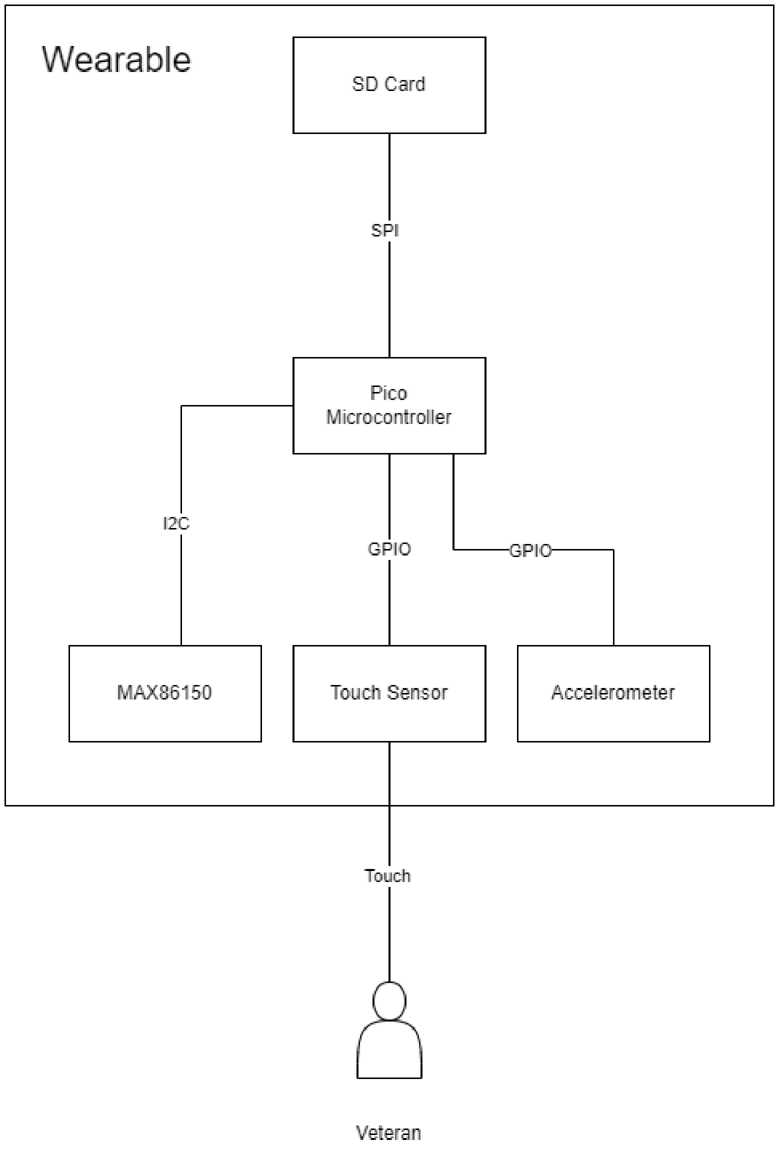

Figure B.1 - Design Stage 1: Wearable for data collection

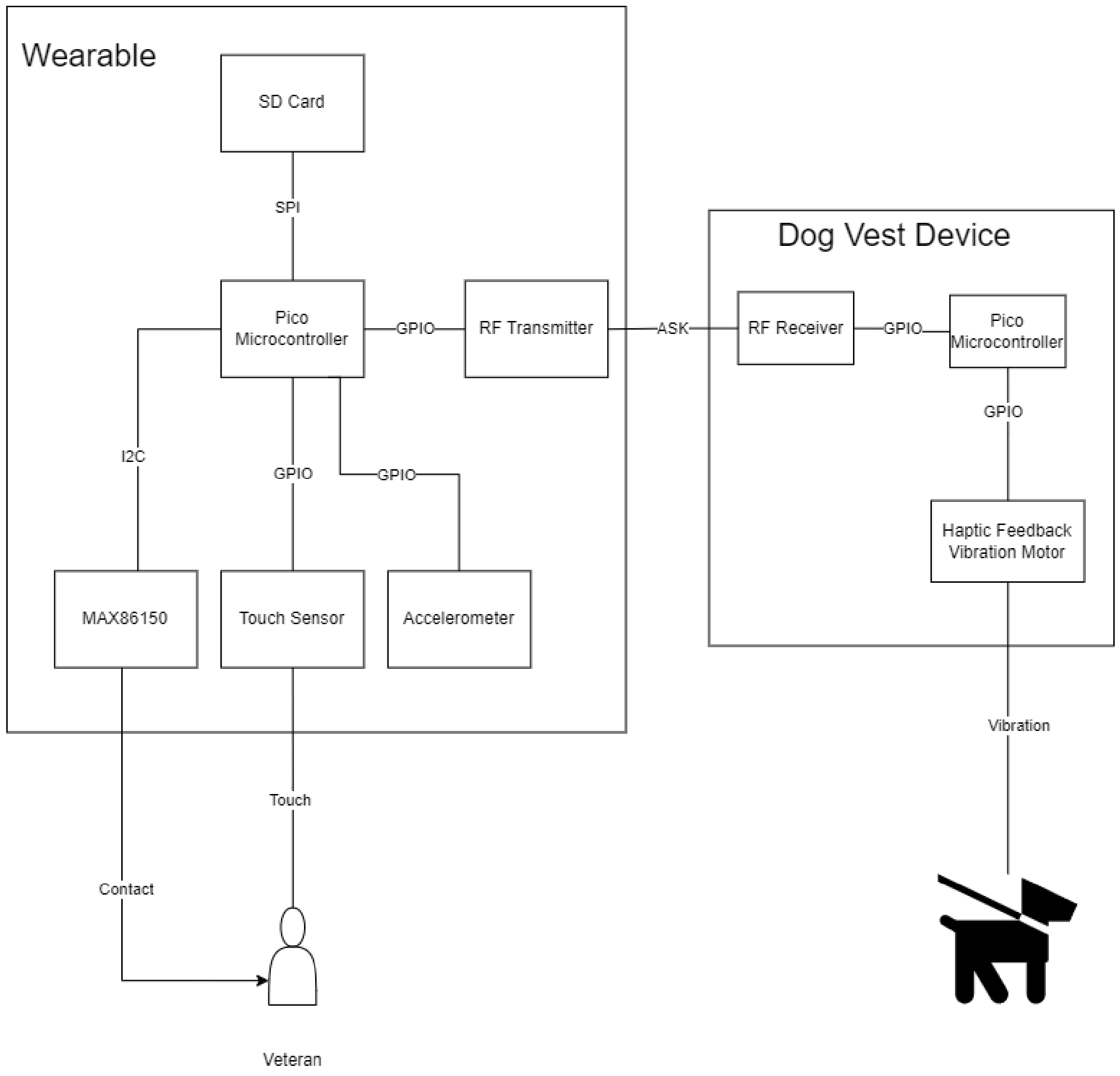

Figure B.2 - Design Stage 2: Wearable with dog feedback device

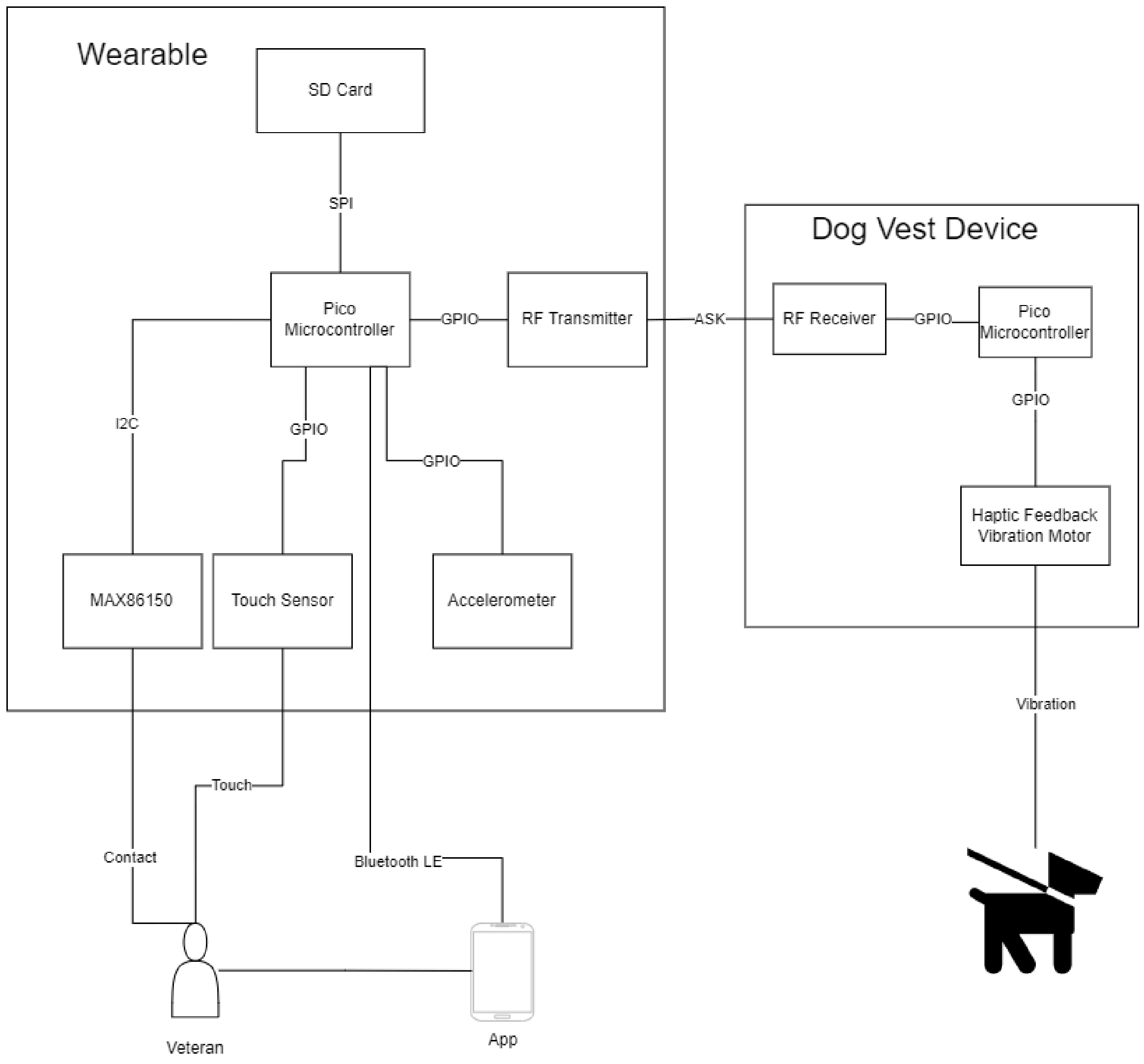

Figure B.3 - Design Stage 3: Integration of app with wearable

3 Appendix C - Other Considerations

Lessons Learned:

- Time commitments are important for collaborative projects. Due to team members having other classes and time commitments, it was difficult to find a schedule to meet and work on the project.
- Procrastination can significantly impact a team's final prototype. Features for our prototype, such as the MAX30101 Sensor and SD Card reader, were put off towards the end of our project timeline which lead to these features lacking in the deliverables we promised.

4 Appendix D - Source Code

main.c:

```c
#include "includes/main.h"
#if CONFIG_WEARABLE_RADIO_EN
#include "Radio.h"
#endif
#include "SD_Card.h"
#include "button.h"

//static const char *TAG = "main.c";
_Noreturn void main_task(void *pvParameters)
{
   while(1) {

   }
}

void app_main(void)
{
   char sensor_data[64];
   strcpy(sensor_data, "Hello World!");

   sd_init_device();
   button_init();

#if CONFIG_WEARABLE_RADIO_EN
   radio_init();
#endif
#if CONFIG_WEARABLE_RADIO_TEST
   ESP_LOGI(TAG, "Testing Radio");
   radio_ping(3);
#endif

   while(1) {
```

```
            if(press == 1) {
                sd_data_config(sensor_data);
                sd_write_file(file_path, data_to_write);
            }
        }

        sd_deact_device();

    }
```

main.h:

```
//
// Created by Cobeasta on 2/28/2024.
//

#ifndef WEARABLE_PTSD_DETECTION_MAIN_H
#define WEARABLE_PTSD_DETECTION_MAIN_H

#include <stdio.h>
#include <inttypes.h>
#include <string.h>
#include <sys/unistd.h>
#include <sys/stat.h>
#include "driver/gpio.h"
#include "freertos/FreeRTOS.h"
#include "esp_log.h"
// STRUCT DEFINITIONS FOR DATA EVENTS TO TRACK

typedef struct event_bio_data {
   long time; // time since chip startup
   long red_count;
   long ir_count;
   long green_count;
}max_sens_event_t;
typedef struct event_butt_press {
   long time; // time since chip startup
   long red_count;
   long ir_count;
   long green_count;
}event_butt_press_t;

#endif //WEARABLE_PTSD_DETECTION_MAIN_H
```

SD_Card.c:

```
//
// Created by Casey Halbmaier (crhalby) on February 7th, 2024
//
```

```
      // This code is heavily inspired by an Espressif example, found on:
      //
https://github.com/espressif/esp-idf/blob/master/examples/storage/sd_card/sdspi
/main/sd_card_example_main.c
      //
      // The code has been altered to fit the needs of our project.
      //

      #include "includes/SD_Card.h"

      #define MAX_CHAR_SIZE    64

      static const char *TAG = "SD_Card.c";

      #define MOUNT_POINT "/sdcard"
      const char mount_point[] = MOUNT_POINT;

      sdmmc_host_t host = SDSPI_HOST_DEFAULT();
      sdmmc_card_t *card;

      char file_path[64];
      char data_to_write[64];

      // PIN Assignments
      const gpio_num_t PIN_NUM_MISO = 13;//CONFIG_WEARABLE_SDSPI_SI;
      const gpio_num_t PIN_NUM_MOSI = 14;//CONFIG_WEARABLE_SDSPI_SO;
      const gpio_num_t PIN_NUM_CLK  = 15;//CONFIG_WEARABLE_SDSPI_SCLK;
      const gpio_num_t PIN_NUM_CS =  2; //CONFIG_WEARABLE_SDSPI_CS;

      // Function that writes the contents of the given string input to the
specified file on the SD Card
      esp_err_t sd_write_file(const char *path, char *data)
      {
         ESP_LOGI(TAG, "Opening file %s", path);
         FILE *f = fopen(path, "a");
         if (f == NULL) {
             ESP_LOGE(TAG, "Failed to open file for writing");
             return ESP_FAIL;
         }
         ESP_LOGI(TAG, "Printing data to card: %s", data);
         fprintf(f, data);
         fclose(f);
         ESP_LOGI(TAG, "File written");

         return ESP_OK;
      }

      // Function that reads the contents of a given file from the SD Card
      static esp_err_t sd_read_file(const char *path)
```

```
{
   ESP_LOGI(TAG, "Reading file %s", path);
   FILE *f = fopen(path, "r");
   if (f == NULL) {
       ESP_LOGE(TAG, "Failed to open file for reading");
       return ESP_FAIL;
   }
   char line[MAX_CHAR_SIZE];
   fgets(line, sizeof(line), f);
   fclose(f);

   // strip newline
   char *pos = strchr(line, '\n');
   if (pos) {
       *pos = '\0';
   }
   ESP_LOGI(TAG, "Read from file: '%s'", line);

   return ESP_OK;
}

// Function to copy the contents of the sensor data to the global
variable data_to_write
void sd_data_config(char *input)
{
   snprintf(data_to_write, MAX_CHAR_SIZE, "%s\n", input);
}

// Function to deactivate the SD Card reader.
void sd_deact_device(void)
{
   ESP_LOGI(TAG, "Beginning SD Card deactivation process...");

   // All done, unmount partition and disable SPI peripheral
   esp_vfs_fat_sdcard_unmount(mount_point, card);
   ESP_LOGI(TAG, "Card unmounted");

   //deinitialize the bus after all devices are removed
   spi_bus_free(host.slot);
   ESP_LOGI(TAG, "Host slot freed");
}

// Function to initialize the SD Card reader.
void sd_init_device(void)
{
   esp_err_t ret;

   // Options for mounting the filesystem.
```

```
         // If format_if_mount_failed is set to true, SD card will be
partitioned and
         // formatted in case when mounting fails.
         esp_vfs_fat_mount_config_t mount_config = {
      #ifdef CONFIG_EXAMPLE_FORMAT_IF_MOUNT_FAILED
             .format_if_mount_failed = true,
      #else
             .format_if_mount_failed = false,
      #endif // EXAMPLE_FORMAT_IF_MOUNT_FAILED
             .max_files = 5,
             .allocation_unit_size = 16 * 1024
         };
         ESP_LOGI(TAG, "Initializing SD card");

         // Use settings defined above to initialize SD card and mount FAT
filesystem.
         // Note: esp_vfs_fat_sdmmc/sdspi_mount is all-in-one convenience
functions.
         // Please check its source code and implement error recovery when
developing
         // production applications.
         ESP_LOGI(TAG, "Using SPI peripheral");

         // By default, SD card frequency is initialized to SDMMC_FREQ_DEFAULT
(20MHz)
         // For setting a specific frequency, use host.max_freq_khz (range
400kHz - 20MHz for SDSPI)
         // Example: for fixed frequency of 10MHz, use host.max_freq_khz =
10000;
         host.max_freq_khz = 10000;

         ESP_LOGI(TAG, "Using SPI peripheral2");

         spi_bus_config_t bus_cfg = {
             .mosi_io_num = PIN_NUM_MOSI,
             .miso_io_num = PIN_NUM_MISO,
             .sclk_io_num = PIN_NUM_CLK,
             .quadwp_io_num = -1,
             .quadhd_io_num = -1,
             .max_transfer_sz = 4000,
         };

         ESP_LOGI(TAG, "Using SPI peripheral3");
         ret = spi_bus_initialize(host.slot, &bus_cfg, SDSPI_DEFAULT_DMA);
         ESP_LOGI(TAG, "Using SPI peripheral4");
         if (ret != ESP_OK) {
             ESP_LOGE(TAG, "Failed to initialize bus.");
             return;
         }
```

```
        // This initializes the slot without card detect (CD) and write
protect (WP) signals.
        // Modify slot_config.gpio_cd and slot_config.gpio_wp if your board
has these signals.
        sdspi_device_config_t slot_config = SDSPI_DEVICE_CONFIG_DEFAULT();
        slot_config.gpio_cs = PIN_NUM_CS;
        slot_config.host_id = host.slot;

        ESP_LOGI(TAG, "Mounting filesystem");
        ret = esp_vfs_fat_sdspi_mount(mount_point, &host, &slot_config,
&mount_config, &card);

        if (ret != ESP_OK) {
            if (ret == ESP_FAIL) {
                ESP_LOGE(TAG, "Failed to mount filesystem. "
                         "If you want the card to be formatted, set the
CONFIG_EXAMPLE_FORMAT_IF_MOUNT_FAILED menuconfig option.");
            } else {
                ESP_LOGE(TAG, "Failed to initialize the card (%s). "
                         "Make sure SD card lines have pull-up resistors in
place.", esp_err_to_name(ret));
      #ifdef CONFIG_EXAMPLE_DEBUG_PIN_CONNECTIONS
                check_sd_card_pins(&config, pin_count);
      #endif
            }
            return;
        }
        ESP_LOGI(TAG, "Filesystem mounted");

        // Card has been initialized, print its properties
        sdmmc_card_print_info(stdout, card);

        // Use POSIX and C standard library functions to work with files.

        // First create a file.
        const char *file_hello = MOUNT_POINT"/TEMP.txt";
        char data[MAX_CHAR_SIZE];
        snprintf(data, MAX_CHAR_SIZE, "%s %s!\n", "TEMP", card->cid.name);
        ret = sd_write_file(file_hello, data);
        if (ret != ESP_OK) {
            return;
        }

        char *file_foo = MOUNT_POINT"/Data.txt";

        // Check if destination file exists before renaming
        struct stat st;
        if (stat(file_foo, &st) == 0)
        {
            //Delete the file if it already exists.
            unlink(file_foo);
        }

        // Rename original file
        ESP_LOGI(TAG, "Renaming file %s to %s", file_hello, file_foo);
        if (rename(file_hello, file_foo) != 0) {
            ESP_LOGE(TAG, "Rename failed");
            return;
        }

        //Copy the name of file_foo to the global variable, file_path
        strcpy(file_path, MOUNT_POINT"/TEMP.txt");
        strcpy(file_path, file_foo);

        //Read the contents of file_foo to ensure it works.
        ret = sd_read_file(file_foo);
        if (ret != ESP_OK) {
            return;
        }

        // Format FATFS
        ret = esp_vfs_fat_sdcard_format(mount_point, card);
        if (ret != ESP_OK) {
            ESP_LOGE(TAG, "Failed to format FATFS (%s)",
esp_err_to_name(ret));
            return;
        }

        if (stat(file_foo, &st) == 0) {
            ESP_LOGI(TAG, "file still exists");
            return;
        } else {
            ESP_LOGI(TAG, "file doesn't exist, format done");
        }
    }
```

SD_Card.h:

```
    //
    // Created by Casey Halbmaier (crhalby) on February 7th, 2024
    //
    #include "main.h"

    #ifndef WEARABLE_PTSD_DETECTION_SD_CARD_H
    #define WEARABLE_PTSD_DETECTION_SD_CARD_H

    #include <string.h>
```

```
      #include <sys/unistd.h>
      #include <sys/stat.h>
      #include "esp_vfs_fat.h"
      #include "sdmmc_cmd.h"

      ////Global Variable Declarations
      extern char file_path[];
      extern char data_to_write[];

      extern int press;

      // Functions used in SD_Card.c
      // Function that writes to the specified file from the SD Card; Takes in
the global variable file_path and data_to_write
      esp_err_t sd_write_file(const char* path, char* data);

      // Function that writes the sensor data to the global variable
data_to_write from the given sensor data
      void sd_data_config(char *input);

      // Function that unmounts the SD Card once data collection is finished
      void sd_deact_device();

      // Function that mounts the SD Card and creates a file for writing
      void sd_init_device();

      #endif //WEARABLE_PTSD_DETECTION_SD_CARD_H
```

Radio.c:

```
      //
      // Created by Cobeasta on 4/25/2024.
      //

      #include "includes/Radio.h"
      static const char *TAG = "Radio.c";

      const gpio_num_t rft_pin_num = CONFIG_WEARABLE_RF_TRANSMIT;
      /**
      * Configure pin as output
      *
      **/
      void radio_init()
      {
         ESP_LOGI(TAG, "init start");
       gpio_config_t pin_config = {
         .pin_bit_mask = (1ULL << rft_pin_num),
         .mode = GPIO_MODE_OUTPUT,
         .pull_down_en = 0,
         .pull_up_en = 0
       };
       gpio_config(&pin_config);
         ESP_LOGI(TAG, "init end");

      }

      /**
      Send signal to ping vibration sensor
      * Send pulse for 2 seconds
      * Wait 2 seconds
      * Do it again
      **/
      void radio_ping(int numPings)
      {
         ESP_LOGI(TAG, "ping");
       int count = 0;
       while(count < numPings)
       {
       gpio_set_level(rft_pin_num, 1);
       vTaskDelay(2000/portTICK_PERIOD_MS);

       gpio_set_level(rft_pin_num, 0);
       vTaskDelay(2000/portTICK_PERIOD_MS);
         count++;
       }

      // Write 1
      // wait 2 seconds
      // write 0
      // wait 2 seconds
      // repeat numPings times
      }
```

Radio.h:

```
#ifndef WEARABLE_PTSD_DETECTION_RADIO_H
#define WEARABLE_PTSD_DETECTION_RADIO_H
#include "main.h"
#include "driver/gpio.h"
#include "freertos/FreeRTOS.h"

void radio_ping(int numPings);
void radio_init();
#endif
```

button.c:

```
//
```

```
// Created by Casey Halbmaier (crhalby) on April 25th, 2024
//

#include "includes/button.h"

static const char *TAG = "button.c";

// State variablesw
int press = 0;

// create gpio button
button_config_t gpio_btn_cfg = {
    .type = BUTTON_TYPE_GPIO,
    .long_press_time = CONFIG_BUTTON_LONG_PRESS_TIME_MS,
    .short_press_time = CONFIG_BUTTON_SHORT_PRESS_TIME_MS,
    .gpio_button_config = {
        .gpio_num = 12,
        .active_level = 0,
    },
};

button_event_config_t cfg = {
    .event = BUTTON_LONG_PRESS_START,
    .event_data.long_press.press_time = 2000,
};

void button_click_radio(void *arg, void *usr_data) {
    ESP_LOGI(TAG, "BUTTON_LONG_CLICK");
     radio_ping(3);
}

void button_click_collect(void *arg, void *usr_data) {
    ESP_LOGI(TAG, "BUTTON_SINGLE_CLICK");

    if(press == 0) {
        press = 1;
    }
    else {
        press = 0;
    }
}

void button_init() {
    button_handle_t gpio_btn = iot_button_create(&gpio_btn_cfg);
    if(NULL == gpio_btn) {
        ESP_LOGE(TAG, "Button create failed");
    }
    ESP_LOGI(TAG, "Button create success!");
```

```
        iot_button_register_cb(gpio_btn, BUTTON_SINGLE_CLICK,
button_click_collect, NULL);

        iot_button_register_event_cb(gpio_btn, cfg, button_click_radio, NULL);
    }
```

button.h:

```
//
// Created by Casey Halbmaier (crhalby) on April 25th, 2024
//

#include "main.h"
#include "Radio.h"

#ifndef WEARABLE_PTSD_DETECTION_BUTTON_H
#define WEARABLE_PTSD_DETECTION_BUTTON_H
#include "esp_vfs_fat.h"
#include "driver/gpio.h"
#include <iot_button.h>

extern button_event_t event;

// Function definitions
void button_init();

#endif
```

biosensor.c

```
// Author: Caden Backen
// MAX30102_Sensor.c
// Last Updated: 2/28/2024

#include "includes/bio_sensor.h"

#define MAX86150_FIFO_CFG_ROLLOVER_EN_MASK    (1 << 4)
#define MAX86150_MAX_NUM_CHANNELS 4;


// fifo registers
static const uint8_t FIFO_READ_PTR = 0x06;
static const uint8_t FIFO_OVFLW_PTR = 0x05;
static const uint8_t FIFO_WRITE_PTR = 0x04;
static const uint8_t FIFO_DATA = 0x7;

// Register addresses

// interrupts
```

```
      static const uint8_t MAX86150_REG_INT_STATUS2 = 0x01;
      static const uint8_t MAX86150_REG_INT_STATUS = 0x02;
      //// fifo
      static const uint8_t MAX86150_REG_FIFOCONFIG = 0x08;
      static const uint8_t MAX86150_REG_FIFODATACTL = 0x09; // Samples FD2, FD
1
      static const uint8_t MAX86150_REG_FIFODATACTL2 = 0x0A; // Samples FD 4,
FD 3
      static const uint8_t MAX86150_REG_SYSCONTROL = 0x0D;
      //
      static const uint8_t MAX86150_REG_PPGCFG = 0x0E;
      static const uint8_t MAX86150_REG_PPGCFG2 = 0x0f;
      //
      ////static const uint8_t MAX86150_REG_PROXINT_THRESH = 0x10;
      //// led waveform config
      static const uint8_t MAX86150_REG_LED_PA = 0x11;
      static const uint8_t MAX86150_REG_LED2_PA = 0x12;
      static const uint8_t MAX86150_REG_LED_RGE = 0x14;
      static const uint8_t MAX86150_REG_LED_PILOT_PA = 0x15;
      //
      static const uint8_t MAX85150_REG_PRODID = 0xff;

      //static const uint8_t MAX86150_PROXINTTHRESH = 0x10;
      // led types
      static const uint8_t FIFO_FDX_IR = 1;
      static const uint8_t FIFO_FDX_RED = 2;
      //static const uint8_t FIFO_FDX_IR_PILOT = 5;
      //static const uint8_t FIFO_FDX_RED_PILOT = 6;
      // I2C config
      static i2c_port_t i2c_port = I2C_NUM_0;
      static const int i2c_timeout = 1000;
      static const char *TAG = "MAX_C";
      gpio_num_t max_sclk = CONFIG_WEARABLE_MAX_SCLK; // set from
Kconfig.projbuild
      gpio_num_t max_sda = CONFIG_WEARABLE_MAX_SDA; // set from
Kconfig.projbuild
      static int max_device_addr = 0x5e;

      // I2C resources
      static i2c_master_bus_handle_t bus_handle;
      static i2c_master_dev_handle_t dev_handle;

      // sensor values

      sense_struct sense;



      //static const uint8_t addr_size = 1;
      /**
      * Setup I2C resources for the sensor
      */
      void bus_init();

      /**
      * Destroy I2C resources for the sensor
      */
      void bus_destroy();

      /**
      * Initialize/configure MAX sensor
      */
      void max_init();

      uint8_t read_part_id();

      /**
      * Get raw IR count
      * @return
      */
      uint8_t get_red();

      uint8_t read_reg8(uint8_t reg); // read byte from register
      void write_reg8(uint8_t reg, uint8_t value); // write byte to register

      uint8_t read_reg8(uint8_t reg)
      {
         uint8_t data;

         ESP_LOGI(TAG, "Reading");
         int ret = i2c_master_transmit_receive(dev_handle, (uint8_t * ) & reg,
1, (uint8_t * ) & data, 1, i2c_timeout);
         if (ret == ESP_OK)
         {
             return data;
         } else if (ret == ESP_ERR_TIMEOUT)
         {
             ESP_LOGW(TAG, "Bus is busy");
             return -1;
         } else
         {
             ESP_LOGW(TAG, "Read failed");
             return -1;
         }
         return -1;
      }

      void write_reg8(uint8_t reg, uint8_t value)
      {
         short data = reg << 8 | value;

         i2c_master_transmit(dev_handle, (uint8_t * ) & data, 2, i2c_timeout);


      }


      uint8_t read_part_id()
      {
         uint8_t pid = read_reg8(MAX85150_REG_PRODID);
         return pid;
      }

      void bus_init()
      {
         i2c_master_bus_config_t i2c_mst_config = {
                 .clk_source = I2C_CLK_SRC_DEFAULT,
                 .i2c_port = i2c_port,
                 .scl_io_num = max_sclk,
                 .sda_io_num = max_sda,
                 .glitch_ignore_cnt = 7,
                 .flags.enable_internal_pullup = true,
         };

         ESP_ERROR_CHECK(i2c_new_master_bus(&i2c_mst_config, &bus_handle));

         i2c_device_config_t dev_cfg = {
                 .device_address = max_device_addr,
                 .scl_speed_hz = 100000,
         };

         ESP_ERROR_CHECK(i2c_master_bus_add_device(bus_handle, &dev_cfg,
&dev_handle));

         ESP_ERROR_CHECK(i2c_master_probe(bus_handle, max_device_addr,
i2c_timeout));

         uint8_t part_id = read_part_id();
         printf("Part ID: %d\n", part_id);
      }

      void bus_destroy()
      {
         i2c_master_bus_rm_device(dev_handle);
      }
      // Initializes the sensor and sets the correct configurations
      void bio_init()
      {
         bus_init();

         // Initialize MAX sensor
         max_init();
      }

      int sample_average = 4; // number of samples of same type to average

      void clear_fifo()
      {
         write_reg8(FIFO_WRITE_PTR, 0);
         write_reg8(FIFO_READ_PTR, 0);
         write_reg8(FIFO_OVFLW_PTR, 0);
      }
      void max_init()
      {
         printf("Initializing max sensor\n");
      //    FIFO settings

         write_reg8(MAX86150_REG_SYSCONTROL, 0x01); // power-on-reset sequenece
      //    write_reg8(0x0d, 0x01); // reset part
      //    vTaskDelay(100 / portTICK_PERIOD_MS);
      //
      //    write_reg8(0x08, 0x1f); //fifo rolls_on_full and read fifo data
when there are 17 samples
      //
      //    write_reg8(0x09, 0x21); //led1 in slot 1, led2 in slot 2
      //    write_reg8(0x0a, 0x00); // empty slot 3 and 4
      //
      //    write_reg8(0x0d, 0x04); //enable fifo
      //
      ////    Acquisition settings
      //    write_reg8(0x11, 0x55); // led1 current setting
      //    write_reg8(0x12, 0x55); // led2 current setting
      //    write_reg8(0x0e, 0xd3); // range setting for ppg adc rge
      //    write_reg8(0x0f, 0x18); // 20 uS delay from rising edge of led to
start of integration
         uint8_t fifo_afull_cfg = 0x7f; // no almost full interrupts, roll on
full, almost full at 15 free samples left


         write_reg8(MAX86150_REG_FIFOCONFIG,
                    fifo_afull_cfg); // trigger interrupt on 15 samples left
free, rollover when full

         // time slots 2, 1: pilot IR/red
         uint8_t fd2_fd1 = ((FIFO_FDX_IR << 4) | FIFO_FDX_RED); //
         write_reg8(MAX86150_REG_FIFODATACTL, fd2_fd1);
      // Interrupts
         uint8_t intstatus = read_reg8(MAX86150_REG_INT_STATUS);
         write_reg8(MAX86150_REG_INT_STATUS, intstatus & 0x0f); // disable all
interrupts

         uint8_t intstatus2 = read_reg8(MAX86150_REG_INT_STATUS2);
         write_reg8(MAX86150_REG_INT_STATUS, intstatus2 & 0x7b); // disable
other interrupts
         // time slots 4, 3 IR/red
      //    uint8_t fd4_fd3 = ((FIFO_FDX_IR << 4) | FIFO_FDX_RED); // ir/red to
get readings
         write_reg8(MAX86150_REG_FIFODATACTL2, 0x00); // disable red/ir


         // adc range: 10 (31.25 pA to 16384 nA)
         // sample_rate: 0010 (50)
         // ppg led PW: 01 (100 uS)

         write_reg8(MAX86150_REG_PPGCFG, 0b10001001);

         write_reg8(MAX86150_REG_PPGCFG2, 0x02); // sample averaging 4

         write_reg8(MAX86150_REG_LED_PA, 0xff); // set max pulse amplitude
         write_reg8(MAX86150_REG_LED2_PA, 0xff); // set max pulse amplitude
         write_reg8(MAX86150_REG_LED_PILOT_PA, 0xff); // set pilot led pulse
amlitude

         write_reg8(MAX86150_REG_LED_RGE, 0x00); // set max pulse amplitude


         write_reg8(MAX86150_REG_SYSCONTROL, 0x04); // power-on-reset sequenece
         clear_fifo();
      }

      uintptr_t  get_read_ptr()
      {
         return read_reg8(FIFO_READ_PTR);
      }

      uintptr_t get_write_ptr()
      {
         return read_reg8(FIFO_WRITE_PTR);
      }

      int burst_read(uint8_t reg, uint8_t * data, int len)
      {
         ESP_LOGI(TAG, "Reading");
         int ret = i2c_master_transmit_receive(dev_handle, (uint8_t * ) & reg,
1, (uint8_t * ) & data, len, i2c_timeout);
         if (ret == ESP_OK)
         {
             return 0;
         } else if (ret == ESP_ERR_TIMEOUT)
         {
             ESP_LOGW(TAG, "Bus is busy");
             return -1;
         } else
         {
             ESP_LOGW(TAG, "Read failed");
             return -2;
         }
      }
      /**
      * Poll sensor for new data.
      * Store data new data that is found
      */
      uint16_t get_sample()
      {
         ESP_LOGI(TAG, "getting sample");
         int active_devices = 2;
         uint8_t read_ptr = get_read_ptr();
         uint8_t write_ptr = get_write_ptr();
         int num_samples = 0;

         if(write_ptr != read_ptr) // if we have new data
         {
             num_samples = write_ptr - read_ptr;
             if(num_samples < 0) num_samples += 32; // wrap condition
             int bytestoRead = num_samples * active_devices * 3;


             while(bytestoRead > 0)
             {
                 sense.head++; // advance head of storage struct
                 sense.head %= STORAGE_SIZE; //wrap if needed

                 // getting first value
                 uint32_t raw_l;
                 uint8_t temp[sizeof(uint32_t)];

                 i2c_master_transmit(dev_handle,  (uint8_t * ) & FIFO_DATA, 1,
i2c_timeout);

                 temp[3]= 0;
```

```
                    i2c_master_receive(dev_handle, (uint8_t * ) &temp[2], 1,
i2c_timeout);
                    i2c_master_receive(dev_handle, (uint8_t * ) &temp[1], 1,
i2c_timeout);
                    i2c_master_receive(dev_handle, (uint8_t * ) &temp[0], 1,
i2c_timeout);
                    memcpy(&raw_l, temp, sizeof(raw_l));
                    sense.red[sense.head] = raw_l;
                    bytestoRead -= active_devices * 3;
                    if(active_devices > 1)
                    {
                        temp[3]= 0;
                        i2c_master_receive(dev_handle, (uint8_t * ) &temp[2], 1,
i2c_timeout);
                        i2c_master_receive(dev_handle, (uint8_t * ) &temp[1], 1,
i2c_timeout);
                        i2c_master_receive(dev_handle, (uint8_t * ) &temp[0], 1,
i2c_timeout);
                        memcpy(&raw_l, temp, sizeof(raw_l));
                        sense.IR[sense.head] = raw_l;
                    }
                    bytestoRead -= active_devices * 3;
                }
                ESP_LOGI(TAG, "Got  samples");
            }
            return num_samples;
        }
```

biosensor.h:

```
        // Author: Caden Backen
        // MAX30102_Sensor.h
        // Last Updated: 2/28/2024
        #include "stdlib.h"
        #include "stdio.h"
        #include "driver/i2c_master.h"
        #include "esp_log.h"
        #include "freertos/FreeRTOS.h"
        #include <string.h>
        // Configuration Pg. 10

        #define i2cWriteAddress 0xAE
        #define i2cReadAddress 0xAF

        //FIFO Configuration (0x08)

            //Sample Average B7-B5
            #define SAMPLE_AVERAGE_1 1
            #define SAMPLE_AVERAGE_2 2
```

```
    #define SAMPLE_AVERAGE_4 3
    #define SAMPLE_AVERAGE_8 4
    #define SAMPLE_AVERAGE_16 5
    #define SAMPLE_AVERAGE_32 6

    //FIFO Roll Over B4

    //FIFO Almost Full B3-B0

//Mode Configuration (0x09)

    //Shut Down Control B7

    //Reset Control B6

    //Mode Control B2-B0
    #define HEARTRATE__REDONLY__MODE 1
    #define SPO2__RED__IR__MODE 2
    #define MULTI_LED__RED__IR__MODE 3

//SPO2 Configuration (0x0A)

    //SPO2 ADC Range Control B6-B5
    #define ADC_RANGE_2048 1
    #define ADC_RANGE_4096 2
    #define ADC_RANGE_8192 3
    #define ADC_RANGE_16384 4

    //SPO2 Sample Rate B4-B2
    #define SPO2_SAMPLE_RATE_50 1
    #define SPO2_SAMPLE_RATE_100 2
    #define SPO2_SAMPLE_RATE_200 3
    #define SPO2_SAMPLE_RATE_400 4
    #define SPO2_SAMPLE_RATE_800 5
    #define SPO2_SAMPLE_RATE_1000 6
    #define SPO2_SAMPLE_RATE_1600 7
    #define SPO2_SAMPLE_RATE_3200 8

    //LED Pulse Width and ADC Resolution B1-B0
    #define LED_Pulse_Width_69 0
    #define LED_Pulse_Width_118 1
    #define LED_Pulse_Width_215 2
    #define LED_Pulse_Width_411 3

//LED Pulse Amplitude (0x0C - 0x0D)


//Multi-LED Mode Configuration (0x11 - 0x12)
```

```
    //SLOTx[2:0] Settings
    #define MULTI_LED_SLOT_ACTIVITY_NONE 1
    #define MULTI_LED_SLOT_ACTIVITY_RED 2
    #define MULTI_LED_SLOT_ACTIVITY_IR 3

#define STORAGE_SIZE 4
    typedef struct Record
    {
        uint32_t red[STORAGE_SIZE];
        uint32_t IR[STORAGE_SIZE];
        int32_t ecg[STORAGE_SIZE];
        uint8_t head;
        uint8_t tail;
    } //circular buffer of readings
sense_struct;

//Initializes the sensor and sets the correct configurations
void bio_init();
uint16_t get_sample();
```