FASCIA: A SLEEP MASK FOR CONDUCTING SLEEP STUDIES

By **Walaa Alkhanaizi**

B.S. M.I.T., 2019

Submitted to the

Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology
May 2020

Author: ____
Department of Electrical Engineering and Computer Science
10th May 2020
Certified by: ____
Pattie Maes, Professor of Media Arts and Sciences, Thesis Supervisor
10th May 2020
Accepted by: ____
Katrina LaCurts, Chair, Master of Engineering Thesis Committee

FASCIA: A SLEEP MASK FOR CONDUCTING SLEEP STUDIES

By
Walaa Alkhanaizi

Submitted to the Department of Electrical Engineering and Computer Science on May 10, 2020 in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

In this thesis, I discuss the importance of sleep and therefore the study of sleep. I highlight limitations with existing methodologies to conduct sleep studies and collect sleep data, and present a solution to overcome current limitations by providing better mechanisms for sensing during sleep in the wild.

This document details the thought process of every aspect of design and development of the progress made on the project so far. First, I present the motivation for the project and provide general background. Second, I discuss the physiological signals that sleep studies monitor and their relationship to sleep. If the reader is familiar with these physiological signals and how they are relevant to sleep studies, they should feel free to skip that section. Next, I provide an overview of some existing alternatives in the market and discuss why they do not satisfy the purpose of in-the-wild sleep studies. Next, I detail the design of the device, physically and on the system level. Then, I go into a detailed description of the components of the device in hardware, firmware, and software. I include a brief description of some of the efforts made in the code to make it easier to debug while developing. Lastly, I discuss what work was completed, and what work remains to be done. I close with a full list of the tasks remaining and some implementation concerns.

There is a glossary near the end of the document of terms and acronyms I use throughout the thesis. Feel free to consult it should any confusion arise regarding the meaning of words used.

The document ends with a list of appendices starting with a complete usage guide for the system in its current state. The other appendices include copies of all the firmware and software code, and circuit and PCB designs.

Thesis Supervisor: Pattie Maes
Title: Professor of Media Arts and Sciences,

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

We know that it is essential for humankind to get sufficient and regular sleep, not just for a good rest, but also for a collection of critical cognitive developments in the brain. Although we still understand very little about sleep, we know that some of the most important mental and physical processes in the human body happen during sleep, such as memory consolidation and immune system fortification. Sleep studies are imperative because they help doctors to diagnose patients with sleep disorders that would otherwise be very difficult to find conclusive symptoms for. Sleep studies require patients to come in and sleep in "sleep centers" which are equipped for people to sleep while the brain and body of the subjects are monitored. The data typically collected involves EEG sensing, eye movement, oxygen levels in the blood, heart rate and breathing rate, snoring, and body movements [1].

Sleep studies have largely been viewed as a nuisance for the subjects being studied. This is due to the major discomfort caused when the subject must come into the research center or hospital and sleep there while their vitals and different physiological signals are constantly monitored by bulky equipment. In order to detect these signals, a variety of electrodes and sensors are distributed across the head and the rest of the body, and secured using tape or glue, and therefore cause significant discomfort. Centers assure patients that they'll "still have plenty of room to move and get comfortable" and that they are being monitored by sleep study technologists who "can help if they need to use the bathroom" [1]. Still, according to the National Sleep Foundation, many people wonder how they will be able to sleep under such conditions. Researchers believe this setup and procedure result in inaccurate or at least inconsistent data as the subjects are not sleeping as they normally would- in the comfort of their home, free of unfamiliar wires and electrodes probing their bodies.

That is where this project comes in. With the previous insights in mind, the project aims to tackle those challenges by creating a comfortable and minimal "sleep mask" which houses all the required sensors and electrodes to record the vitals and signals needed for sleep studies, in a compact and user-friendly format. The device takes the form of a sleep mask, which consists of a flexible printed circuit board with integrated electrodes and sensors that are close to the skin, and two conventional PCBs to house the components that perform the signal processing, data analysis, signal forwarding and storage, farther away from the skin. The resulting mask can be taken home, enabling sleep studies to be conducted "in the wild."

BACKGROUND

SIGNALS OF INTEREST

In order to conduct polysomnography (sleep studies), technologists typically place sensors on the patient's scalp, temples, chest, and legs, as well as a clip on the finger, all of which are connected by wires to a computer. This is in preparation to monitor the following signals: brain waves, eye movements, heart rate, breathing pattern, blood oxygen level, body position, chest and abdominal movement, limb movement, and snoring [4].

The device we are building for the purpose of improving sleep studies aims to integrate each of the following sensors, which detect the signals specified below:

EEG (Electroencephalogram)

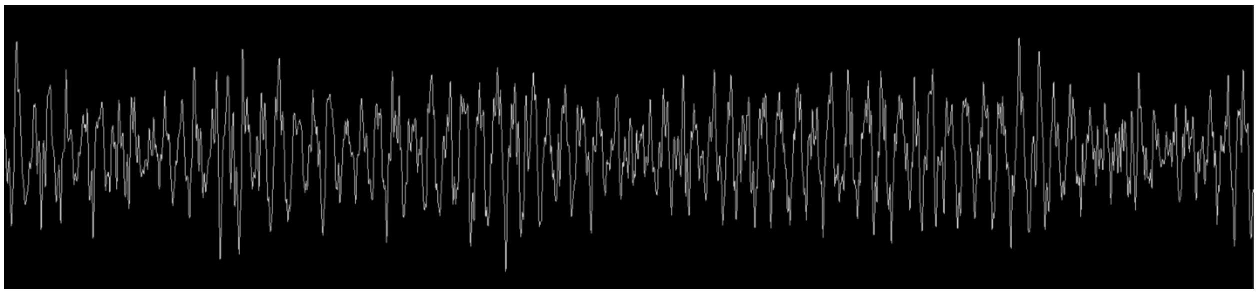


FIGURE 1: EEG SIGNAL (ALPHA)

EEG is an electrophysiological detection mechanism to monitor electrical activity of the brain and record brain wave patterns. Usually this sensor takes the form of noninvasive electrodes (small metal surface connected with thin wire) contacting the scalp, although versions of it which pierce the skin also exist. EEG wave patterns are well-studied and there are known patterns that healthy brains emit, so doctors can observe abnormal patterns and study whether they are a cause for concern and what they might entail [5]. For our purposes, EEG can be used to monitor sleep stages and cycles; whether the patient is in REM or NREM.

EOG (Electrooculography)

EOG is a physiological signal that detects and measures eye movements (through the eyelid) by measuring the corneo-retinal distance between the front and back of the eye. This is done by placing two electrodes on both sides of the eye, either the right and left, or front and back, and measuring the potential difference between them, which would vary as the eye moves [7]. For our purposes, this is used to detect what stage of sleep the patient is in. REM is an acronym for rapid eye movement, so we can know that the patient is in REM by detecting random and rapid eye movement.

EMG (Electromyography)

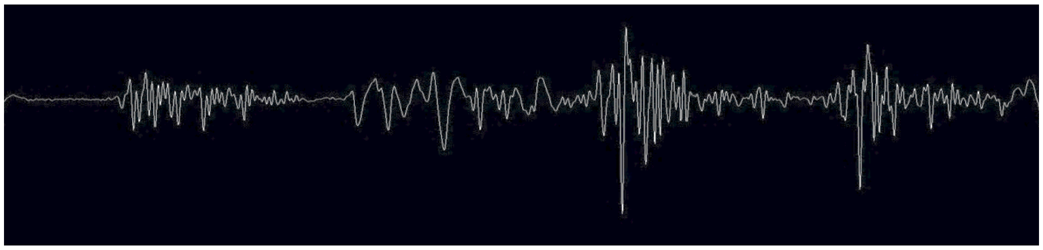


FIGURE 2: EMG SIGNAL

EMG is performed to evaluate the healthiness of muscles and the associated motor neurons (nerves that control those muscles). The motor neurons transmit signals to the muscles that cause muscles to either contract or relax. The electrical measurement of such a signal is called EMG. Monitoring and studying the EMG signal can allow doctors to detect muscle and nerve disorders [6]. For our application, EMG variance can be used to assess sleep behaviors in terms of muscle movements around the body. Specifically, cheek, forehead, and chin EMG signals are the ones we focus on in our device.

EDA (Electrodermal Activity)

EDA is also known as GSR (galvanic skin response), and it represents skin conductance which continuously varies in the human body. EDA is measured by using two electrodes that make contact with the skin and measuring the resistance between them. This relies on the discovery that the resistance of skin changes based on the (even minute) activity of the sweat glands in the skin, which are controlled by the sympathetic nervous system. EDA is a measure of psychological and physiological arousal. For our application, this signal means we are able to detect the emotional response and state of the patient.

PPG (Photoplethysmogram)

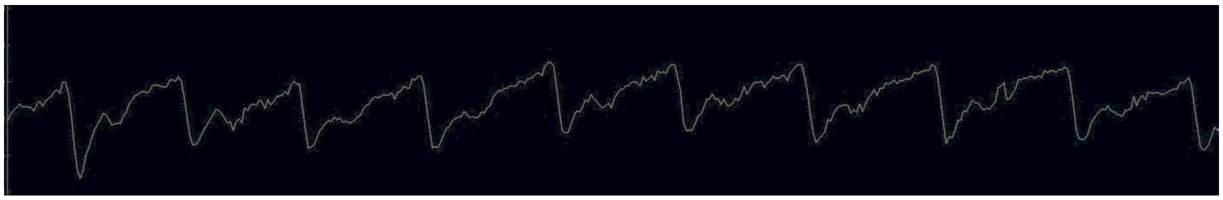


FIGURE 3: PPG SIGNAL SHOWING HEART BEATS

PPG uses a type of sensor which is an optically detected plethysmogram, used to measure the volume of blood going through the veins under the skin. This is often done by shining a light on the skin and measuring variations in light absorption. Because the volume of blood changes as the heart pumps it to the periphery, this enables us to measure heart rate.

Temperature

By measuring the temperature of the body, we are able to deduce the stage and depth of sleep that the patient is in. Sleep stages are associated with temperature ranges; we can observe drops and changes depending on the stage in the cycle.

Motion sensing

Measuring the movements of the patients via Gyroscopes, magnetometers, or accelerometers enables us to detect muscle spasms and whether the patient is tossing and turning, which can be valuable data for doctors to be able to diagnose certain sleep disorders.

EXISTING SOLUTIONS

The main challenges with conventional sleep studies are twofold: the patient's comfort, and the accuracy of the collected data, given the context in which the patients sleep. There are three existing products in the market that try to tackle those issues, that we are aware of.

First, a product called Neuroon Open, which is marketed as a sleep enhancing wearable device. This is an IoT product whose selling point is helping customers improve the quality of their sleep through EEG monitoring and lucid dreaming induction, as well as smart meditation sessions. This is achieved by allowing the IoT device to control the lighting, music, and the temperature in the bedroom of the customers, which the device adjusts in accordance with what the customer needs are based on the sleep stage they are in [2]. This device is mainly aimed at helping an individual sleep better by monitoring brain waves, and less at helping researchers study sleep in order to diagnose patients with potential sleep disorders, and better understand sleep in general. Therefore, the use-case for this product is limited to helping the customer sleep more soundly, and is thus much narrower than the goal of this project.

Second, a product called Muse Headband. Muse monitors mental activity and uses it to produce "guiding" nature sounds to help the user reach a mental state of what they call a "focused calm." Muse selects different sounds to represent different states of mind: if the subject is calm, it plays peaceful weather sounds, and as the customer starts to get more distracted or busy-minded, it starts to play more stormy and loud weather sounds to cue the user to focus their attention back to their meditation routine. Muse is also mostly limited to the use of EEG sensing to help guide users through immersive meditation sessions. Some iterations of the product include PPG sensors, as well as a gyroscope and accelerometers, although all of these added sensors are used to optimize the same functionality: helping the user to calm their mind [3].

Third, a product called ZMax by Hypnodyne Corp. This product takes the form of an elastic headband with a box that sits on the forehead housing all the sensors and electronics. It has two EEG channels, measures heartrate, skin temperature, light and noise levels, and head position and movement. It comes with a range of software suites and options, as well as add-on sensors to optionally monitor other signals. ZMax seems to target a wide range of audiences including researchers as well as consumers, attracting them with lucid-dreaming specific setups and tutorials [32].

All of these products roughly satisfy the form factor for this project (though ZMax is on the bulkier side), but they lack sensing a some of the signals that this project aims to encompass with a design that supports medical and scientific sleep research.

GETTING STARTED

The PhD Research Assistant that I am working with for this project with, Guillermo Bernal, had previously developed an AR/VR headset with most of the same sensors and some powerful signal processing as the application required measuring more signals and using them to manipulate other signals related to graphics. Starting from that project, it was decided that the form factor would definitely need to be reduced, which was not an issue since this project did not require any of the optics-related components from the previous project.

THE DEVICE: DESIGN

To overcome the shortcomings of existing products for the purpose of sleep research, we made sure to design our product so as to maximize both the quantity and quality of sensor signals, as well as user comfort (which helps produce more accurate data).

PHYSICAL DESIGN

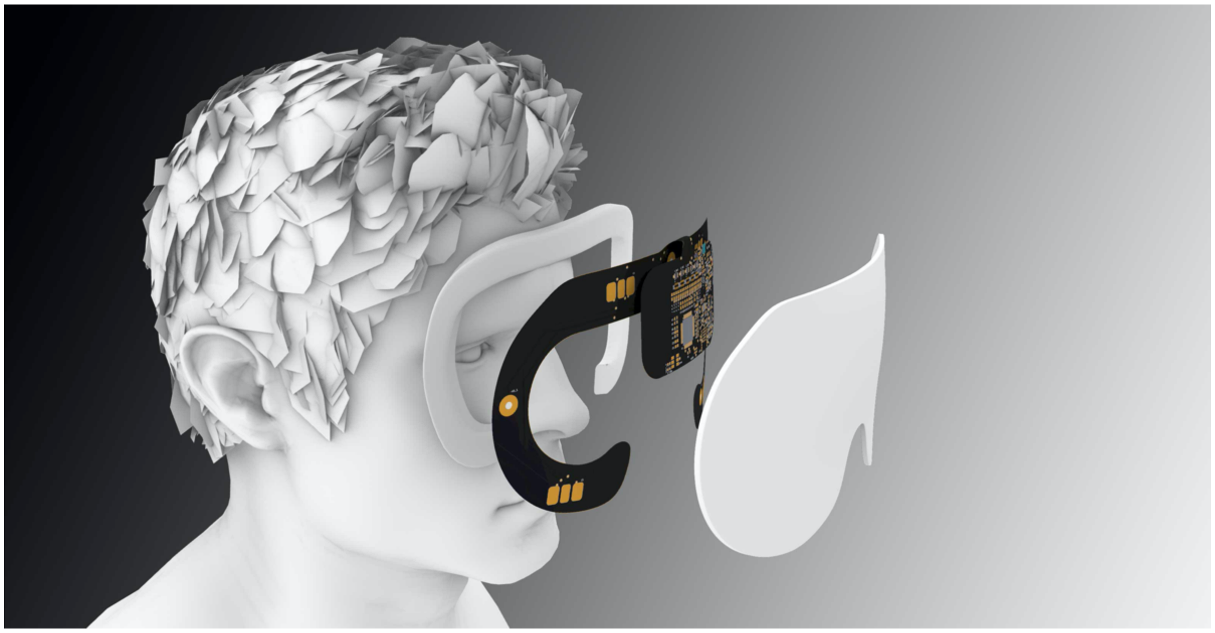


FIGURE 4: FORM FACTOR OF THE SLEEP MASK DESIGNED FOR THIS PROJECT

The physical design of the device was decided with the most minimal footprint to be the least intrusive it possibly can, so as to enable the user to be as comfortable as possible, and the data to be as accurate and noise and error-free as possible. This resulted in the selection of the sleep mask as the form of the device- a garment used only while people sleep for the purpose of aiding in sleep. For the sensors requiring electrodes, a flexible PCB design was used to maximize user comfort. Other considerations included placement of components other than electrodes: those were initially going to be in one or two larger conventional PCBs on the forehead, but they were recently changed to be two equally sized conventional PCBs in the eye area where foam padding and fabric will cover them.

Work was done to ensure the non-flexible PCBs in the device were appropriate. The initial design involved placing a somewhat large PCB on the forehead, connected to a smaller PCB above it. In order to maximize comfort, the form factor of each PCB was

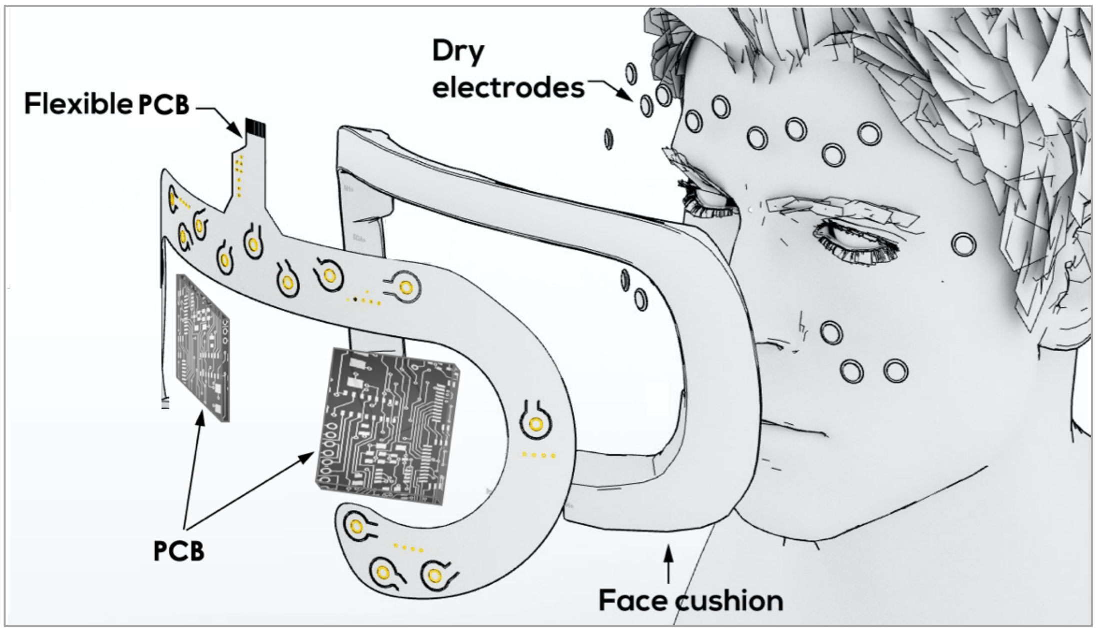


FIGURE 5: EXPLODED VIEW DISSECTING THE COMPONENTS OF THE DEVICE

reduced such that we can fit each PCB in one eye-patch of the sleep mask. This was done by revising the PCB design and, while checking for errors and correcting them, also removing unnecessary connections or unused wires.

As shown in figures 5 & 6, the emptiness inside the rim of the mask, which is the flexible PCB, houses the two conventional PCBs, with additional padding on both sides.

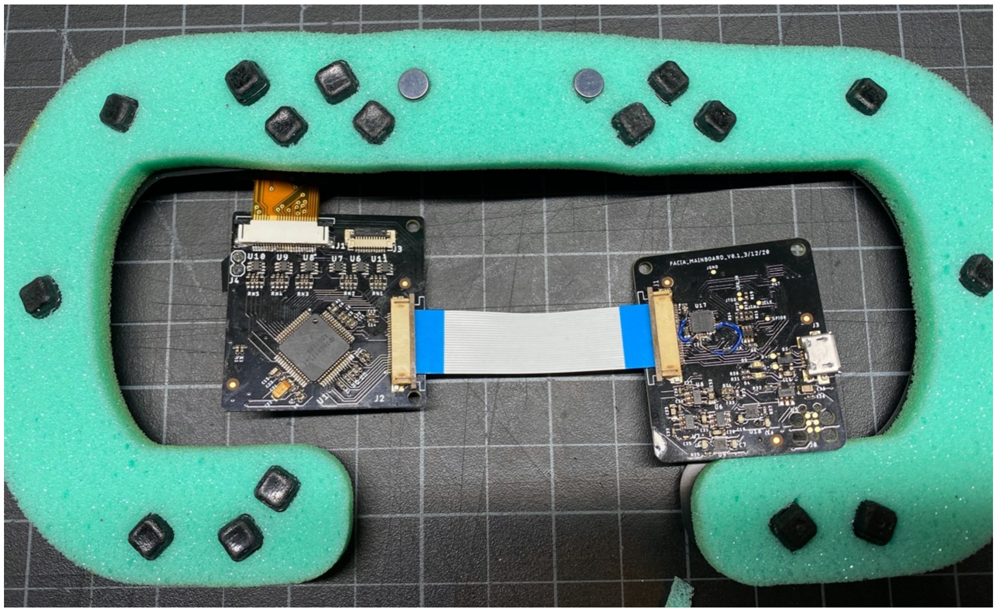


FIGURE 6: EARLY PROTOTYPE OF FASCIA, SHOWING THE MAIN PARTS

SYSTEM DESIGN

The device was designed with free and open-source software ethics in mind, therefore all the designs and information are publicly available for anyone to benefit from. This also guided many design decisions regarding hardware and software selection: the hardware design was developed in Eagle and the firmware was developed in the Arduino language and IDE, and the visualization software is written in Python.

Additionally, due to the importance of privacy of information, the goal is to eventually have the device encrypt the data before sending it over the network to keep it safe from any sniffers.

Speed and power efficiency are essential to the project as the device is going to be wirelessly worn throughout an entire night. This means we have a relatively small battery that we must conserve to last as the system's power source all night. This led to some decisions regarding data rates and burst data packaging, as well as using interrupts versus polling the integrated circuit chips (ICs) that comprise the device. The data rate is as fast as possible to allow the processing code to perform its tasks without interruption, while not being so fast that we receive too much data that is redundant or unnecessary. To ensure these delicate timing constraints, we only use one device's interrupt feature and service it with a routine which reads, converts, and stores the data, so that the other devices do not interrupt while this is being serviced or while the code is processing something else, to prevent messier handling and less predictable behaviors and timings.

HARDWARE

The hardware architecture consists of a main processing unit and wireless communication, a couple of physiological sensing units, and a flexible PCB that works as an electrode array. This section discusses the blocks, design decisions and progress that were made for each of them.

EDA Block

To detect and monitor EDA, we began by implementing, incorporating, and testing a new circuit for EDA measurements which promises simpler signal processing for feature extraction from the data while also offering simple circuitry. This new method was detailed in the paper "Electrodermal Activity Sensor for Classification of Calm/Distress Condition" published by Universidad de Castilla-La Mancha in Spain [8]. The circuit for this EDA sensor, as cited in the paper, looks as in Figure 7.

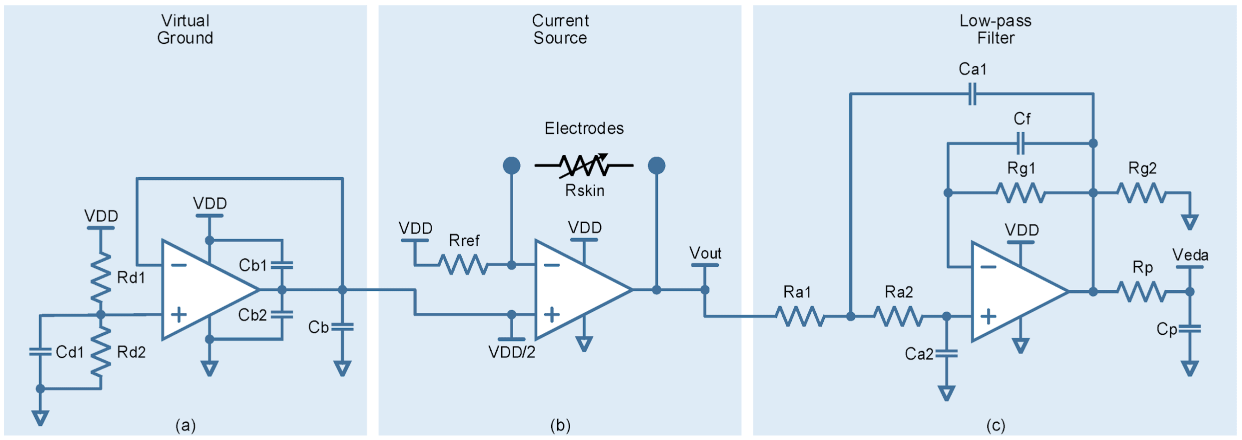


FIGURE 7: EDA SENSOR CIRCUIT, FROM [8]

The circuit consists of three distinct stages. The first stage is an operational amplifier (op-amp) used to isolate a voltage divider, creating what the paper calls a "virtual ground." This signal, labeled as VDD/2 in figure 1, is the reference voltage for the next part of the circuit, which is the sensor. The sensor stage, called "current source," is measuring the EDA signal using a current source by connecting two electrodes to the skin, which are connected to the negative input and the output of the op-amp, enabling the op-amp to generate a current which is injected into the skin and fed-back into the negative input terminal. R_{ref} is used to limit the current going through the skin of the wearer. The output signal of this stage is referenced as V_{out}. The next and last stage of the circuit is an elaborate low-pass filter to clean up the signal of the many potential sources of noise.

This circuit was first built and tested on a breadboard, and then integrated into an existing initial prototype PCB, and tested in that context as well. This testing was important because it required ensuring that the circuit worked under our use conditions which included isolated power and ground rails to minimize noise and interference from other parts of the PCB dealing with other signals. To perform the isolation, an opto-isolation chip (HCNR200-300E) was integrated, which uses LEDs and light sensors to replicate the voltage on the LED side of the circuit without allowing any of the varying ground and power levels (which are isolated for the EDA circuit, and therefore not necessarily equal to the common power and ground) to interfere with it by measuring the optical brightness of that LED and using that sensed value.

Another aspect of testing that took place was optimizing the value for the virtual ground and inspecting the behavior of values lower and higher than VDD/2. This is to optimize accuracy because the final signal is going to be read by an ADC (analog to digital converter) on the microcontroller, which has set resolution and input range. By examining and tweaking the final output range, focusing on the extreme values, in volts (given what we expect the range of inputs of human skin resistances to be), we can make sure that out output range utilizes the ADC range maximally, enabling us maximum precision by mapping each ADC bit to a smaller voltage unit, without saturating the ADC. I tried out different values for the voltage divider and calculated their resolution to optimize the signal read by the microcontroller (to perform the signal processing on). The software to process this signal and classify the results is yet to be done.

EEG/EMG/EOG Sensing

All three signals of EEG, EMG, and EOG were collected in the same hardware unit using a device from Texas Instruments which is specifically made for collecting and measuring biological signals, the ADS1299. This is a very powerful device (and the most costly component in our setup) that enables the measurement of eight different biopotential signal channels simultaneously, and apply independent gains (between 1 and 24 as follows: 1, 2, 4, 6, 8, 12, 24) and different modes of operation on each of them. For example, it allows each channel to have both a positive (P) and a negative (N) lead, and it also allows you to use only one channel lead, and use a "bias" probe as a reference for the signal, which can be used for as many channels as the user desires. Additionally, it implements a Right-Leg-Drive (RLD or DRL, for Driven-Right-Leg) circuit which senses what is called "common-mode interference," which is noise in the body that could interfere with the signals of interest, and uses an op-amp to subtract those noisy signals from the bio-signals of interest. To enable this feature for a specific channel, you must connect SRB2 for that channel. Even more usefully, the chip has a very customizable internal circuit to detect whether the probes and leads on the human's body are loose or completely off, given false or inaccurate bio-signals. This feature is referred to as "lead-off detection." This device also offers two modes of communication: both I2C and SPI; we use SPI in this project. The device also offers a "continuous conversion" mode which pulls an interrupt pin low when the data is ready, and a "single shot" mode in which you poll the device for data; we are using the single shot mode due to some wiring issues in the first version, but plan on switching to continuous conversion mode in the future versions.

We are using the eight channels of the ADS1299 as follows: one EOG channel, three EMG channels, and four EEG channels. Each EOG and EMG channel uses two leads, both the positive and negative probes of a channel, which are connected to two sides of a muscle to measure the potential across it. Using this setup, which is referred to as a sequential montage, the ADS1299 measures the potential difference across the pair of electrodes which should be placed across a muscle; one on each end. We use a gain of 2 or 4 for each of these channels as they are fairly visible already due to their fairly large magnitudes. For EEG, we only use one channel probe (either the positive or the negative), and connect the other one to the bias probe, measuring the EEG signal with respect to a single reference electrode. This setup is called a referential montage, and it helps eliminate noise and clean up the EEG signals which are much smaller in magnitude compared to other biopotential signals. Additionally, we use higher gains of 12 or 24 for these electrodes due to the miniscule size of the EEG signals. We also connect these channels to SRB2 to enable the RLD circuit to remove as much noise as possible from these sensitive and minute signals. To add even more accuracy, we use two types of electrodes for EEG detection: passive (such as the ones used for EMG and EOG), and active, which has built-in circuitry to actively magnify the signal as it is sensed. For the active electrode channels, we can use a smaller gain of as low as 1.

To begin testing this set up, we first took advantage of a built-in functionality in the ADS1299: generating test signals. This is another feature of the device where it internally generates square waves at selectable frequency and amplitude and feeds that signal into the channels that are enabled to be tested. This is a great feature to test the initialization routine of the chip such as wiring, power levels, register settings, and data out package reception.

The next step in the testing process for this device was to inject external signals into each of the channels and be able to receive the correct data. For this, we used a function generator to produce arbitrary waveforms, and used a simple attenuation circuit to make the signal even smaller (since we want to test the ability of the device to detect miniscule signals), and tested each of the channels' ability to transfer those arbitrary waveforms (sine, sawtooth, etc.) at diminishing amplitudes (1-0.01 volts) and varying frequencies (10-1000 Hz).

PPG/Temperature Sensor

For our PPG sensor, we use the MAX30105 which is a particle and proximity sensor with PPG and temperature sensing. This device is placed in its own mini PCB board and connected to the rest of the system using a 4-pin JST connector to connect the I2C and power lines to the board. This separation from the main and even secondary PCBs allows the temperature sensor to be more accurate in representing the temperature of the patient, since it is not skewed by any nearby hardware. It Also enables the PPG sensor to have reduced noise data as it uses a red LED and infrared sensor to measure the pulses in the veins, and by having it live in a separate unit (its own mini PCB) it better ensures a more secure connection to the patient's skin, acting as a probe on their forehead. To test this unit, it was only necessary to collect the data and graph it, to see if the heart beats are visible in the graph. The PPG sensor also has a data ready interrupt pin that is not in use currently.

IMU Unit

The IMU being used for this project is the MPU6050, which is a three-axis accelerometer and gyroscope (and no magnetometer) which communicates through I2C. Because of its lack of magnetometer, and because it was wired improperly (sadly due to a funky Eagle library) in the first version of the PCB design, there were some considerations of replacing the MPU6050 for a complete triaxial one with an accelerometer, gyroscope and magnetometer. Upon doing some research and reading of the literature in the topic, we determined that the accelerometer and gyroscope should be enough for the kinds of analyses we aim to do with our device for sleep studies. This sensor also has a data ready interrupt pin that we are not currently using. To test this device's wiring (and how we revealed the wiring error in the first PCB design), we attempt to establish a connection with the device via I2C and take a look at the data we receive. In the case of the incorrect wiring, the device was never found, because it was improperly wired and thus disconnected from power and ground, and the data and clock lines were all jumbled up.

Communication and Networking

To ensure the project meets state-of-the-art performance, the device need not only collect data and process it, but it also has to export it and send it to a server or device which collects and stores all the sensor data. We incorporated a Bluetooth/WiFi chip to keep our options open in terms of which method of communication we prefer to use based on which could work better and be faster or more reliable. We use the ublox NINA-W102 which is a chip that is rated safe to be in close proximity to the human body, which our device will be.

Security

Due to the nature of the project which involves personal data, security is a serious and important consideration. To this end, we integrated a specialized security chip into the design of the device. We selected the ATECC508A which enables us to encrypt any data using I2C before sending or sharing it with the world outside of the device's PCB. Although this part is important, it is not a priority in terms of development and testing of the device at its current state, therefore there has been no testing done or software written for this cryptographic device yet.

Microcontroller (MCU)

The device has a single microcontroller, in which all the sensor and signal data is collected and processed before being sent over the network. The MCU in the device is the ARM M0, a 32-bit ATSAMD21G which communicates with the WiFi/Bluetooth module, NINA-W102 using SPI. The selection of the microcontroller was due to the design decision to make the whole project, hardware and software, open-source and accessible, without increasing the price too much: Arduino is an open-source and affordable resource, and offers a familiar IDE and programming interface for many people.

FIRMWARE

Each of the types of data that are collected in hardware had to be processed and cleaned up differently in order to be viewable and interpretable. In this section, I will detail these procedures that are in place to receive and process the data from the sensors to make it more visualizable.

But before I do so, we must discuss a few important aspects of data collection that are at play in this system: data size, data rate, and data conversion.

DATA ASPECTS

Data Size

It is important to take into consideration the size of our data in bits or bytes for two main reasons. The first one is that each device could give data in different sizes and formats, which could affect their precision and how we can interpret them and correctly read them. The second is that our data size affects the total size of the data packets that we send over the WiFi network. The Arduino WiFiNINA library seems to have a limit to the size of the packet it can send as one packet without splitting it, and the size of each of our data points plays an important role in the creation and packing of this data packet.

Data Rate

Data rate is an essential variable in our system. Clearly, it matters because we aim to make our device fast and efficient. More importantly, however, is its effect on the data which we collect, which is twofold. Firstly, for each type of signal we collect, there is a range of frequencies which that biological signal tends to remain within. For us to be able to capture that signal accurately, our data collection rate must be at least twice the highest frequency (or the bandwidth) of the signal we are interested in collecting, in theory. For example, if signal S has a nominal bandwidth of f, but its frequency can go up to f_{max}, then the theoretical minimum rate with which we can sample signal S would be $2 \times f_{max}$. This is called the Nyquist Rate, and it specifies this minimum value as a prerequisite for being able to reproduce the signal exactly identically to the original signal, constituting alias-free sampling. Secondly, we apply digital filters to clean up and remove unwanted noise in our collected signals. The way these digital filters work depends on the data rate at which the data was collected, in order to allow some frequencies to pass the filter (within the bandpass, BP), and some frequencies blocked out of the filter (within the band stop, BS), as desired.

These frequencies and the accuracy of the points at which the filter allows and stops allowing data to pass through depends heavily on the accuracy and steadiness of our data rate.

Data Conversion

Most of the data collected from the sensors and hardware blocks is received through SPI or I2C, for which it is more efficient if it is packed in ADC counts or non-standard sizes of data. Therefore, upon receiving the data, some work has to be done in order to make sure that the data makes sense and is usable and readable for our final use cases. This could include conversions from ADC counts to real, metric units, or simply sign-extending the data to a standard size, or even both.

PHYSIOLOGICAL & SENSOR SIGNAL DATA

In this section, I will discuss how each aspect of data was handled for each type of signal received from hardware blocks.

ADS1299 Data (EMG/EOG/EEG Signals)

The ADS1299 is the single device which gives us the most data in our device, eight data points to be precise. Moreover, the kind of data that we collect from it is the most sensitive in terms of timing and frequency of collection. This is because we use it to sense EEG, EMG, and EOG signals, which require sensitive circuitry in order to detect and measure. Having a sensitive circuit naturally exposes our target signals to electromagnetic noise from the environment and even from other parts of the circuit. For this reason, all of our signals from the ADS1299 need to be filtered. The most prominent electromagnetic environmental noise is the power line AC frequency, of 60 Hz in the U.S.A, and 50 Hz in most other countries. Therefore, the first filter we run the ADS1299 data through is a digital notch (band stop) filter at 60 Hz. The second filter we run each of the signals through is a band pass filter to allow in only the range of frequencies in which that signal lives into the filter, discarding all other frequencies from our final signal. Because of the sensitivity of the signals and their frequency ranges, and because we run them through two different digital filters which rely on a constant and accurate data collection rate, we allow the ADS1299 to lead and determine the frequency of data collection for all the other devices and signals in the system of the device.

Data Rate

The data rate of the ADS1299 is programmable, anywhere between 250 samples per second (SPS), and 16,000 SPS. There are two main criteria for selecting the data rate of the ADS1299: the frequency range for each of the signals it collects, and the frequency ranges for the other signals to be collected in other hardware blocks in the system. Firstly, the frequency of the signals collected in the ADS1299. As mentioned in the Data Rate section above, the theoretical minimum sampling rate for a signal is double the maximum frequency for that signal. In practice, however, we found that going a bit below that minimum still works quite well. This is likely because the theoretical value is tied to the ability to recreate an identical signal to the input signal. Since we are not concerned with re-forming the same signals, we have found that the features and shapes of the signals can be faithfully collected by using a data rate lower than the Nyquist Rate. Secondly, the data rate of the ADS1299 depends on the frequencies of the other signals to be collected in the system using other IC devices. This is because I made the decision to use the ADS1299 as the data rate leader, for aforementioned reasons. Therefore, the other signals and their frequency ranges have to be taken into account when making decisions regarding the ADS1299 data rate. Because of these restrictions, the acceptable data ranges for the ADS1299 go from a minimum of 500 SPS and up.

The data rate that is programmed into the ADS1299 determines the rate at which it samples, converts, and signals that the data is ready to be read, or sends the data over, depending on the settings for receiving the data. As previously mentioned in the Hardware section, the ADS1299 can be set either to continuous conversion mode or single shot mode. In continuous conversion mode, the data ready pin would be pulled low at the specified rate as the data gets sampled and converted. In the single shot mode, the device would continue to sample and convert at the given rate, but would not incur interrupts or pull the data ready pin low until polled for the data, in which case it would pull the data ready pin low and proceed to send the requested data.

Data Conversion

Each data packet we receive from the device includes two main components. First is information on which leads might be off of the patient’s body (a.k.a. lead-off status) This section of the packet would only carry meaningful information if the device configured to keep track of the leads and report the status of their contact with the patient’s body. Following the lead-off status is the converted data from all eight input channels of the device. The size of the status portion, in addition to all the channel data combined is 216 Bits (54 Bytes), or 24 Bits (three Bytes) for each of the 9 data points. The ADS1299 sends each of these data points MSB (most significant bit) first, and LSB (least significant bit) last, as shown in the figure below. The data for each of the channels is sent in the form of ADC counts, which need to be converted to volts.

To convert from ADS counts to volts, we first need to determine the value of one ADC count Bit in Volts. That turns out to be a simple ratio: the full-scale range of volts, divided by the full-scale range of bits. Since the data is given in 24 bits, that makes the range of bits equal to 2^{24}. As to the range in Volts, the maximum voltage we can measure is fed to the IC, V_{REF}. There is one more variable, however. The gain of each channel is adjustable, which limits the original scale of voltage to V_{REF}/Gain, on a per channel basis. Finally, because the voltage can range from $+V_{REF}$ to $-V_{REF}$, the full-scale range in Volts is actually $2\times(V_{REF}/\text{Gain})$. The result is then the ratio = $(2\times(V_{REF}/\text{Gain})$ Volts) $/(2^{24}$ Bits); that is the value of one LSB (or one Bit). This means that to convert the received ADC counts value to volts, we simply multiply the received value by the ratio we just derived, and get the voltage sensed at that channel, given that we know our V_{REF} and the gain we used for that specific channel.

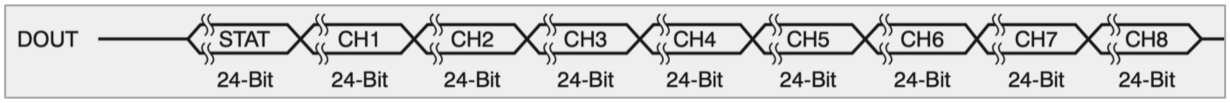


FIGURE 8: THE DATA OUTPUT PACKET FROM THE ADS1299, DISSECTED, [9]

Now, the data conversion story is not quite over yet. The voltage is encoded in 24 bits of what's known as binary two's complement, which is an algorithm for representing positive and negative numbers in binary format. This is actually the format most computers and programming languages use and understand, except for one important detail: the data type. Most computers and programming languages only use data types whose size is a power of two: two, four, eight, sixteen, thirty-two, and sixty-four Bits of size. However, the data we receive from this device is 24 Bits, three Bytes. This means we have to sign-extend the data to a standard size, of which the closest one is four Bytes, or 32 Bits. In order to explain that, I'll first briefly explain how binary two's complement encoding works.

In Two's Complement, the MSB encodes whether the number is negative. If the MSB is set, meaning it's a one, then the number that is represented is negative. If it is not set, or cleared, meaning it's a zero, then the number represented is positive. Each set bit, or one, at a specific index in the binary representation corresponds to the addition of the value $2^{\texttt{bit\_index}}$, including the MSB, which always corresponds to the addition of the negative of that value. For example: `0b01` is the largest positive number that can be represented with two-bit two's complement: just one, and `0b10` is the largest negative number that can be represented in the same system, negative two. Only two more numbers can be represented in between: `0b00` for zero, and `0b11` for negative one. As you can see, Two's Complement is always asymmetric- enabling the representation of one additional number in the negative domain than in the positive domain.

To sign-extend a number is to represent the same number in binary two's complement in more bits than the original representation. In our case, we want to sign extend a value from 24 to 32 Bits, adding 8 Bits, or a whole Byte. The simplest way to do that is by copying the MSB from the original representation and filling the additional bits in the new representation, on the MSB side of the number, with the same value. More concretely, if the MSB is zero, we know the number is positive, so we can duplicate the MSB of zero over eight more times at the MSB side of the number, producing an unchanged positive number, just padded with more zeros. On the flip side, if we have a negative number, with an MSB of one. This turns out to be slightly more difficult to explain, but when you duplicate that one over the most significant Byte of a 32 Bit value, you wind up taking the value of the original number as though it was a direct representation and not a two's complement value, and adding the values corresponding to the first seven Bits added, and subtracting the value corresponding to the MSB (eighth Bit) added, which is the largest one. This preserves its negative sign and value (you can prove it to yourself by doing some examples!).

There is no current Arduino library for the ADS1299, so I had to build a lot of the code from scratch using the elaborate (and very often confusing) datasheet. In order to test the ADS1299 register settings, ADC counts to Volts conversion, and sign-extension algorithm, I set up the internal test signal generation feature and graphed the results. The resulting square-wave graph is shown in the figure below, which passes through zero, showing both positive and negative numbers, at the expected amplitude and frequency (yippee!).

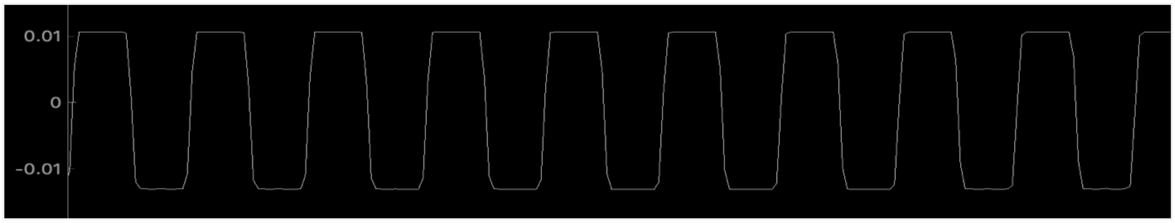


FIGURE 9: ADS1299 INTERNALLY GENERATED TEST SIGNAL UPON REQUIRED CONVERSIONS

Because there is currently no Arduino Library for the ADS1299, I have had to develop, debug, and test firmware from scratch to do everything from the simplest reading and setting registers, to understanding the register settings that would work for each use case and application. A more detailed description of these development efforts is in a section to follow, "Development & Debugging."

Now that we have discussed the device in general, and all the common processes pertaining to all the data collected from it, we can move on to specifics, depending on the signal type. In the next two sub-sections, I will discuss the specific biological signals collected in the ADS1299, and the difference between them in terms of the programming of the firmware of the device.

EMG & EOG Signals

The range of frequencies in which EMG activity lives is wide and uncertain; it starts at 5 or 10 Hz, and ends anywhere between 250 and 500 Hz, but even up to 3000 in some cases [10]. The bandwidth of the EOG signal ranges between DC-100 Hz [11]

To program the ADS1299 to detect disconnected probes (activate lead-off detection), we first select the power type to use: AC or DC. We use DC for simplicity, since AC has a frequency which could, based on the frequency, interfere with our signals of interest. Then, we decide on the comparator thresholds. Lead off detection works by injecting a current or voltage, and monitoring how much of it reaches the other side; if there is no connection to the other side, then the two ends hit the rails, power and ground [12]. The comparator thresholds set the limit of how close the channel leads have to get to the rails in order to be considered "off". This setup is for all channels.

Then, we set the bits corresponding to the channels for EMG and EOG in `LOFF_SENSEP` and `LOFF_SENSN`, indicating that the lead-off for these channels should be detected for both ends of the probes: positive and negative.

EEG Signal

This signal has multiple types: Alpha, Beta, Delta, Theta, Gamma, and Mu, each of which lives in its own range of frequencies. There is a tremendous amount of variability and uncertainty in the range of frequencies in each band, the figure below depicts the typical ranges based on the number of publications pertaining to the signal type, and the extremes of those ranges [13].

For EEG channels, lead-off is a bit more complex since we only use the negative terminal of each channel for the signal. The return path for the electrical signals is through the bias probs, which the ADS1299 does check for, if you set it to. Currently, I have not managed to get lead-off detection to work for EEG signals, but the current set up also involves flipping the lead-off input signal through the channel terminals, entering at the N terminal instead of the P (which we are not using).

	% of Publications	Typical range (Hz)	Minimum start value (Hz)	Maximum end value (Hz)
Delta	70	1.3-3.5	0	6
Theta	84	4-7.5	2.5	8
Alpha	85	8-13	6	14
Beta	80	12.5-30	12	50
Gamma	18	30-40	20	100

FIGURE 10: TABLE DETAILING THE RANGES OF EACH EEG BAND, [13]

EDA Data

The EDA data is collected through a built-in Arduino ADC, on demand, meaning, the value at the Analog pin which corresponds to the EDA data can be sampled and read at any point in time, and the Arduino bootloader would read the Analog signal at that pin, then feed it to an ADC, produce a digital value corresponding to the original analog voltage, and return it to the firmware code which called the analog-pin-read function.

Data Rate

Because the EDA signals tend to have lower frequencies (ranging 1-10 Hz) [14], and because the data is fairly erratic and noisy, the best processing for this data is to smooth it via averaging every set number of samples. Given the frequency with which we collect the entire set of signals in the packet, we can afford to average every ten EDA data samples to produce one mean value which we actually use and send in our data packet. Performing this averaging technique also increases the effective resolution of our ADC: when you take multiple integers (e.g. 2 and 3) and average them, you could get a non-integer average (2+3 = 5, 5/2 = 2.5), adding a decimal point of precision to the value that is going to be used in the rest of the system and analysis. This technique is referred to as oversampling and averaging [15]. Oversampling because we take more samples than we use, and we average a certain chunk of those samples to produce data at a lower sample.

The default setting for the Arduino ADC (set in its bootloader), is 10-bits of resolution, and the "prescaler" value, which determines the speed of the conversion (speed equals clock speed divided by the pre-scaler value) is set to a default of 512. When I ran into issues with speed of data processing in the firmware, I realized that the default was that slow value, and changed the prescaler value to 16, multiplying the speed of the conversion by 32. I also changed the resolution of the ADS from ten bits to twelve, increasing the precision of our ADC conversions and thereby our measurements of the EDA signal. Both of these changes are done by modifying the `ADC->CTRLB.reg` variable in the Arduino SAMD bootloader in the line: `ADC->CTRLB.reg = ADC_CTRLB_PRESCALER_DIV16 | ADC_CTRLB_RESSEL_12BIT`.

Even still, the EDA measurement was still slow compared to the ADS1299 firmware code. That fact, in addition to that the EDA signal is already at slower frequencies than the other signals, lead me to decide to only sample the EDA every once in a while. I experimented with different frequencies, and landed on sampling the EDA signal at every ten samples of the ADS1299. Both of the above techniques combined (assuming a negligible conversion delay after changing the prescaler to 16), in addition to this last note, would have our EDA signal being sampled at $f_{EDA} = f_{ADS1299}/10/10 = f_{ADS1299}/100$. The first division by ten is for the sampling every ten ADS1299 samples, and the second one is from oversampling and averaging.

Data Conversion

The 12-bit ADC count value we get then needs to be converted to a useful unit. The most popular way to measure EDA is by measuring the skin conductance (G) in micro-Siemens (μS). The relationship between resistance (R_{skin}) in Ohms (Ω), which is the most direct value we can get by using the voltage level measured and encoded by the ADC, and skin conductance (G_{skin}) is simply reciprocal. So, the conversion is as follows: $G_{skin} = 1/R_{skin}$ [16]. Now that we know how to get conductance from resistance, we can talk about how to get resistance from ADC counts.

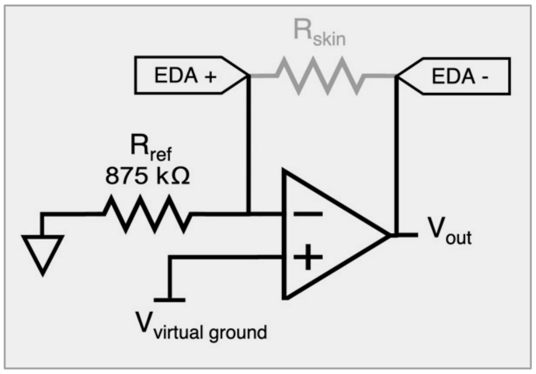


FIGURE 11: ABRIDGED EDA CIRCUIT- EDA PROBES. ADAPTED FROM BUT NOT IDENTICAL TO PART OF THE CIRCUIT IN [8]

R_{skin} is the resistance of the human as measured between the two EDA probes (plus, and minus) placed on the human body, as shown in the figure to the left. We can measure this value by first converting the ADC counts to voltage. This is the voltage measured at the output of the op-amp, or V_{out}. We also know that, by the ideal op-amp assumption, the voltage at the two terminals must be equal. This means that the voltage across R_{ref} is known and is equal to $V_{virtualground}$. We can use this information to find the current, i, across R_{ref}, which is the same current going through R_{skin}. Now we know the voltage across R_{skin}, V_{skin} = `ADC counts (bits) × 3.3 (volts) /2`^{12} (bits), and the current across it, $i = V_{virtual}$

ground / R_{ref}. We can use those two pieces of information to calculate $R_{skin} = V_{skin}/i$, and from there take the reciprocal to find the skin conductance.

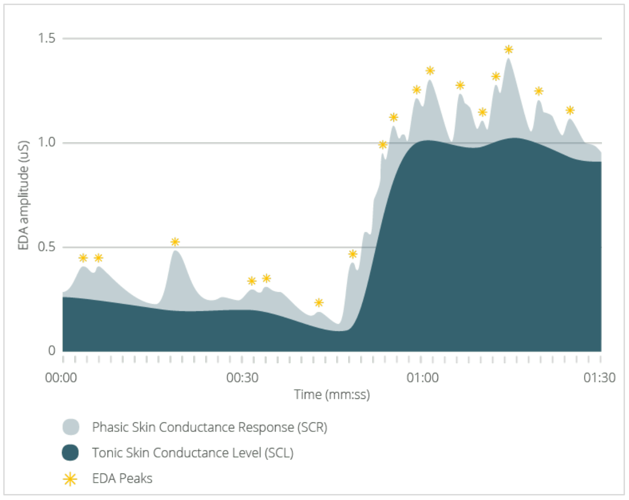


FIGURE 12: SCR VS SCL COMPONENTS OF EDA, [14]

The EDA signal is comprised of two main components: skin conductance response (SCR), and skin conductance level (SCL). The SCR can be considered the micro changes in the EDA signal, the smaller and faster disturbances and waveforms that appear superimposed on top of a more constant "level," which is the SCL. So the SCL is the component which can be considered as the macro view of the EDA signal without paying much mind to the smaller changes in what, from a zoomed-out view, seems like a smooth line. SCL is the slower component and one which can resemble the DC (value of offset) component of the signal. SCR levels can reveal information on the subject's emotional arousal, whereas SCL is more generally an indication of the sympathetic nervous system's arousal [17]. A depiction of the distinction between SCL and SCR in the EDA signal can be seen in the figure to the left. This analysis and separation of SCL and SCR is yet to be performed in our system and our device.

PPG / Temperature Data

The MAX30105 sensor lives on its own hard PCB connected to the flexible PCB in order to optimize the quality of the data we receive from it. This sensor is equipped with an interrupt pin which we do not currently utilize (due to the decision that the ADS1299 would lead the data rate and determine the frequency of collection of the other signals), and two I2C lines which we use to communicate with the device.

There is a SparkFun Arduino library for the MAX30105, so instead of writing my own API to interface with it from scratch, we decided to go with the existing library. The library is called SparkFun MAX3010x Pulse and Proximity Sensor Library and can be used by importing `MAX30105.h` [18]. First, we must create and initialize the class, using the line: `MAX30105 particleSensor_name`. The library allows us to set the sampling rate, the data averaging rate, an even the power consumption level. We can use the library's API to set up the device first using the function call: `particleSensor_name.setup(powerLevel, sampleAverage, ledMode,` `sampleRate, pulseWidth, adcRange)`. Then, we can retrieve both PPG and temperature data from the device, using the following two function calls: `particleSensor_name.getIR()`, and `particleSensor_name.readTemperature()`, respectively.

Data Rate

Because I noticed a clear and drastic slow-down in the data collection rate of the device (even when I setup the MAX30105 to have a high data rate), I had to investigate the source of this slow-down. I revised the code repeatedly and narrowed down the source of the issue to be the library implementation. When I took a deeper look at the library code, I realize that both the temperature and IR retrieval functions include some sort of (long) wait or delay. The `readTemperature()` function includes a register-write to request data before a 100 millisecond wait as the device awaits the response for its request. The `getIR()` function includes an up to 250 millisecond wait for new data to appear at the IR sense output FIFO, where it just repeatedly checks for the availability of new data.

So, I modified the library by first removing each of the delays in both of the functions and taking a look at the data they were returning. Both of the functions were returning that there was no data available more than fifty percent of the time. Because of this, I adjusted the wait time to a middle-ground compromise value and observed that many of the data points were actually available. At this point, I decided to make the functions more performant based on the use-case of the consumer code, so I changed the function signatures. For the `readTemperature()` function, I added a `requestTemperature()` function that performs the requesting of the temperature reading with no wait, and modified `readTemperature()` to simply check the availability of a temperature reading without waiting. For the `getIR()` function, I modified it to take as input the number of milliseconds the consumer wants to wait for data to be available, `getIR(wait_ms)`. Then, in the firmware code, I first call `requestTemperature()`, then `getIR(1)` (which happens to be a very good value and we can still see the heart beats and waveforms with great accuracy), and finally I call `readTemperature()`, indirectly inserting a wait in the temperature collection routine.

Data Conversion

The Sparkfun library collects the PPG data and returns it as a 32 Bit integer (even though according to the range of values that ADC range can be set to, it cannot be larger than a 16 Bit integer). The PPG data units do not matter as what we care about. From that signal is the shape of it, in order to be able to detect heart beats and calculate heart rates, and therefore reveal any irregularities in the cardiovascular system of the patient. One important aspect of the measurement of PPG is that having very good contact with the device is essential in order to get accurate data. Luckily, the MAX30105 is a particle and proximity sensor as well, and once the finger (or hand, or, in our case, forehead) loses good contact with the sensor, it automatically switches to proximity sensing instead, which produces numbers, returned by the `getIR()` function which are multiple orders of magnitude smaller than the PPG data. This way, we can detect if the patient has good contact with the sensor, and therefore know if our data is accurate.

The Temperature received using the library function `getTemperature()` already has appropriate units, degrees Celsius, received as a Float type (four Bytes). So, no processing needs to be done to this data, as the library takes care of the conversions.

Temperature change is a very slow process, and heart beat signals (ranging between 60-100 BPM for average resting heart rate in humans [19]) is a relatively slow signal compared to the other vitals and biological signals being sensed in our system, we do not need to sample these signals as frequently as we do the others. For this reason, we only sample this data one out of every 10 times we sample the ADS1299 data, resulting in an effective data rate which is a tenth of the frequency of the ADS1299, for both the PPG and the temperature sensor. The effective data rate for these signals, in practice, is much less regular and more erratic, and often slower. This is because of the reduced wait times in the API functions, and the frequency of availability of data (or lack thereof). This does not affect the quality of our data, as we are able to monitor the availability and discard the data returned when it wasn't available, but it does affect our practical sampling rate for these signals.

IMU Data

There are numerous Arduino libraries for the MPU6050, which is the IMU unit we decided to go with for our device. The library we chose was the MPU6050 by the Electronic Cats and can be used by importing `MPU6050.h` [20]. Before using the library, we must first create the class using the line: `MPU6050 accelgyro`, and initialize it using: `accelgyro.initialize()`. We can optionally adjust other parameters including data rate and data value ranges using the appropriate API functions, but the default settings work well for us, so we do not utilize those functions. The library has different API functions to retrieve each of the acceleration and gyroscope values in each of the three axes. But it also has the handy function to get all of them at the same time with one function call: `accelgyro.getMotion6(ax, ay, az, gx, gy, gz)`.

Data Conversion

Each of the data points returned by the library is a 16-bit two's complement value raw ADC count value. In order to get any useful units out of this value once fetched, a conversion must take place. For the acceleration, the default settings have a full-scale range of $\pm 2g$ ($4g$), and since the ADC has a resolution of 16 bits. The conversion to acceleration units is then $a_{val} \times 4g/2^{16} = a_{val} \times g/16384$ (m/s^2), where a is the acceleration and g is the earth's gravity. For the gyroscope data, the default settings have a full-scale range of $\pm 250°/s$ ($500°/s$). The conversion to gyroscope units is then $g_{val} \times 500°/s/2^{16} = g_{val}/131$ ($°/s$) [21].

These conversions convert a 16 Bit integer value to a 32 Bit float value, which presents a qualm. It's a tradeoff between having standard units and a bigger WiFi packet, versus having a (potentially marginally) faster WiFi packet. For now, I have elaborate comments explaining this conversion in detail in the code, and commented-out code in place to perform the conversions, because I do not believe we currently need the IMU data in general to be in any specific units. No matter the units, we are able to detect the motion, the type of movement, and the axis, without needing the metric units. Still, I left the code in there in case that situation changes and a conversion to metric units becomes necessary.

Data Rate

Movement and motion, especially during sleep, are fairly slow processes, especially compared to the EEG, EMG, and EOG waves collected on the ADS1299. For this reason, this is yet another signal that can be sampled and collected at a fraction of the frequency as the ADS1299. Specifically, like the others, it is sampled at an effective rate of a tenth of the frequency of the ADS1299 sampling rate.

WIFI PACKET SPECIFICATIONS

There is an Arduino library for using the WiFi/Bluetooth module in our device, the ublox NINA-W102, which is used by including `WiFiNINA.h` [22]. Because of this existence of this library, the project was developed to send data over WiFi, for simplicity and speed of development. However, the prospect of using Bluetooth for communications and data transfer is still viable and possibly desirable. For now, let's discuss the details of the data sent from the device as a WiFi packet over to the software for storage and further processing.

WiFi Packet Anatomy

All the aforementioned data and signals needs to be combined into the WiFi packet to be sent over to the software. A full data packet has four EEG data points, three EMG data points, one EOG data point, three acceleration data points, three gyroscope data points, one EDA data point, one heart data point, one temperature data point, one serial number, one valid array, and one time stamp, for a total of 20 data points, in the order shown in the table below. The number and sizes of all the elements total to 68 Bytes for the whole packet.

Sequential order	Signal	Size (Bytes)	Byte at which it starts
1	Serial packet number	4	0
2	Valid array	4	4
3	EMG	4	8
4	EMG	4	12
5	EOG	4	16
6	EMG	4	20
7	EEG	4	24
8	EEG	4	28
9	EEG	4	32
10	EEG	4	36
11	Acceleration x	2	40
12	Acceleration y	2	42
13	Acceleration z	2	44
14	Gyroscope x	2	46
15	Gyroscope y	2	48
16	Gyroscope z	2	50
17	EDA	4	52
18	Temperature	4	56
19	PPG	4	60
20	Time stamp	4	64

FIGURE 13: TABLE DETAILING THE COMPONENTS OF A DATA PACKET

It is worth noting here that the IMU data, all six data points, are currently only two Bytes each because I do not perform the conversion as I deemed it currently unnecessary. However, if the conversion is to be performed and the converted data to be packed into the data packet instead, each of the IMU data points would be four Bytes instead of two, which would increase the total size of the packet by 12 Bytes, making it 80 Bytes instead of 68.

Valid Array

Because not all the data is collected at the same (highest) rate, there are some packets where the only valid data we send is the ADS1299 data, as well as the obvious serial packet number, and time stamps. Some data is sent less frequently and is therefore invalid much of the time as it was not collected. Some data is truly invalid because of a system issue, such as an ADS1299 having a lead-off detected, or the MAX30105 revealing a lack of a secure connection to the patient. For these reasons, I've introduced a component in the packet called the valid array, which can be viewed as an array of bits. It has one bit mapped to each item in the packet. The way it works is this: if the bit which maps to a specific element is set (is one), then that data point is not valid. Calling it a valid array might be a misnomer then, since what it really is instead is an invalid array. I opted not to call it an "invalid array" in order to combat the potential confusion of the data which the element contains being invalid (as though it is the array which is invalid itself). This array can be used as described to recognize which data points are valid and should be kept, processed, stored, graphed, filtered, etc., and which data was not collected in this packet and should be ignored.

Every time a new packet is created, the field for the valid array is initialized to zero. As the signals are received and inserted into the packet, or not, this array is updated. If a certain signal is not to be collected at this time and this packet will not be updated with it, then the valid array is updated with a one set in the index corresponding to that data point, marking that signal invalid in this packet. Similarly, if a sensor reports that the probe for one or more of its inputs no longer has a reliable connection to the patient's body, then the signal received from that device for that data point is discarded, and the valid array is updated to reflect that by setting the bit corresponding to that device's specific channel to one.

WiFi Packet Send Rate

In order to increase the system's efficiency, we collect multiple packets of data and group them together into one WiFi packet to send over to the software. This method enables us to amortize the cost of preparing, sending, and receiving a WiFi packet through the network over the number of data packets it contains. This aggregate WiFi packet, in our current system, contains 22 data packets. This was a number that is maximized, given the number of data points in our data packet, and therefore a data packet's size in bytes, to the size limit of the WiFi packet that the WiFiNINA library appears to be able to support. So, this number was bigger when our data packet size was smaller. (There doesn't seem to be any documentation the issue of WiFi packets having a maximum send size–this is just a problem I observed repeatedly).

SOFTWARE

Building the software for the system had more than the obvious purpose. Surely, we must provide an interface with which the data collected with the device can be examined, but we also needed a playground, a place where we could firstly, and most importantly, see the data, but also manipulate the data, play around with filtering it, zoom in and out, and make sure we can see, visibly, all the signals of interest. For simplicity and breadth of options, we selected Python to be the language of development for the software. The initial version of this GUI code was created by Guillermo Bernal and Junqing Qiao, and when I joined the project for my MEng, I started modifying and expanding on it—building off of it as the project grew and more progress was made.

DATA VISIBILITY

In order to confirm that our system and device were both working end-to-end, we had to verify that we could see the data from hardware input, through firmware, to software output. To this end, we used an arbitrary function generator to create waveforms as inputs to the ADS1299 and tried to observe and monitor them on the software side.

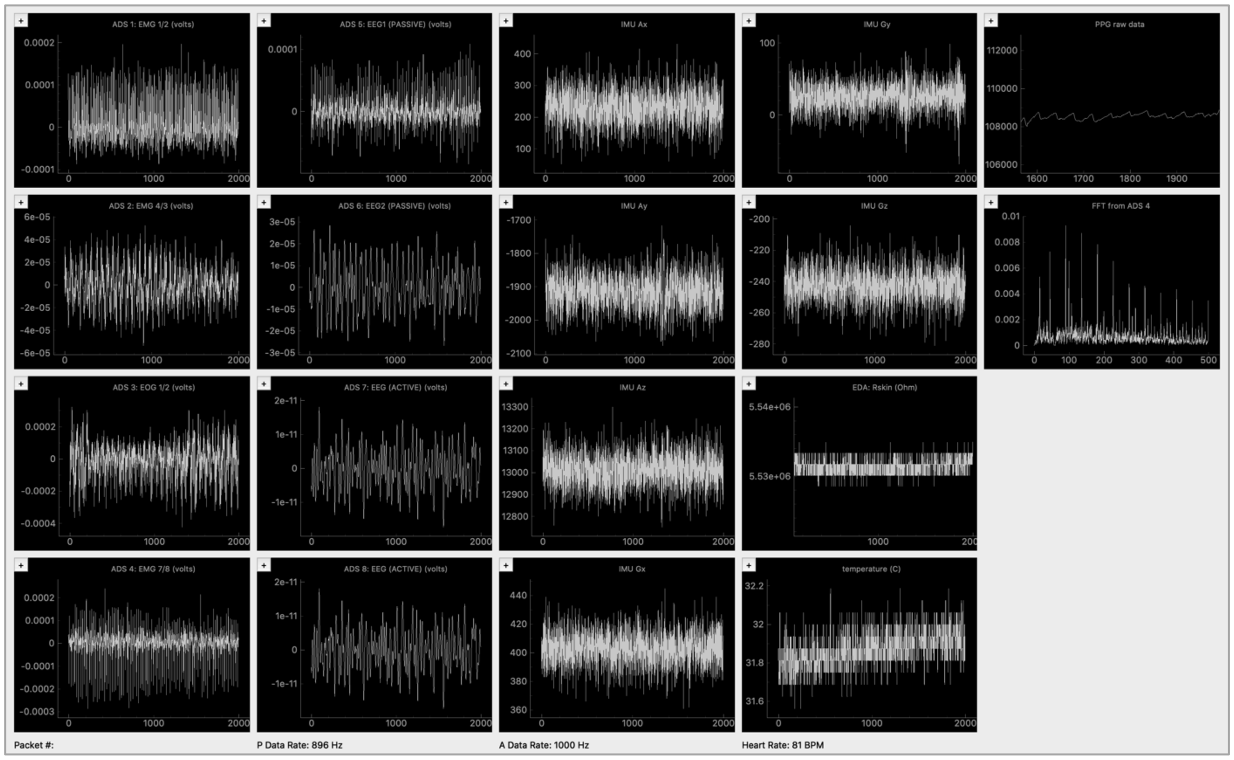


FIGURE 14: THE CURRENT APPEARANCE OF THE GUI INTERFACE

GUI Graphs

For the GUI, we are using the Python libraries `pyqtgraph` and `pyQt5`, whose graphs offer an interactive interface for moving and zooming. The graphs automatically snap to a widget grid, and automatically detect gestures to zoom in and out of each graph, as well as move the plot around within the plot window.

During earlier stage debugging, I had the serial count of the packet graphed, observing and monitoring the slope of the line in that graph. Because the serial number was supposed to be sequential, the slope and linearity of the graph should not change so long as the zoom and window size remained constant. Using this technique, I was able to observe perfect lines in earlier stages of the development of the software. After I included all the sensor data from the system in the data packet, and developed some software algorithms for debugging and analyzing data, however, I noticed that breaks in the line started appearing every once in a while, indicating that the existence of missing packets: there were some packets being dropped, or getting lost in some other way. This, I discovered later, was due to inefficiencies and slow-downs in the software programming.

Now, since I am done with that part of the development, I skip graphing the serial count, and also the valid array and time stamp, so that I can focus on viewing and observing the actual data in the graphs. However, I've programmed this such that skipping the graphs of those data points is an option, in case I need to revisit those graphs for further debugging while continuing the development, or someone using this code in the future needs to do so.

GUI Improvements

In order to improve readability and observability, I added titles to each graph, which became necessary as the size of the data packet grew larger while I integrated more of the sensors in the system. I also modified the titles to include the units in which each signal is graphed.

In order to better observe the data rate, which I previously printed to the Python console, I instead included it as a textbox on the GUI window, more visible and monitorable, updating with the newest data rate as it is calculated.

SIGNAL VISIBILITY

Next, we had to make sure that not only can we see the correct data, but that we can detect and see the signals of interest clearly and reliably.

Digital Filters: Research and Implementation

Since the ADS1299 device is delicate and very sensitive to noise, in order to start visualizing the signals from it, we must first apply some digital filters to remove parts of the signal which we know are caused by environmental noise factors. The biggest and most prominent one is the power line frequency, at 60 Hz, needs to be extracted from every ADS1299 channel. Therefore, a notch filter, band stopping between 55 and 65 Hz, is applied to each of the eight ADS1299 channels. This is the first stage of filtering the biological signal data.

The second stage of filtering the ADS1299 biological signal data is applying a digital band pass filter which filters in the bandwidth for the signal of interest. This is crucial as we might select a range that isn't what might be strictly, or even classically for that matter, considered the typical range for the signal of interest. This is simply because certain bands within that signal type might be more interesting to us given our application is specific to sleep studies, and is not difficult to change on the fly later if we do decide we need a more inclusive range.

To select the target bandwidth for each of the biological signals of interest, we reviewed some literature around these signals and their applicability to sleep studies and consulted a gold-standard ADS1299 development board/kit. Based on this research, the numbers we settled on are as follows: bandpass 10-500 Hz for both EMG and EOG signals, and bandpass 5-50 Hz for EEG signals.

FFT Integration

Some of the ADS1299 data is easier to see than others. For example: EMG is a very strong signal whose magnitude dominates most sources of noise. Conversely, EEG signals have very miniscule amplitudes and are therefore more sensitive and prone to distortion and interference by most sources of noise. This makes most EEG signals very difficult to detect and/or observe. The easiest EEG band to observe is alpha waves, which should be visible when a person closes their eyes. As we were testing signal visibility, this waveform proved even more difficult to observe than we expected. We could not tell whether the data was there. We didn't know if we were zoomed in too much or too far zoomed out, or in which axis: time or amplitude. The signal could have very well been there, but since is it miniscule, and possibly very brief, we just missed it. At the same time, it is also possible that we are not seeing anything because something is wrong with our system or device's setup, and there really is no signal being captured there to be seen.

As a solution to this issue, since we know the target bandwidth of the signal, and we might even know more specifically the sub-bandwidth for the specific band of the signal we are trying to observe, we decided it would be nice if we could view the data in the frequency instead of time domain. The Fast Fourier Transform (FFT) allows us to do just that! It takes as input the waveform data, and extracts the frequencies of the (possibly infinitely, in the case of a square wave) many sine waves that comprise it. Based on the amplitude and how often a certain frequency occurs, the FFT algorithm produces a value mapped to each frequency bin. By looking at the output of an FFT, you can see how much of each frequency bin your waveform data is made up of. This allows us to see the most dominant frequency in a given data set, which is powerful. But if we produce this plot in real-time as we receive and graph the data itself, it also importantly enables us to observe the change in the amount of each frequency our signal contains. This is the key to enabling us to detect biological signals without really seeing them.

The Python implementation of the FFT algorithm I am using is the one in the `numpy` library. Because the FFT algorithm is a computationally intensive algorithm (even if efficiently implemented), I decided to create a separate thread just for the purpose of performing the FFT calculations. This is important because it improves the reaction time of the FFT, enabling us to observe the changes in frequency presence and patterns in the signal more directly and easily. Once this was all done, we tested it by once again trying to see an EEG signal. The easiest one to see is alpha, we know that we expect to see a "spike" in the FFT plot between around 6 and 10 Hz at the time we would expect the alpha wave to appear. When a person closes their eye, the brain generates alpha signals. So, when the subject closed their eyes, surely enough, the FFT graph in the range 6-10 Hz grew a considerable spike, which slowly (because the data is graphed in big batches, and so new data moves out of frame fairly slowly) faded away after the person opened their eyes again. Hoor*eye*!

Heart Rate Algorithm

In order for the PPG data to be more digestible for our purposes, we must process it to calculate the heart rate of the patient, which is a very important measure of their wellbeing and state of rest. I expected there might be a myriad of Python libraries which would do exactly what I needed: take raw PPG data and find the heartrate by finding all the heartbeats and when they occur to calculate the heartrate. After some research I realized that the options are much more limited than I thought. The only library I could find was called `heartpy` [23], and after a few days' worth of futile attempts at using it with the PPG sensor data from our device, I decided to stop trying and give up on using this library.

I realized I had to develop my own real-time heartrate measuring algorithm. To get started with this task, I took some time to observe and understand the signal, and the shape of each heartbeat as it appears in the final GUI graph. I took note of its amplitude and its width, and the variability of each of those things. Since heartrate is a rate, or speed, measured in number of beats per minute, the goal is to count the beats, and measure time correctly. These are the most important building blocks of the algorithm: first is detecting heartbeats accurately and reliably (every heartbeat is detected, and nothing which isn't a heartbeat is mis-detected as one), and second is the accuracy of the timing of the heartbeat. For this reason, I determined it was necessary to add the timestamp field in each packet. With these things in mind, I started developing the algorithm, trying as much as possible to keep it parametrized and adjustable, in order to be able to fine tune it well enough to be applicable and generalizable to a broader range of people's resting heartrates. The current (and for now, final) version of the algorithm is depicted in the figure below.

There are three important arrays in the algorithm: `heart_sig_arr` which keeps all the live PPG signals we are currently looking at and trying to detect a heartbeat in, `heartbeat_ts` which saves the timestamps of the 100 most recent detected heartbeats, and `heartrate_avg`, which stores the 100 most recent "local" measured heartrates, in order to have all the elements be averaged before sharing any of the local data. In Line 1, I am checking whether the PPG data is valid–if it is, we can proceed with the heartrate measuring algorithm, otherwise, since there is no new data to process and incorporate, we do not make any adjustments in the arrays. This is unless (as checked in line 29) we detect that the PPG signal is invalid due to a lost connection to the human (and not just a lack of sampling). If the PPG signal is invalid because the sensor is not connected to the patient (indicated by setting the time stamp bit in the valid array, while the one for PPG is also set), then we reset all the arrays, since we want to be able to start from scratch when a secure connection is re-established to the patient.

If there is valid new PPG data, the real processing happens. Firstly, we take a look at up to the ten most recent PPG signal values received and try to detect a large enough drop in magnitude. Since the heartbeats in PPG signal are roughly triangular shaped, there is a fairly quick and drastic drop (negative edge) at the end of each beat, and that is what we are trying to check for in lines 2-5. If we do not detect a heartbeat after checking the relevant most recent PPG signal values, then all we do is append the newest PPG signal to the `heart_sig_arr` array, and trim all the arrays to the predetermined maximum size, lines 21-28. If there is a heartbeat detected, however, we have some work to do. We are now studying the code lines starting at line 7. Once we detect a heartbeat, we store the timestamp of the packet from which the most recent PPG signal came in the `heartbeat_ts` array, line 7. Then, if we have multiple (more than one) heartbeats detected (line 10), we calculate two heartrates: a local and an average. The local heartrate is calculated by taking the currently detected heartbeat (the last timestamp in the array), and the one before it (the second to last in the array), and differencing them, resulting in the time in milliseconds it took for one heartbeat (line 12). We take the millisecond difference and convert it to seconds (line 13), then to minutes, and dividing one by that (line 14) resulting in the local heartrate in BPM (beats per minute). Once we have the local heartrate, we can append it to the `heartrate_avg` array (line 15) and move on to calculating the average heartrate. The average heartrate is simply calculated by taking the average of all the local heartrates stored in the `heartrate_avg` array (line 16). This is the heartrate value we use, test, and display in the GUI window.

To test this algorithm, I started by ensuring that the beats are all correctly detected, and nothing extraneous was misinterpreted as a beat. I used this method to fine tune the numbers in line 5: the minimum and maximum drops in magnitude for a beat to be considered detected, and the minimum number of PPG samples collected since the previous detected heartbeat for it to be viable timing for another heartbeat. To do this test, I graphed a line superimposed on the PPG signal graph to indicate to me that this specific spot in the signal is where a heartbeat was detected (line 8). Once I was fairly certain that only and all legitimate heartbeats were being detected by the thresholds, I moved on to test the actual heartrate. To test the accuracy of the measured heartrate, I simply compared the result to the displayed heartrate on a smart watch, which, after some time needed for convergence, fairly consistently produced values within about 5 BPM of the smart watch. I did the development by myself, so I did these tests with myself and three other people to make sure it was not too specific or overfitted to me and my body.

```python
if not ((invalid_arr >> i_PPG) & 1):
    ppg_sig = newData[i_PPG]
    l = len(heart_sig_arr);
    for i in range(min(10, l)):
        if heart_sig_arr[l-1-i]-ppg_sig >= 100 and heart_sig_arr[l-1-i]-ppg_sig < 700 and l>20:
            # heartbeat detected!
            heartbeat_ts.append(newData[i_TIM])  # store the time stamp for this beat
            plotBufs[i_PPG - start_idx][-1] *=-1 # mark the spot where the heart beat was detected
            # calculate heart rate
            if len(heartbeat_ts) > 1:
                # delta_ts = time in ms difference between the current and most recent heart beat
                delta_ts = heartbeat_ts[-1] - heartbeat_ts[len(heartbeat_ts)-2]
                delta_sec = delta_ts / 1000
                bpm = 1/(delta_sec/60)              # calculate local heartrate
                heartrate_avg.append(bpm)           # append local heart rate
                bpm = np.average(heartrate_avg)     # calculate average heartrate
                t = "Heart Rate: " + str(int(bpm)) + " BPM"
                dataPlottingWidget.HR.setText(t)
            heart_sig_arr = []
            break
    heart_sig_arr.append(ppg_sig)
    # trim arrays to max lengths
    if len(heart_sig_arr) > 25:
        heart_sig_arr = heart_sig_arr[1:]
    if len(heartbeat_ts) > 100:
        heartbeat_ts = heartbeat_ts[1:]
    if len(heartrate_avg) > 100:
        heartrate_avg = heartrate_avg[1:]
elif ((invalid_arr >> i_TIM) & 1):
    # reset all data: no contact with ppg sensor
    heart_sig_arr = []
    heartbeat_ts = []
    heartrate_avg = []
```

FIGURE 15: HEARTRATE MEASURING ALGORITHM CODE

IMPROVING DATA RATE

While developing the software, I noticed that the rate at which the data packets were received and processed was very irregular and erratic. There were glitches and hiccups in the graphing of the plots. As I mentioned previously, I also observed some dips in the packet serial count graph, indicating missing or dropped packets. I did some investigations to narrow down the source of the issue and found that doing less in the Python software allowed it to receive and service the new packets more quickly. I realized, therefore, that there is some optimization work that needs to be done in the python scripts in order to improve its efficiency.

Optimizing

To get started on making efficiency improvements in the software code, I started by just going through it and making any adjustments with potential that I can think of. This included removing some avoidable branches (if/else statements), decreasing computation time when possible, and reducing the number of function calls so as not to overwhelm the software stack.

Because the first version of the GUI software was not written by me, there were still some parts of the code which I didn't have familiarity with because I simply didn't have a reason to review or understand them previously. The first confusion I had was when I noticed a forever waiting loop in the main program loop: `while True: time.sleep(1)`. When I deleted this, nothing changed in the performance of the program. This led me to wonder what it is in the program that updates the graphs with the new data. I discovered that the graphs are updated using a timer which calls the update plots function, which use updated arrays with the new data prepared by the `dataReadyCallback()` function called when a new WiFi packet is received as each data packet is extracted from it. This timer was originally set to go off every 30 milliseconds (at a frequency of 33 Hz), which is slow compared to our data rates and graph data array updating speeds (the absolute minimum we use, and only for debugging purposes, is 250 Hz, but we generally use 1000 Hz). So I changed the timer to go off instead at every millisecond, increasing the graph updating speed to 1000 Hz. To optimize this even further, I added a variable to indicate whether new data has been processed and the data arrays have been updated since the last graph update: `newData`. The way it works is simple: Every time the `dataReadyCallback()` function readies a new batch of data in the arrays to update the graphs, it sets the `newData` variable to True, and every time the timer goes off, and the graph update function runs, it first checks if `newData` is `True`. If it is, it sets it to `False` before updating the graphs. If it is False, then this function doesn't do anything. This saves time that could have been wasted re-graphing the plots when there is no new data, and therefore no reason to plot the graph again.

Another optimization done in the software is running the FFT analysis algorithm in its own thread, so as not to hog machine time and slow down the graphing and processing of the data in other plots.

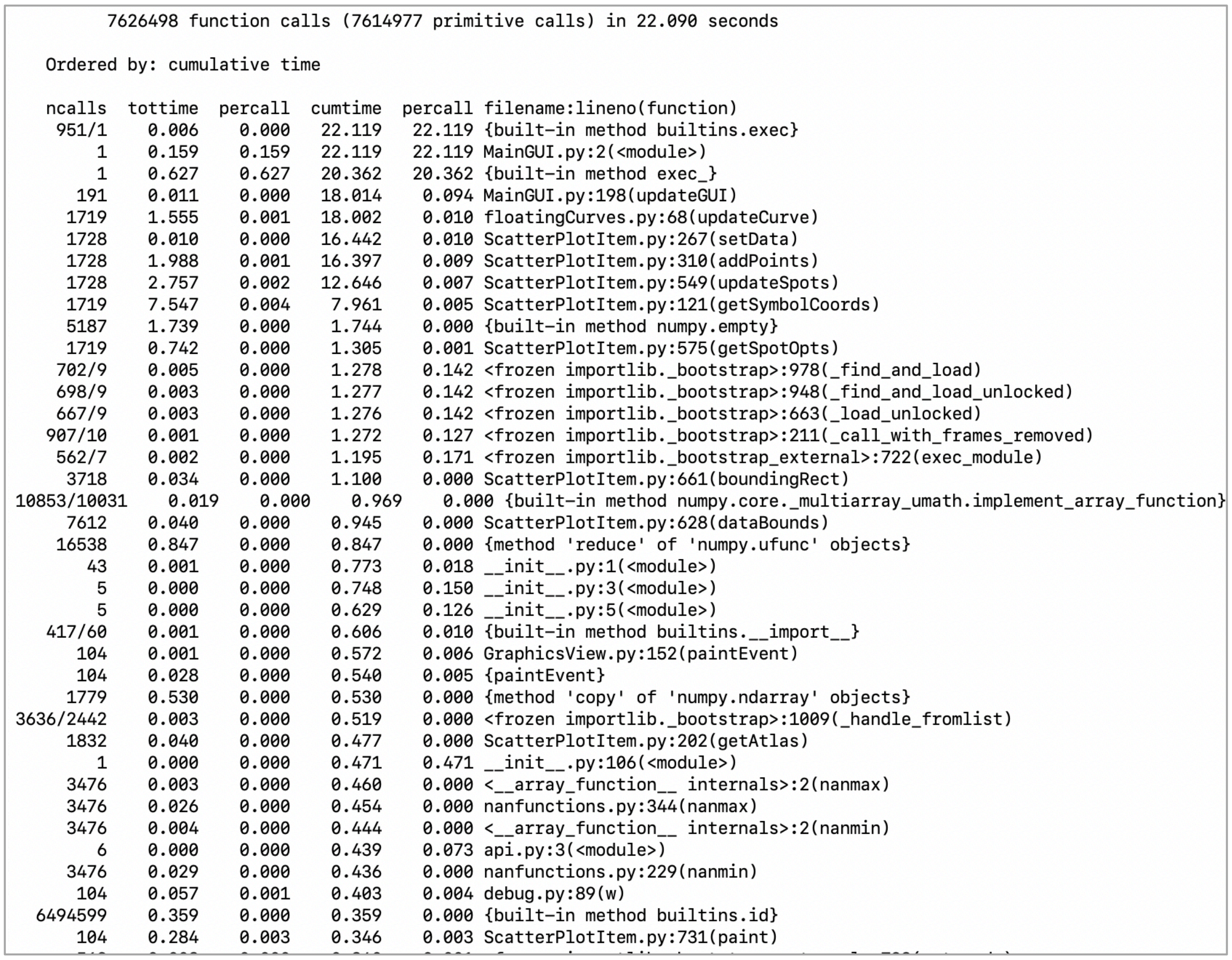
Profiling

Profiling is a tool for analyzing the performance of a program. It is the dynamic analysis which measures the execution time in every nested part and each function of the program and reports it back after the termination of that program.

Some modifications had to be made to the software code in order to enable it to be profiled. For example, the software program is meant to run indefinitely, receiving WiFi packets over the network from the device as long as they are being sent. This strategy does not work if I want to profile the program– the program must terminate in order to get any profiling stats. To achieve this, I came up with a way for the program to reach a termination condition: collecting a certain number of data packets. I implemented this by introducing a variable for the number of desired data packets to be received and graphed before the program halts execution and using a running count of the number of packets the system has received so far to compare it to the set threshold. If the program reaches that point, it terminates all processes and closes all windows. If the program is to be run normally without a limit to the number of data packets received, that variable threshold must simply be set to `math.inf`.

Once the program is set to automatically terminate, it is ready to be profiled. A Python program can be profiled by running the following line in the terminal window: `python -m cProfile -s cumulative path/to/python/file/to_profile.py` []. The output of this looks like in the figure below. `ncalls` refers to the number of calls to that function/line. `tottime` is the total amount of time spent in that function/line. `percall` is ratio tottime/ncalls resulting in the time spent at each call of the function/line. `cumtime` is the cumulative time spent in this function/line and all the function calls that happen within it, nested. This output is sorted by cumulative time (using the `-s cumulative` setting). This can be used to observe which functions take longer than expected and find where the bottleneck of the program could lie.

In the case of the Python GUI program, as shown in the figure below, the functions that take the most time to execute are almost entirely graphing related functions such as `updateCurve`, `setData`, `addPoints`, `updateSpots`, etc. This is also due to the sheer number of times they are called, which means that even miniscule improvements in efficiency could make a difference if the function in which they exist is called enough number of times. Because of this, the efforts were focused on making optimizations in the graphing code which in the GUI code (even though the functions which appear in the profiling output here are mostly library graphing implementation functions). Another investigation I conducted was using different graph types offered by the library, wondering if one of them was more efficiently implemented than the others. I compared using `scatterPlotItem()`, `plotDataItem()`, and `plotCurveItem()`, and even though at first it seemed like one of them was more slightly performant, I realized after running a few trials that it is just due to the randomness in execution and within a margin of inaccuracy.



```
         7626498 function calls (7614977 primitive calls) in 22.090 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    951/1    0.006    0.000   22.119   22.119 {built-in method builtins.exec}
        1    0.159    0.159   22.119   22.119 MainGUI.py:2(<module>)
        1    0.627    0.627   20.362   20.362 {built-in method exec_}
      191    0.011    0.000   18.014    0.094 MainGUI.py:198(updateGUI)
     1719    1.555    0.001   18.002    0.010 floatingCurves.py:68(updateCurve)
     1728    0.010    0.000   16.442    0.010 ScatterPlotItem.py:267(setData)
     1728    1.988    0.001   16.397    0.009 ScatterPlotItem.py:310(addPoints)
     1728    2.757    0.002   12.646    0.007 ScatterPlotItem.py:549(updateSpots)
     1719    7.547    0.004    7.961    0.005 ScatterPlotItem.py:121(getSymbolCoords)
     5187    1.739    0.000    1.744    0.000 {built-in method numpy.empty}
     1719    0.742    0.000    1.305    0.001 ScatterPlotItem.py:575(getSpotOpts)
    702/9    0.005    0.000    1.278    0.142 <frozen importlib._bootstrap>:978(_find_and_load)
    698/9    0.003    0.000    1.277    0.142 <frozen importlib._bootstrap>:948(_find_and_load_unlocked)
    667/9    0.003    0.000    1.276    0.142 <frozen importlib._bootstrap>:663(_load_unlocked)
   907/10    0.001    0.000    1.272    0.127 <frozen importlib._bootstrap>:211(_call_with_frames_removed)
    562/7    0.002    0.000    1.195    0.171 <frozen importlib._bootstrap_external>:722(exec_module)
     3718    0.034    0.000    1.100    0.000 ScatterPlotItem.py:661(boundingRect)
10853/10031    0.019    0.000    0.969    0.000 {built-in method numpy.core._multiarray_umath.implement_array_function}
     7612    0.040    0.000    0.945    0.000 ScatterPlotItem.py:628(dataBounds)
    16538    0.847    0.000    0.847    0.000 {method 'reduce' of 'numpy.ufunc' objects}
       43    0.001    0.000    0.773    0.018 __init__.py:1(<module>)
        5    0.000    0.000    0.748    0.150 __init__.py:3(<module>)
        5    0.000    0.000    0.629    0.126 __init__.py:5(<module>)
   417/60    0.001    0.000    0.606    0.010 {built-in method builtins.__import__}
      104    0.001    0.000    0.572    0.006 GraphicsView.py:152(paintEvent)
      104    0.028    0.000    0.540    0.005 {paintEvent}
     1779    0.530    0.000    0.530    0.000 {method 'copy' of 'numpy.ndarray' objects}
3636/2442    0.003    0.000    0.519    0.000 <frozen importlib._bootstrap>:1009(_handle_fromlist)
     1832    0.040    0.000    0.477    0.000 ScatterPlotItem.py:202(getAtlas)
        1    0.000    0.000    0.471    0.471 __init__.py:106(<module>)
     3476    0.003    0.000    0.460    0.000 <__array_function__ internals>:2(nanmax)
     3476    0.026    0.000    0.454    0.000 nanfunctions.py:344(nanmax)
     3476    0.004    0.000    0.444    0.000 <__array_function__ internals>:2(nanmin)
        6    0.000    0.000    0.439    0.073 api.py:3(<module>)
     3476    0.029    0.000    0.436    0.000 nanfunctions.py:229(nanmin)
      104    0.057    0.001    0.403    0.004 debug.py:89(w)
  6494599    0.359    0.000    0.359    0.000 {built-in method builtins.id}
      104    0.284    0.003    0.346    0.003 ScatterPlotItem.py:731(paint)
```

FIGURE 16: SAMPLE OUTPUT OF PROFILING THE PYTHON GUI

THE DATA: SUMMARY

Signal	Brief Description	Hardware	Number	Data	Data size	Process of conversion	units	idx packet	Byte	Software filter	$f_{effective}$
EEG	Brain wave	ADS1299	4 channels	`float`	4 Bytes each = 16 Bytes	$\text{Sign\_Extend}\left(\frac{2 \times V_{REF}}{\text{Channel\_Gain} \times 2^{24}}\right)$	volts	7	24	Bandstop at 60 Hz, and Bandpass 5-50 Hz	Data rate is programmable by setting the ADS1299's data rate in the firmware, $f_{ADS1299}$
								8	28		
								9	32		
								10	36		
EMG	Muscle wave	ADS1299	3 channels	`float`	4 Bytes each = 12 Bytes		volts	3	8	Bandstop at 60 Hz, and Bandpass 10-500 Hz	
								4	12		
								6	20		
EOG	Eye movement	ADS1299	1 channel	`float`	4 Bytes		volts	5	16		
EDA	Skin resistance	Arduino ADC	1	`float`	4 Bytes	ADC counts $\times$ 3.3$\times$ R_{ref} /($2^{12}\times$ $V_{virtualgnd}$)	Ohms	17	52	Oversampling and averaging	$\frac{1}{100} f_{ADS1299}$
IMU A	acceleration	MPU6050	3 axes	`int`	2 Bytes each = 6 Bytes	Not performed: value $\times$ 9.81 / 16384	m/s^2	11	40	No filtering at the moment. Data smoothing algorithms could be applied.	$\frac{1}{10} f_{ADS1299}$
								12	42		
								13	44		
IMU G	gyroscope	MPU6050	3 axes	`int`	2 Bytes each = 6 Bytes	Not performed: value / 131	$^{\circ}/s$	14	46		
								15	48		
								16	50		
PPG	Heart signal	MAX30105	1	`int`	4 Bytes	No conversion	N/A	19	60	Heartrate measuring	
Temp	Body temperature	MAX30105	1	`float`	4 Bytes	Arduino Library converts	$^{\circ}$ C	18	56	Not filtered; can be smoothed	

FIGURE 17: TABLE SUMMARIZING HOW SIGNALS IN THE SYSTEM ARE COLLECTED AND PROCESSED

DEVELOPMENT & DEBUGGING

In order to debug the system smoothly and efficiently as I developed the firmware (and, eventually, the software), I started by focusing on the device which needed the most firmware development work: the ADS1299. Currently, there is no public Arduino library that exists for this device. The ADS1299 is a complex device which has many different options for different settings and configurations. For this reason, I started off by creating, developing, and building and expanding on a script which acted as a library to handle setting up, communicating with, and changing the configurations of the ADS1299.

In the beginning of testing, the development and investigations were not specific to the Fascia project. As I mentioned in the Background section, this project builds off of and is developed alongside the AR/VR headset project, and so I wrote the library program to be able to handle both (and multiple versions of) board types for both projects.

FIRMWARE

ADS1299_SS_CC_WiFi

This was the first set of iterations to develop firmware to interface with, set up, and change the configurations of the ADS1299 [24]. At this point, setting up the device was a big black box that had to be revealed and understood. In this version of the firmware was the first successful ADS1299 internal test signal generation signal set up. This firmware tested both continuous conversion mode and single shot mode, enabling the discovery of hardware wiring bugs in the first version of the PCB design of the project (an incorrect connection from the data ready pin to an Arduino GPIO pin which cannot be mapped as in external interrupt trigger pin. This was also the version in which the integration of WiFi connection and basic packet sending first got implemented and tested.

In this version, the serial debugging interface started being developed with different functionalities as they became necessary or useful. It allowed a user to change the gain of any channel, turn each channel on and off, and eventually connect and disconnect each channel from SRB2.

This version was configurable to run the ADS1299 channels as either regular data collection probes, or generating internal test signal. It also enabled setting up the code for multiple version of both the Fascia project boards, and the previous AR/VR project which it diverged from, in order to test and compare the two. It enabled selecting the ADS1299 data collection mode: single shot versus continuous conversion. And lastly, it enabled selectively turning on WiFi and actually sending data over the network.

Fascia_collect_sensor_data

This version started with the previous one (ADS1299_SS_CC_WiFi) as a base, but then grew to include more ADS1299 functionalities as they became needed and expanded to integrate data collection for the other devices and sensors in the system [25]. The ADS1299 serial debugging interface grew in this version to include the functionality to connect each channel's positive or negative leads to the bias probe or disconnected any of them from it. Additionally, the interface now supports a command to print the current status of all the ADS1299 registers.

This software features two new modes: debug mode, and verbose mode. Debug mode, if activated allows you to use the aforementioned serial debugging interface for the ADS1299. It prints values and responses to the serial monitor, and takes in commands from it. If deactivated, no commands can be sent through the serial monitor. Verbose mode allows the user to enable or disable serial print statements which continuously stream the values of all the data collected from all the sensors in the device as it is collected. Since this version integrates all the other devices in the system, not just the ADS1299, this is actually a lot of data and can be an overwhelming amount to look at, and it can cause major slow-downs in the data collection and packet sending routine. That is why we have verbose mode: to enable it when you need to see the data on the firmware side, but disable it otherwise as it can compromise the efficiency of the system.

SOFTWARE

GUI

While developing the GUI [26 & 27], it became very helpful to make some adjustments to help with debugging and observing program behavior. One of the biggest issues was integrating an FFT graph to perform frequency analysis on a given (selectable) signal. This enables us to see the channel data in the frequency domain and determine whether a certain signal of specific bandwidth exists, grows stronger, or diminishes and disappears. However, because the FFT algorithm can be quite computationally intensive, this graphing feature was also selectively turned on or off. Another change to increase performance was to allow optionally skipping graphing the first n data points in the data packet, so as to be able to focus on the ones that come later (from other sensors) as they were integrated into the system, bit by bit.

Finally, in order to be able to profile the code, a change had to be made to end the program eventually. A feature was implemented to allow the option to terminate the GUI program after collecting a certain specified number of samples.

Signals

In order to ensure the device is detecting the signals of interest, even before making the specific signal itself visible, there were means of ensuring the system was behaving appropriately. By following a standard setup procedure of a known developed system, and observing the results and looking for the target changes in our device, and comparing those to the expected ones, we can gain some confidence that the system is functional. The procedure referenced here is setup for EEG as guided by the Open BCI ADS1299 development kit documentation [28].

Once we know the signal is there, we can focus on being able to view it. For this, we need to ensure that the digital filters we have in place for each signal are appropriate and correct for the signals and the bandwidth we are targeting. To this end, after doing some research to select the settings and filters, the Open BCI documentation on the firmware and GUI was a very helpful reference to cross check if our selected firmware settings and digital filter thresholds and constants were reasonable [29 & 30].

CLOSING

To conclude, this project aims to support and simplify sleep research data by improving the form factor of the sensors and electrodes placed on a subject during sleep studies, which enables the patient to sleep more comfortably and thus results in more authentic signals. The state of the art in the field does not satisfy those requirements, so the design of the hardware, software, and physical appearance of the device for this project takes this into account in order to produce a comfortable, flexible, and, importantly, comprehensive data collection device, which is completely open-source.

The hardware and firmware facets of the project are almost done and mostly finalized, but there is still some software work to be done. The remaining work is detailed in the next section, “Moving Forward.”

I know that this project has great potential, and the amount of progress made on it in the past year is just the beginning! Once we have algorithms in place to analyze the collected data, there is no telling how many applications this could have and how many people it could help. I hope this effort can aid doctors and researchers in making advancements in the understanding of sleep, and diagnosis of sleep related issues. Ultimately, I hope this project can help people out there suffering from any kind of sleep problem. I am honored to have had the opportunity to be part of this project.

MOVING FORWARD

Next Steps

- ☐ Assemble a complete prototype of the device.
- ☐ Compare sensor data with golden standard.
- ☐ Record or store device data as it is collected in the software.
- ☐ Implement reliable lead-off detection (current firmware setup is at the bottom of the `ADS_init()` function in `Fascia_collect_sensor_data`).
- ☐ Integrate the security chip and encrypt the data being transferred.

Future Work

- ☐ Use Machine Learning to extract features of interest from signals.

Nice to Finalize

- ☐ Further profile and optimize the code.
- ☐ Investigate and finalize the digital filters used.

Things to Look Into

- ☐ Transferring data using Bluetooth instead of WiFi.
- ☐ Using Lab Streaming Layer (LSL) to collect and stream data: https://github.com/sccn/labstreaminglayer
- ☐ Utilizing Brainflow to parse and analyze biosensor data: https://brainflow.readthedocs.io/en/stable/UserAPI.html#python-api-reference

GLOSSARY

Op-amp	Operational Amplifier— an electronic chip which can be used to amplify, difference, and/or buffer analog signals in a circuit.
PCB	Printed Circuit Board— a small board which uses printed copper traces to connect very small surface-mount electronic devices to create a task-specific circuit.
DRL	Driven Right Leg— a circuit for removing the “common-mode noise” from a physiological signal by sensing the noise from another part of the human body.
ADC	Analog to Digital Converter— a device which performs a conversion from an analog value (e.g. voltage in a circuit), to a digital value in a binary format.
IC	Integrated Circuit— an electronic device packed into a small packet to perform a specific task.
MCU	Microcontroller— a programmable device which enabled the collection and manipulation of other hardware devices and data.
DC	Direct Current— constant amount of uninterrupted current flowing.
AC	Alternating Current—current flows in a sine wave varying in amplitude between ± maximum amplitude at a set frequency.
SPS	Samples Per Second—a measure of the speed with which a data signal is sampled.
SPI	Serial Peripheral Interface— communication protocol between two IC devices which uses 1 clock line, 1 data in line, 1 data out line, and 1 “chip select” line to facilitate the exchange of data.
I2C	Inter-IC (integrated circuit)— communication protocol using 1 data line and 1 clock line to facilitate the data exchange between two IC devices.
MSB	Most Significant Bit— highest order bit in a binary value or bit-array or bitmap.
LSB	Least Significant Bit— lowest order bit in a binary value or bit-array or bitmap.
FIFO	First In, First Out— a data structure or type of buffer in which items are sequential in insertion order, which is the same as the order of retrieval from it.
BPM	Beats Per Minute— the unit for measuring heartrate.
GUI	Graphical User Interface— a program which allows users to view and change software behavior using a graphical interface.
FFT	Fast Fourier Transform— an algorithm for converting waveforms to frequency data.
GPIO	General Purpose Input/Output— specific pins on electronic chips which can be used to output data and/or read-in data.

BIBLIOGRAPHY

0. "6. The MEng Thesis Proposal." *6. The MEng Thesis Proposal | MIT EECS*, https://www.eecs.mit.edu/node/5422.

1. "How Does a Sleep Study Work?" *National Sleep Foundation*, https://www.sleepfoundation.org/excessive-sleepiness/diagnosis/how-does-sleep-study-work.

2. Adamczyk, Kamil. "Neuroon Open - Advanced Open-Source Sleep Tracking EEG Mask and Band." *Product Hunt*, 11 Sept. 2019, https://www.producthunt.com/posts/neuroon-open.

3. "Neurofeedback EEG Device - How It Works." *Muse*, https://choosemuse.com/how-it-works/.

4. "Polysomnography (Sleep Study)." *Mayo Clinic*, Mayo Foundation for Medical Education and Research, 17 Nov. 2018, https://www.mayoclinic.org/tests-procedures/polysomnography/about/pac-20394877.

5. "EEG (Electroencephalogram) (for Parents) - Nemours KidsHealth." Edited by KidsHealth Medical Experts, *KidsHealth*, The Nemours Foundation, https://kidshealth.org/en/parents/eeg.html.

6. "Electromyography (EMG)." *Healthline*, Healthline Media, https://www.healthline.com/health/electromyography/.

7. "Electrooculography." *Wikipedia*, Wikimedia Foundation, 28 Nov. 2019, https://en.wikipedia.org/wiki/Electrooculography.

8. Zangróniz, et al. "Electrodermal Activity Sensor for Classification of Calm/Distress Condition." *MDPI*, Multidisciplinary Digital Publishing Institute, 12 Oct. 2017, https://www.mdpi.com/1424-8220/17/10/2324/htm#B25-sensors-17-02324.

9. "ADS1299-x Low-Noise, 4-, 6-, 8-Channel, 24-Bit, Analog-to-Digital Converter for EEG and Biopotential Measurements." TI, Texas Instruments, July 2012, revised Jan 2017. http://www.ti.com/lit/ds/symlink/ads1299.pdf?&ts=1588889413443

10. Reis, Pedro M. R. "What Is the Range of Human EMG Signal Frequencies: Min. and Max?," August 21, 2013. https://www.researchgate.net/post/What_is_the_range_of_Human_EMG_signal_frequencies_Min_and_Max2.

11. "Electrooculography." Electrooculography - an overview | ScienceDirect Topics, n.d. https://www.sciencedirect.com/topics/engineering/electrooculography.

12. "Understanding Lead-Off Detection in ECG." TI, Texas Instruments, May 2012, revised Jan 2015. http://www.ti.com/lit/an/sbaa196a/sbaa196a.pdf

13. Newson, Jennifer J., Thiagarajan, and Tara C. "EEG Frequency Bands in Psychiatric Disorders: A Review of Resting State Studies." Frontiers. Frontiers, December 11, 2018. https://www.frontiersin.org/articles/10.3389/fnhum.2018.00521/full.

14. Farnsworth, Bryan. "What Is EDA? And How Does It Work?" imotions, n.d, 4 June 2019. https://imotions.com/blog/eda/.

15. "AN118 Improving ADC Resolution by Oversampling and Averaging." Cypress. Silicon Laboratories, July 2013. https://www.cypress.com/file/236481/download.

16. Boucsein, Wolfram. *Electrodermal Activity*. New York: Springer, 2012, pg 49.

17. "Publication Recommendations for Electrodermal Measurements." Society for Psychophysiological Research, 2012. https://onlinelibrary.wiley.com/doi/epdf/10.1111/j.1469-8986.2012.01384.x.

18. "GitHub." *GitHub*. SparkFun, n.d. https://github.com/sparkfun/SparkFun_MAX3010x_Sensor_Library/blob/master/.

19. Gholipour, Bahar. "What Is a Normal Heart Rate?" LiveScience. Purch, January 12, 2018. https://www.livescience.com/42081-normal-heart-rate.html.

20. "GitHub." *GitHub*. Electronic Cats, n.d. https://github.com/ElectronicCats/mpu6050.

21. "MPU-6000 and MPU-6050 Register Map and Descriptions" InvenSense Inc, 19 August 2013. https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf

22. "GitHub." *GitHub*. Arduino, n.d. 22. https://github.com/arduino-libraries/WiFiNINA.

23. "Python Heart Rate Analysis Toolkit." *Python Heart Rate Analysis Toolkit*, n.d. https://python-heart-rate-analysis-toolkit.readthedocs.io/en/latest/.

24. Alkhanaizi, Walaa, and Bernal, Guillermo. "GitHub." *GitHub*. MIT, n.d. https://github.mit.edu/gbernal/Fascia_nucleus/tree/master/Firmware/development/ADS1299_SS_CC_WiFi

25. Alkhanaizi, Walaa, and Bernal, Guillermo. "GitHub." *GitHub*. MIT, n.d. https://github.mit.edu/gbernal/Fascia_nucleus/tree/master/Firmware/development/Fascia_collect_sensor_data.

26. Alkhanaizi, Walaa, Bernal, Guillermo, and Qiao, Junqing. "GitHub." *GitHub*. MIT, n.d. https://github.mit.edu/gbernal/Fascia_nucleus/tree/master/Fascia_dataViz/ADS1299_data plotter

27. Alkhanaizi, Walaa, Bernal, Guillermo, and Qiao, Junqing. "GitHub." *GitHub*. MIT, n.d. https://github.mit.edu/gbernal/Fascia_nucleus/tree/master/Fascia_dataViz/Fascia_sensor_data_plotter

28. "Setting up for EEG · OpenBCI Documentation." OpenBCI Documentation, n.d. https://docs.openbci.com/docs/01GettingStarted/02-Biosensing-Setups/EEGSetup#3-launch-the-gui-and-adjust-your-channel-settings.

29. "GitHub." *GitHub*. OpenBCI, n.d. https://github.com/OpenBCI/OpenBCI_GUI.

30. "GitHub." *GitHub*. OpenBCI, n.d. https://github.com/OpenBCI/OpenBCI_Cyton_Library

31. Made with the initial help of notion.so

32. "Hypnodyne ZMax Is an Advanced and Simple to Use EEG Home Sleep Monitor Used by Researchers but Available to Everyone." Hypnodyne ZMax, hypnodynecorp.com/index.php.

APPENDICES

APPENDIX A: HOW TO USE (A GUIDE)

About

This guide explains the usage and functionality specification of the Fascia_collect_sensor_data Arduino firmware in the Fascia_nucleus repo (Fascia_nucleus/Firmware/development/Fascia_collect_sensor_data) commit hash: commit `12a1f3e36ee31194a78e70afae84bb2bf27b8d47` "some clean up" [31].

Set Up

Software

In order to begin running this code, you must have all the required libraries installed. This includes `WiFiNINA`, and `MPU6050` by Electronic Cats, both of which you can get from the Arduino library manager, and `SparkFun_MAX3010x_Pulse_and_Proximity_Sensor_Library` which you need the modified version of in the repo. This is located in Fascia_nucleus/Firmware/development/Libraries/SparkFun_MAX3010x_Pulse_and_Proximity_Sensor_Library. I have made changes to speed up the data acquisition in the library, and reduce wasted wait time. Namely, I made changes to two different functions:

- `readTemperature()`

I split this function into two functions: `requestTemperature()`, and `readTemperature()`. Previous to this, the library would request a temperature reading from the MAX3010x and wait for a while for the data to be ready. With this change, I request the temperature using the request function, then perform other things in the consumer code, and the read the temperature by calling the read function, without waiting idly for the sensor to be ready.

- `getIR()`

This function did a similar thing where once you called it, it would wait 250 ms for the new data to be ready. Instead of doing this, I made the function take as input the number of milli seconds you'd like it to wait for the data to be ready, and if the data is still not ready by that time, the function returns `0`, which indicates an invalid/unavailable reading of the PPG sensor.

In order to run the code, you must copy this modified library and paste it into your Arduino libraries folder (usually in your documents folder).

Hardware

Once you're able to start running the code, you might notice that the code hangs in certain areas: this will happen if any of the sensors the code expects you to have proper connection to are unreachable. Make sure your ADS1299, MAX3010, and MPU5060 are properly connected all the way to the Fascia Main Board, and that the Arduino can communicate with them.

💡 If you are missing one or more of the sensors, you can get the code to run without expecting said sensor(s) to be connected by commenting out the lines where the code is setting the sensor(s) up, and where it is retrieving data from the sensor(s). You can find these lines in the two functions `setup()` and `loop()`, where sensor setup and data retrieval occur, respectively.

Configurations

In order to properly get the code to run and do what you'd like to do with it, you'll want to make sure you have correctly configured the top of the .ino file with your current setup and desired output. These settings can be found on lines 19-28 of Fascia_collect_sensor_data.ino:

```
// settings
#define CONNECT_WIFI 1
#define BOARD_V FASCIA_V0_0
#define DATA_MODE RDATA_SS_MODE
#define RUN_MODE NORMAL_ELECTRODES
// v for verbose: lots of prints
#define v 0
// debug: serial reads and writes
#define debug 1
```

- `CONNECT_WIFI` would enable(`1`)/disable(`0`) WiFi on the chip. Fascia will not send any data packets over WiFi when this is disabled (`0`).

- `BOARD_V` allows you to specify the board you are working with. This code is specifically written for Fascia, so I cannot guarantee that any data, other than the ADS1299 data, would be correct or using the proper pin to which that sensor is connected, if you use any other `BOARD_V` other than `FASCIA_V0_0` or `FASCIA_V0_1` with this version of this code.

- `DATA_MODE` this version of the code only supports `RDATA_SS_MODE` currently; please do not change this line. This is referring to the ADS1299 Single Shot mode, versus the Continuous Conversion mode

- `RUN_MODE` enables you to generate internal-test signals in the ADS1299 instead of reading external data and signals from the electrodes on the 8 channels of the ADS1299 if you set it to `GEN_TEST_SIGNAL`. Keep it as `NORMAL_ELECTRODES` for any purpose other than generating the mentioned internal test signals.

- `v` enables/disables verbose print statements in the Serial port. If `1` (enabled) you'll see all the collected sensor data values printed in the Arduino Serial Monitor. `0` disables this.

- `debug` enables/disables the ability to change some specific ADS1299 channel settings via serial messages through the serial monitor. The detailed description of the things you can do will be printed once setup is complete, if you set `debug` `1` and run the code. These things include enabling/disabling Bias, SRB2, and changing the gain of each channel, as well as turning each channel on/off, and even checking the current status of all the ADS1299 registers.

Using the Serial Debug Interface

Here is a summary of the commands you can run in the debug serial interface, and what they do:

- type the channel number to print that ADS1299 channel's data [1-8] (and plot, if you switch to Serial Plotter)
- or type '0' to stop printing the data.
- type BN#0 to deactivate biasN for channel # and BN#1 to activate it
- type BP#0 to deactivate biasP for channel # and BP#1 to activate it
- type S#0 to deactivate SRB2 for channel # and B#1 to activate it
- type G#N to set the gain for channel # to N=0:1, N=1:2, N=2:4, N=3:6, N=4:8, N=5:12, N=6:24
- type T#0 to toggle channel # off, and T#1 to toggle channel # on
- type 'R' or 'r' to print the current register settings of the ADS1299
- type 'P' or 'p' to print these instructions again

Configuring Data Rate

You can adjust the data rate of the ADS1299 (and since all the other signals' data rates are programmed to be a tenth of it) in the firmware, in line 183 of Fascia_collect_sensor_data.ino by changing the last argument of this line `ADS_WREG(ADS1299_REGADDR_CONFIG1,________);`

The available sampling rates are listed starting on line 152 in the header file ADS1299.h:

```
#define ADS1299_REG_CONFIG1_16kSPS 0 // Data is output at FMOD/64, or 16 kHz at 2.048 MHz
#define ADS1299_REG_CONFIG1_8kSPS  1 // Data is output at FMOD/128, or 8 kHz at 2.048 MHz
#define ADS1299_REG_CONFIG1_4kSPS  2 // Data is output at FMOD/256, or 4 kHz at 2.048 MHz
#define ADS1299_REG_CONFIG1_2kSPS  3 // Data is output at FMOD/512, or 2 kHz at 2.048 MHz
#define ADS1299_REG_CONFIG1_1kSPS  4 // Data is output at FMOD/1024, or 1 kHz at 2.048 MHz
#define ADS1299_REG_CONFIG1_500SPS 5 // Data is output at FMOD/2048, or 500 Hz at 2.048 MHz
#define ADS1299_REG_CONFIG1_250SPS 6 // Data is output at FMOD/4096, or 250 Hz at 2.048 MHz
```

Running/visualizing the code

We have Python code in place to help visualize the sensor data received over WiFi. In order for this to work, you must set `CONNECT_WIFI` to `1` in the .ino file in order for the two scripts to be able to connect and share the data. In addition to this, you have to make sure that both the Arduino and the python code are connecting to the correct WiFi network.

Firmware (.ino)

In the file WiFi_Settings.h

```
#define SECRET_SSID "raspi_wifi"
#define SECRET_PASS "fluidfluid"
#define HOST_ID     "192.168.0.101"
```

Ensure that `SECRET_SSID` has the name of your private WiFi network, and `SECRET_PASS` has the password to that network. `HOST_ID` should be the IP address of your computer if you type `ifconfig` in your terminal, and find `en0`, include the IP address listed under that.

In the same file, take a note of these lines:

```
#define SEND_SIZE 22
#define NUM_ELEMENTS 17
#define ELEM_SIZE 4
```

You'll need these values to ensure that the connection to the python script is correct.

Data Visualization Code (.py)

In the file MainGUI.py, you must make sure the IP address that it is connecting to is correct, and the same as the one you inputted into the firmware code. This is in line 120:

```
self.ip = '192.168.0.101'
```

Next, take a look at BCI_Data_Receiver.py, and the lines 48-50:

```
num_elements = 17
num_bytes = 4*num_elements
num_packets = 22
```

Make sure that `num_elements` in the python script matches the `NUM_ELEMENTS` in the Arduino header file, and that `num_packets` in the python script matches `SEND_SIZE` in the Arduino header file. Lastly, ensure that `ELEM_SIZE` in the Arduino header file matches the multiplier in calculating `num_bytes` in the python script (currently correctly `4`).

Now, you are ready to set GUI options to your preference, which are all the lines marked with a `TODO` comment in the MainGUI.py file, near the top (lines 37-59):

```
# for plotting
self.start_idx = 2   #TODO: make this 0 if you want to graph all the packet data
...
# for FFT
self.graph_fft = 1   #TODO: change this to 1 if you dont want FFT graph
self.fft_channel = 4 #TODO: make sure this is the channel you want the FFT for (0 indexed)
...
# for filters
data_rate = 1000     #TODO: make sure this matches the data rate of the ADS1299 in the firmware
```

- `start_idx` is the index of the data packet at which to begin graphing. You can start at 0 in order to graph *all* the data, or you can skip the first n data points by setting this value to n
- `graph_fft` is a boolean, if set to `1`, an FFT plot will be generated, and if it is `0`, no FFT graph will be plotted in the GUI window
- `fft_channel` if you would like to have an FFT, this allows you to select which ADS1299 channel to run the FFT algorithm on and produce the frequency data for
- `data_rate` must match the ADS1299 data rate that is set in the firmware code—specifically this would be the settings for the ADS1299 register `config1`

Once you run the Python GUI program, you should be able to view the signals you selected, and move around and zoom in and out of each graph, as well as see the Current data rate and current measured heart rate.

Packet break-down

Order	Type	Size (byte)	data	Description
0	int	4	count	Serial number of the packet
1	int	4	valid array	array of bits indicating valid/invalid data in the packet
2-9	float	4 * 8	ADS 8 channel data	sign extended ADS channel data, in order
10-15	int	2 * 6	IMU 6 data points	dummy data for now, since sensor not properly connected
16	float	4	EDA	raw Arduino ADC counts
17	float	4	temperature	
18	int	4	PPG	raw IR sensor data
19	int	4	time stamp	

💡 Please note: the valid array holds a 1 in the bit location corresponding to the data point in the packet which is invalid. For example, if the PPG data is invalid, then the valid array's 18th bit will be a 1. The index of the bit maps to the location of the data in the packet.

IMU Data Conversion

If you decide you'd like the IMU data to be converted to metric units instead of being unitless ADC counts, you must uncomment the lines which perform this conversion in the `get_IMU_data()` function (lines 752-755 for gyroscope data, and lines 765-767 for acceleration data), and update the WiFi packet with the floating point data (lines 770-775).

Since this modification changes the size of each IMU data point, it also affects the size of the data packets, so it is crucial to modify `num_elements` in the python script, and `NUM_ELEMENTS` in the Arduino code to *20*. This enables the firmware to send a data packet of the correct size, and the software to receive a packet of the correct size.

The last thing you need to change in order to get the IMU conversion working end-to-end is ensure that the python code can interpret the sent packet data correctly. This is done by specifying the type, in order, of the elements of the packet. In BCI_data_receiver.py, you'll find the line (line # 37):

```
unpacked_data = struct.unpack('i'+'i'+'f'*8+'h'*6+'f'+'f'+'ii', data[inum_bytes:(i+1)*num_bytes])
```

The data types and the order they appear in maps to the order in the packet, as shown in the table above. In order to complete this set up, you must change the `'h'*6` which means six elements of type `short` (which is 2 Bytes), to `'f'*6`, meaning six elements of type `float`, which is 4 Bytes and holds our converted data values.

APPENDIX B: FIRMWARE

https://github.mit.edu/gbernal/Fascia_nucleus/tree/master/Firmware/development/Fascia_collect_sensor_data

ads1299.h

```
#define int32_t unsigned long
#define SIGN_EXT_24(VAL)    ((int32_t)((uint32_t)(VAL) ^ (1UL<<(23))) - (1L<<(23)))

/*****************************************************************************************************************************
 *              Sampling Config                                                                                              *
 *****************************************************************************************************************************/

/**
 *  \brief Default data rate from the ADS1299.
 *
 *    To monitor electrode impedance continuously, an AC current is pulsed through each electrode and the corresponding
 *  voltage perturbation observed in the measured signal. This signal will not be easily separable from the EEG if it
 *  is within the typical 0-100 Hz EEG bandwidth; since the fastest possible AC excitation rate the ADS1299 can
 *  generate is (data rate)/4, the lowest recommended data rate that allows continuous impedance monitoring is 1000 Hz.
 *  Using a 500 Hz data rate will generate an AC excitation at 125 Hz, which is dangerously close to, if not in,
 *  the EEG band.
 */
#define DEFAULT_SAMPLE_RATE         250
#define MAX_CHANNELS            8



/*****************************************************************************************************************************
*               Other Useful Definitions                                                                                     *
*****************************************************************************************************************************/



#define SIGN_EXT_24(VAL)   ((int32_t)((uint32_t)(VAL) ^ (1UL<<(23))) - (1L<<(23)))// from Junqing inital code, it works but can't understand how it works

#define ADS_data_MSB_mask 0x00800000L
#define sign_extend_bytes 0xFF000000L
#define SIGNEXTEND(VAL)     (VAL & ADS_data_MSB_mask)? (VAL|sign_extend_bytes) : VAL;

/* Default register values */
#define ADS1299_REGDEFAULT_ID           ADS1299_DEVICE_ID
#define ADS1299_REGDEFAULT_CONFIG1      0x96            ///< Multiple readback mode, OSC output disabled, DR = FMOD/4096
#define ADS1299_REGDEFAULT_CONFIG2      0xD0            ///< Test signal sourced internally, low-amplitude test signal pulsed at FCLK/(2^21)
#define ADS1299_REGDEFAULT_CONFIG3      0x68            ///< Ref buffer off, bias measurement off, internal bias ref, bias buffer off, bias sense disabled
#define ADS1299_REGDEFAULT_LOFF         0x00            ///< 95%/5% LOFF comparator threshold, DC lead-off at 6 nA
#define ADS1299_REGDEFAULT_CHNSET       0xE0            ///< Channel off, gain 24, SRB2 disconnected, normal electrode input
#define ADS1299_REGDEFAULT_BIAS_SENSP   0x00            ///< All BIAS channels disconnected from positive leads
#define ADS1299_REGDEFAULT_BIAS_SENSN   0x00            ///< All BIAS channels disconnected from negative leads
#define ADS1299_REGDEFAULT_LOFF_SENSP   0x00            ///< All LOFF channels disconnected from positive leads
#define ADS1299_REGDEFAULT_LOFF_SENSN   0x00            ///< All LOFF channels disconnected from negative leads
#define ADS1299_REGDEFAULT_LOFF_FLIP    0x00            ///< No flipping in this house; source/pull-up at INP, sink/pull-down at INN
#define ADS1299_REGDEFAULT_LOFF_STATP   0x00            ///< This is a read-only register; reset value is 0x00
#define ADS1299_REGDEFAULT_LOFF_STATN   0x00            ///< This is a read-only register; reset value is 0x00
#define ADS1299_REGDEFAULT_GPIO         0x0F            ///< All GPIO set to inputs
#define ADS1299_REGDEFAULT_MISC1        0x00            ///< SRB1 disconnected
#define ADS1299_REGDEFAULT_MISC2        0x00            ///< Register not used in this silicon; should stay at 0x00
#define ADS1299_REGDEFAULT_CONFIG4      0x00            ///< Continuous conversion, LOFF comparator powered down

String ADS_reg_names[24]= {"ID","CONFIG1","CONFIG2","CONFIG3","LOFF","CH1SET","CH2SET",
                           "CH3SET","CH4SET","CH5SET","CH6SET","CH7SET","CH8SET",
                           "BIAS_SENSP","BIAS_SENSN","LOFF_SENSP","LOFF_SENSN","LOFF_FLIP",
                           "LOFF_STATP","LOFF_STATN","GPIO","MISC1","MISC2","CONFIG4"};

/*****************************************************************************************************************************
*               Typedefs and Struct Declarations/Definitions                                                                 *
*****************************************************************************************************************************/

/**
 *  \brief Error codes for interacting with the ADS1299.
 *
 */
typedef int ads1299_error_t;

/**
 *  \brief ADS1299 register addresses.
 *
 * Consult the ADS1299 datasheet and user's guide for more information.
 */
#define ADS1299_REGADDR_ID          0x00            ///< Chip ID register. Read-only. ID[4:0] should be 11110.
#define ADS1299_REGADDR_CONFIG1     0x01            ///< Configuration register 1. Controls daisy-chain mode; clock output; and data rate.
#define ADS1299_REGADDR_CONFIG2     0x02            ///< Configuration register 2. Controls calibration signal source, amplitude, and frequency.
#define ADS1299_REGADDR_CONFIG3     0x03            ///< Configuration register 3. Controls reference buffer power and the bias functionality.
#define ADS1299_REGADDR_LOFF        0x04            ///< Lead-off control register. Controls lead-off frequency, magnitude, and threshold.
#define ADS1299_REGADDR_CH1SET      0x05            ///< Channel 1 settings register. Controls channel 1 input mux, SRB2 switch, gain, and power-down.
#define ADS1299_REGADDR_CH2SET      0x06            ///< Channel 2 settings register. Controls channel 2 input mux, SRB2 switch, gain, and power-down.
#define ADS1299_REGADDR_CH3SET      0x07            ///< Channel 3 settings register. Controls channel 3 input mux, SRB2 switch, gain, and power-down.
#define ADS1299_REGADDR_CH4SET      0x08            ///< Channel 4 settings register. Controls channel 4 input mux, SRB2 switch, gain, and power-down.
```

```
#define ADS1299_REGADDR_CH5SET       0x09           ///< Channel 5 settings register. Controls channel 5 input mux, SRB2 switch, gain, and power-down.
#define ADS1299_REGADDR_CH6SET       0x0A           ///< Channel 6 settings register. Controls channel 6 input mux, SRB2 switch, gain, and power-down.
#define ADS1299_REGADDR_CH7SET       0x0B           ///< Channel 7 settings register. Controls channel 7 input mux, SRB2 switch, gain, and power-down.
#define ADS1299_REGADDR_CH8SET       0x0C           ///< Channel 8 settings register. Controls channel 8 input mux, SRB2 switch, gain, and power-down.
#define ADS1299_REGADDR_BIAS_SENSP   0x0D           ///< Bias drive positive sense selection. Selects channels for bias drive derivation (positive side).
#define ADS1299_REGADDR_BIAS_SENSN   0x0E           ///< Bias drive negative sense selection. Selects channels for bias drive derivation (negative side).
#define ADS1299_REGADDR_LOFF_SENSP   0x0F           ///< Lead-off positive sense selection. Selects channels that will use lead-off detection (positive
side).
#define ADS1299_REGADDR_LOFF_SENSN   0x10           ///< Lead-off negative sense selection. Selects channels that will use lead-off detection (negative
side).
#define ADS1299_REGADDR_LOFF_FLIP    0x11           ///< 1: Swaps lead-off current source and sink on the corresponding channel. See datasheet.
#define ADS1299_REGADDR_LOFF_STATP   0x12           ///< Lead-off positive side status register. Read-only. 0: lead on, 1: lead off.
#define ADS1299_REGADDR_LOFF_STATN   0x13           ///< Lead-off negative side status register. Read-only. 0: lead on, 1: lead off.
#define ADS1299_REGADDR_GPIO         0x14           ///< GPIO register. Controls state and direction of the ADS1299 GPIO pins.
#define ADS1299_REGADDR_MISC1        0x15           ///< Miscellaneous 1. Connects/disconnects SRB1 to all channels' inverting inputs.
#define ADS1299_REGADDR_MISC2        0x16           ///< Miscellaneous 2. No functionality in current revision of ADS1299.
#define ADS1299_REGADDR_CONFIG4      0x17           ///< Configuration register 4. Enables/disables single-shot mode and controls lead-off comparator
power.

byte CHANNELS[8] = {ADS1299_REGADDR_CH1SET,
                    ADS1299_REGADDR_CH2SET,
                    ADS1299_REGADDR_CH3SET,
                    ADS1299_REGADDR_CH4SET,
                    ADS1299_REGADDR_CH5SET,
                    ADS1299_REGADDR_CH6SET,
                    ADS1299_REGADDR_CH7SET,
                    ADS1299_REGADDR_CH8SET};
/**
 *  \brief ADS1299 SPI communication opcodes.
 *
 * Consult the ADS1299 datasheet and user's guide for more information.
 * For RREG and WREG opcodes, the first byte (opcode) must be ORed with the address of the register to be read/written.
 * The command is completed with a second byte 000n nnnn, where n nnnn is (# registers to read) - 1.
 */
#define ADS1299_OPC_WAKEUP       0x02           ///< Wake up from standby.
#define ADS1299_OPC_STANDBY      0x04           ///< Enter standby.
#define ADS1299_OPC_RESET        0x06           ///< Reset all registers.
#define ADS1299_OPC_START        0x08           ///< Start data conversions.
#define ADS1299_OPC_STOP         0x0A           ///< Stop data conversions.

#define ADS1299_OPC_RDATAC       0x10           ///< Read data continuously (registers cannot be read or written in this mode).
#define ADS1299_OPC_SDATAC       0x11           ///< Stop continuous data read.
#define ADS1299_OPC_RDATA        0x12           ///< Read single data value.

#define ADS1299_OPC_RREG         0x20           ///< Read register value.
#define ADS1299_OPC_WREG         0x40           ///< Write register value.

/* ID REGISTER ********************************************************************************************************************/

/**
 *  \brief Factory-programmed device ID for ADS1299, stored in ID[3:0].
 */
// Factory-programmed device ID for ADS1299. Stored in ID[3:0].
#define ADS1299_DEVICE_ID       0b1110

/* CONFIG1 REGISTER ***************************************************************************************************************/

/**
 *  \brief Bit location and size definitions for CONFIG1.CLK_EN bit (oscillator output on CLK pin en/disabled).
 *
 * Consult the ADS1299 datasheet, page 40, for more information.
 */
#define ADS1299_REG_CONFIG1_CLOCK_OUTPUT_DISABLED    (0<<5)
#define ADS1299_REG_CONFIG1_CLOCK_OUTPUT_ENABLED     (1<<5)

/**
 *  \brief Bit location and size definitions for CONFIG1.DAISY_EN bit.
 *
 * Consult the ADS1299 datasheet, pp. 40 and 31-34, for more information.
 */
#define ADS1299_REG_CONFIG1_DAISY_CHAIN_MODE         (0<<6)
#define ADS1299_REG_CONFIG1_MULTI_READBACK_MODE      (1<<6)

/**
 *  \brief Bit mask definitions for CONFIG1.DR (data rate).
 *
 * FMOD = FCLK/2, where FCLK is the clock frequency of the ADS1299. This is normally 2.048 MHz.
 */
#define ADS1299_REG_CONFIG1_16kSPS         0      ///< Data is output at FMOD/64, or 16 kHz at 2.048 MHz.
#define ADS1299_REG_CONFIG1_8kSPS          1      ///< Data is output at FMOD/128, or 8 kHz at 2.048 MHz.
#define ADS1299_REG_CONFIG1_4kSPS          2      ///< Data is output at FMOD/256, or 4 kHz at 2.048 MHz.
#define ADS1299_REG_CONFIG1_2kSPS          3      ///< Data is output at FMOD/512, or 2 kHz at 2.048 MHz.
#define ADS1299_REG_CONFIG1_1kSPS          4      ///< Data is output at FMOD/1024, or 1 kHz at 2.048 MHz.
#define ADS1299_REG_CONFIG1_500SPS         5      ///< Data is output at FMOD/2048, or 500 Hz at 2.048 MHz.
#define ADS1299_REG_CONFIG1_250SPS         6      ///< Data is output at FMOD/4096, or 250 Hz at 2.048 MHz.

/**
 *  \brief Combined value of reserved bits in CONFIG1 register.
 *
 * Consult the ADS1299 datasheet, page 40, for more information.
 */
#define ADS1299_REG_CONFIG1_RESERVED_VALUE          (1<<7)|(1<<4)

/* CONFIG2 REGISTER ***************************************************************************************************************/

/**
 *  \brief Bit mask definitions for CONFIG2.CAL_FREQ (calibration signal frequency).
 *
 * FCLK is the clock frequency of the ADS1299. This is normally 20 MHz.
 */
```

```
#define ADS1299_REG_CONFIG2_CAL_PULSE_FCLK_DIV_2_21     0       ///< Calibration signal pulsed at FCLK/2^21, or approx. 1 Hz at 2.048 MHz.
#define ADS1299_REG_CONFIG2_CAL_PULSE_FCLK_DIV_2_20     1       ///< Calibration signal pulsed at FCLK/2^20, or approx. 2 Hz at 2.048 MHz.
#define ADS1299_REG_CONFIG2_CAL_DC                      3       ///< Calibration signal is not pulsed.



/**
 *  \brief Bit mask definitions for CONFIG2.CAL_AMP0 (calibration signal amplitude).
 */
#define ADS1299_REG_CONFIG2_CAL_AMP_VREF_DIV_2_4_MV     (0<<2)      ///< Calibration signal amplitude is 1 x (VREFP - VREFN)/(2.4 mV).
#define ADS1299_REG_CONFIG2_CAL_AMP_2VREF_DIV_2_4_MV    (1<<2)      ///< Calibration signal amplitude is 2 x (VREFP - VREFN)/(2.4 mV).

/**
 *  \brief Bit mask definitions for CONFIG2.INT_CAL (calibration signal source).
 */
#define ADS1299_REG_CONFIG2_CAL_EXT                     (0<<4)      ///< Calibration signal is driven externally.
#define ADS1299_REG_CONFIG2_CAL_INT                     (1<<4)      ///< Calibration signal is driven internally.

/**
 *  \brief Combined value of reserved bits in CONFIG2 register.
 *
 * Consult the ADS1299 datasheet, page 41, for more information.
 */
#define ADS1299_REG_CONFIG2_RESERVED_VALUE              (6<<5)



/* CONFIG3 REGISTER ***********************************************************************************************************************/

/**
 *  \brief Bit mask definitions for CONFIG3.PD_REFBUF (internal voltage reference buffer enable/disable).
 *
 * Note that disabling the buffer for the internal voltage reference requires that a reference voltage
 * must be externally applied on VREFP for proper operation. This is not related to the reference ELECTRODE
 * buffer, which is an external op-amp on the PCB. Brainboard does not apply a voltage to VREFP, and thus
 * the buffer must be enabled.
 */
#define ADS1299_REG_CONFIG3_REFBUF_DISABLED         (0<<7)
#define ADS1299_REG_CONFIG3_REFBUF_ENABLED          (1<<7)

/**
 *  \brief Bit mask definitions for CONFIG3.BIAS_MEAS (enable or disable bias measurement through BIASIN pin).
 */
#define ADS1299_REG_CONFIG3_BIAS_MEAS_DISABLED          (0<<4)
#define ADS1299_REG_CONFIG3_BIAS_MEAS_ENABLED           (1<<4)

/**
 *  \brief Bit mask definitions for CONFIG3.BIASREF_INT (bias reference internally or externally generated).
 */
#define ADS1299_REG_CONFIG3_BIASREF_EXT                 (0<<3)
#define ADS1299_REG_CONFIG3_BIASREF_INT                 (1<<3)

/**
 *  \brief Bit mask definitions for CONFIG3.PD_BIAS (power-down or enable bias buffer amplifier).
 */
#define ADS1299_REG_CONFIG3_BIASBUF_DISABLED            (0<<2)
#define ADS1299_REG_CONFIG3_BIASBUF_ENABLED             (1<<2)

/**
 *  \brief Bit mask definitions for CONFIG3.BIAS_LOFF_SENS (detection of bias lead-off en/disable).
 */
#define ADS1299_REG_CONFIG3_BIAS_LOFF_SENSE_DISABLED    (0<<1)
#define ADS1299_REG_CONFIG3_BIAS_LOFF_SENSE_ENABLED     (1<<1)

/**
 *  \brief Combined value of reserved bits in CONFIG3 register.
 *
 * Consult the ADS1299 datasheet, page 42, for more information.
 */
#define ADS1299_REG_CONFIG3_RESERVED_VALUE              (3<<5)

/* CONFIG4 REGISTER ***********************************************************************************************************************/

/**
 *  \brief Bit mask definitions for CONFIG4.SINGLE_SHOT (single-shot or continuous conversion setting).
 *
 * This can more easily be set with the RDATAC/SDATAC opcodes.
 */
#define ADS1299_REG_CONFIG4_CONTINUOUS_CONVERSION_MODE      (0<<3)
#define ADS1299_REG_CONFIG4_SINGLE_SHOT_MODE                (1<<3)

/**
 *  \brief Bit mask definitions for CONFIG4.PD_LOFF_COMP (power-down lead-off comparators).
 *
 */
#define ADS1299_REG_CONFIG4_LEAD_OFF_DISABLED               (0<<1)
#define ADS1299_REG_CONFIG4_LEAD_OFF_ENABLED                (1<<1)

/**
 *  \brief Combined value of reserved bits in CONFIG4 register.
 *
 * Consult the ADS1299 datasheet, page 47, for more information.
 */
#define ADS1299_REG_CONFIG4_RESERVED_VALUE      0

/* LOFF REGISTER **************************************************************************************************************************/

/**
 *  \brief Bit mask definitions for LOFF.COMP_TH (lead-off comparator threshold).
 *
 * Definition names are for the positive side (LOFFP). The corresponding LOFFN thresholds
 * are the difference between these thresholds and 100%. Default value is _95_PERCENT.
 */
#define ADS1299_REG_LOFF_95_PERCENT                 (0<<5)
#define ADS1299_REG_LOFF_92_5_PERCENT               (1<<5)
#define ADS1299_REG_LOFF_90_PERCENT                 (2<<5)
#define ADS1299_REG_LOFF_87_5_PERCENT               (3<<5)
#define ADS1299_REG_LOFF_85_PERCENT                 (4<<5)
#define ADS1299_REG_LOFF_80_PERCENT                 (5<<5)
#define ADS1299_REG_LOFF_75_PERCENT                 (6<<5)
#define ADS1299_REG_LOFF_70_PERCENT                 (7<<5)

/**
 *  \brief Bit mask definitions for LOFF.ILEAD_OFF (lead-off current magnitude).
 *
 * This should be as small as possible for continuous lead-off detection, so as not to noticeably alter
 * the acquired signal. Default is _6_NA.
 */
#define ADS1299_REG_LOFF_6_NA                       (0<<2)          ///< 6 nA lead-off current.
#define ADS1299_REG_LOFF_24_NA                      (1<<2)          ///< 24 nA lead-off current.
#define ADS1299_REG_LOFF_6_UA                       (2<<2)          ///< 6 uA lead-off current.
#define ADS1299_REG_LOFF_24_UA                      (3<<2)          ///< 24 uA lead-off current.

/**
 *  \brief Bit mask definitions for LOFF.FLEAD_OFF (lead-off current frequency).
 *
 * This should be as large as possible for continuous AC lead-off detection to ensure that it is out
 * of the EEG frequency band (approx. 0-100 Hz for most applications). The excitation signal can then
 * be filtered out of the acquired overall signal, and its voltage amplitude measured in order to determine
 * the electrode impedance.
 * FCLK is the clock frequency of the ADS1299. This is normally 2.048 MHz.
 * FDR is the output data rate. With the default clock, this must be at least 1 kHz in order to use
 * continuous AC impedance monitoring, since the excitation frequency of FDR/4 = 250 Hz is the lowest
 * possible frequency outside of the EEG band. If only a specific band is needed and it is lower than
 * 62.5 Hz or 125 Hz, the 250/500 Hz settings may be used.
 */
#define ADS1299_REG_LOFF_DC_LEAD_OFF                0       ///< Lead-off current is at DC.
#define ADS1299_REG_LOFF_AC_LEAD_OFF_FCLK_DIV_2_18  1       ///< Lead-off current is at FCLK/2^18, or 7.8125 Hz at 2.048 MHz.
#define ADS1299_REG_LOFF_AC_LEAD_OFF_FCLK_DIV_2_16  2       ///< Lead-off current is at FCLK/2^16, or 31.25 Hz at 2.048 MHz.
#define ADS1299_REG_LOFF_AC_LEAD_OFF_FDR_DIV_4      3       ///< Lead-off current is at FDR/4.

/**
 *  \brief Combined value of reserved bits in LOFF register.
 *
 */
#define ADS1299_REG_LOFF_RESERVED_VALUE             0

/* CHnSET REGISTERS ******************************************************************************************************************/

/**
 *  \brief Bit mask definitions for CHnSET.PD (channel power-down).
 */
#define ADS1299_REG_CHNSET_CHANNEL_ON           (0<<7)
#define ADS1299_REG_CHNSET_CHANNEL_OFF          (1<<7)

/**
 *  \brief Bit mask definitions for CHnSET.GAIN (channel PGA gain).
 *
 * Take care to ensure that the gain is appropriate for the common-mode level of the device inputs.
 * Higher gain settings have lower input-referred noise.
 * Consult the ADS1299 datasheet, pages 6-7 and 19-20, for more information.
 */
#define ADS1299_REG_CHNSET_GAIN_1               (0<<4)          ///< PGA gain = 1.
#define ADS1299_REG_CHNSET_GAIN_2               (1<<4)          ///< PGA gain = 2.
#define ADS1299_REG_CHNSET_GAIN_4               (2<<4)          ///< PGA gain = 4.
#define ADS1299_REG_CHNSET_GAIN_6               (3<<4)          ///< PGA gain = 6.
#define ADS1299_REG_CHNSET_GAIN_8               (4<<4)          ///< PGA gain = 8.
#define ADS1299_REG_CHNSET_GAIN_12              (5<<4)          ///< PGA gain = 12.
#define ADS1299_REG_CHNSET_GAIN_24              (6<<4)          ///< PGA gain = 24.

byte gain_mask = 0b111<<4;
byte GAINS[7] = {ADS1299_REG_CHNSET_GAIN_1,
                 ADS1299_REG_CHNSET_GAIN_2,
                 ADS1299_REG_CHNSET_GAIN_4,
                 ADS1299_REG_CHNSET_GAIN_6,
                 ADS1299_REG_CHNSET_GAIN_8,
                 ADS1299_REG_CHNSET_GAIN_12,
                 ADS1299_REG_CHNSET_GAIN_24};
int ADS_GAINS[7] = { 1,
                     2,
                     4,
                     6,
                     8,
                     12,
                     24 };
/**
 *  \brief Bit mask definitions for CHnSET.SRB2 (channel internal connection to SRB2 pin).
 */
#define ADS1299_REG_CHNSET_SRB2_DISCONNECTED    (0<<3)
#define ADS1299_REG_CHNSET_SRB2_CONNECTED       (1<<3)

/**
 *  \brief Bit mask definitions for CHnSET.MUX (channel mux setting).
 *
 * Controls the channel multiplexing on the ADS1299.
 * Consult the ADS1299 datasheet, pages 16-17, for more information.
 */
#define ADS1299_REG_CHNSET_NORMAL_ELECTRODE     0           ///< Channel is connected to the corresponding positive and negative input pins.
#define ADS1299_REG_CHNSET_INPUT_SHORTED        1           ///< Channel inputs are shorted together. Used for offset and noise measurements.
#define ADS1299_REG_CHNSET_BIAS_MEASUREMENT     2           ///< Used with CONFIG3.BIAS_MEAS for bias measurement. See ADS1299 datasheet, pp. 53-54.
```

```
#define ADS1299_REG_CHNSET_MVDD_SUPPLY          3           ///< Used for measuring analog and digital supplies. See ADS1299 datasheet, p. 17.
#define ADS1299_REG_CHNSET_TEMPERATURE_SENSOR   4           ///< Measures device temperature. See ADS1299 datasheet, p. 17.
#define ADS1299_REG_CHNSET_TEST_SIGNAL          5           ///< Measures calibration signal. See ADS1299 datasheet, pp. 17 and 41.
#define ADS1299_REG_CHNSET_BIAS_DRIVE_P         6           ///< Connects positive side of channel to bias drive output.
#define ADS1299_REG_CHNSET_BIAS_DRIVE_N         7           ///< Connects negative side of channel to bias drive output.

/**
 *  \brief Combined value of reserved bits in CHnSET registers.
 *
 */
#define ADS1299_REG_CHNSET_RESERVED_VALUE       0

/* BIAS_SENSP REGISTER ***************************************************************************************************************/

/**
 *  \brief Bit mask definitions for BIAS_SENSP register (read-only).
 *
 * Consult the ADS1299 datasheet, page 44, for more information.
 */
#define ADS1299_REG_BIAS_SENSP_BIASP8   (1<<7)
#define ADS1299_REG_BIAS_SENSP_BIASP7   (1<<6)
#define ADS1299_REG_BIAS_SENSP_BIASP6   (1<<5)
#define ADS1299_REG_BIAS_SENSP_BIASP5   (1<<4)
#define ADS1299_REG_BIAS_SENSP_BIASP4   (1<<3)
#define ADS1299_REG_BIAS_SENSP_BIASP3   (1<<2)
#define ADS1299_REG_BIAS_SENSP_BIASP2   (1<<1)
#define ADS1299_REG_BIAS_SENSP_BIASP1   (1<<0)

byte BIAS_SENSP[7] = {ADS1299_REG_BIAS_SENSP_BIASP1,
                      ADS1299_REG_BIAS_SENSP_BIASP2,
                      ADS1299_REG_BIAS_SENSP_BIASP3,
                      ADS1299_REG_BIAS_SENSP_BIASP4,
                      ADS1299_REG_BIAS_SENSP_BIASP5,
                      ADS1299_REG_BIAS_SENSP_BIASP6,
                      ADS1299_REG_BIAS_SENSP_BIASP8};

/* BIAS_SENSN REGISTER ***************************************************************************************************************/

/**
 *  \brief Bit mask definitions for BIAS_SENSN register (read-only).
 *
 * Consult the ADS1299 datasheet, page 44, for more information.
 */
#define ADS1299_REG_BIAS_SENSN_BIASN8   (1<<7)
#define ADS1299_REG_BIAS_SENSN_BIASN7   (1<<6)
#define ADS1299_REG_BIAS_SENSN_BIASN6   (1<<5)
#define ADS1299_REG_BIAS_SENSN_BIASN5   (1<<4)
#define ADS1299_REG_BIAS_SENSN_BIASN4   (1<<3)
#define ADS1299_REG_BIAS_SENSN_BIASN3   (1<<2)
#define ADS1299_REG_BIAS_SENSN_BIASN2   (1<<1)
#define ADS1299_REG_BIAS_SENSN_BIASN1   (1<<0)

byte BIAS_SENSN[7] = {ADS1299_REG_BIAS_SENSN_BIASN1,
                      ADS1299_REG_BIAS_SENSN_BIASN2,
                      ADS1299_REG_BIAS_SENSN_BIASN3,
                      ADS1299_REG_BIAS_SENSN_BIASN4,
                      ADS1299_REG_BIAS_SENSN_BIASN5,
                      ADS1299_REG_BIAS_SENSN_BIASN6,
                      ADS1299_REG_BIAS_SENSN_BIASN8};

/* LOFF_SENSP REGISTER ***************************************************************************************************************/

/**
 *  \brief Bit mask definitions for LOFF_SENSP register (read-only).
 *
 * Consult the ADS1299 datasheet, page 45, for more information.
 */
#define ADS1299_REG_LOFF_SENSP_LOFFP8   (1<<7)
#define ADS1299_REG_LOFF_SENSP_LOFFP7   (1<<6)
#define ADS1299_REG_LOFF_SENSP_LOFFP6   (1<<5)
#define ADS1299_REG_LOFF_SENSP_LOFFP5   (1<<4)
#define ADS1299_REG_LOFF_SENSP_LOFFP4   (1<<3)
#define ADS1299_REG_LOFF_SENSP_LOFFP3   (1<<2)
#define ADS1299_REG_LOFF_SENSP_LOFFP2   (1<<1)
#define ADS1299_REG_LOFF_SENSP_LOFFP1   (1<<0)



/* LOFF_SENSN REGISTER ***************************************************************************************************************/

/**
 *  \brief Bit mask definitions for LOFF_SENSN register (read-only).
 *
 * Consult the ADS1299 datasheet, page 45, for more information.
 */
#define ADS1299_REG_LOFF_SENSN_LOFFN8   (1<<7)
#define ADS1299_REG_LOFF_SENSN_LOFFN7   (1<<6)
#define ADS1299_REG_LOFF_SENSN_LOFFN6   (1<<5)
#define ADS1299_REG_LOFF_SENSN_LOFFN5   (1<<4)
#define ADS1299_REG_LOFF_SENSN_LOFFN4   (1<<3)
#define ADS1299_REG_LOFF_SENSN_LOFFN3   (1<<2)
#define ADS1299_REG_LOFF_SENSN_LOFFN2   (1<<1)
#define ADS1299_REG_LOFF_SENSN_LOFFN1   (1<<0)

/* LOFF_FLIP REGISTER ***************************************************************************************************************/

/**
 *  \brief Bit mask definitions for LOFF_FLIP register (read-only).
 *
 * Consult the ADS1299 datasheet, page 45, for more information.
 */
#define ADS1299_REG_LOFF_FLIP_LOFF_FLIP8    (1<<7)
#define ADS1299_REG_LOFF_FLIP_LOFF_FLIP7    (1<<6)
#define ADS1299_REG_LOFF_FLIP_LOFF_FLIP6    (1<<5)
#define ADS1299_REG_LOFF_FLIP_LOFF_FLIP5    (1<<4)
#define ADS1299_REG_LOFF_FLIP_LOFF_FLIP4    (1<<3)
#define ADS1299_REG_LOFF_FLIP_LOFF_FLIP3    (1<<2)
#define ADS1299_REG_LOFF_FLIP_LOFF_FLIP2    (1<<1)
#define ADS1299_REG_LOFF_FLIP_LOFF_FLIP1    (1<<0)



/* LOFF_STATP REGISTER ***************************************************************************************************************/

/**
 *  \brief Bit mask definitions for LOFF_STATP register (read-only).
 *
 * Consult the ADS1299 datasheet, page 45, for more information.
 */
#define ADS1299_REG_LOFF_STATP_IN8P_OFF (1<<7)
#define ADS1299_REG_LOFF_STATP_IN7P_OFF (1<<6)
#define ADS1299_REG_LOFF_STATP_IN6P_OFF (1<<5)
#define ADS1299_REG_LOFF_STATP_IN5P_OFF (1<<4)
#define ADS1299_REG_LOFF_STATP_IN4P_OFF (1<<3)
#define ADS1299_REG_LOFF_STATP_IN3P_OFF (1<<2)
#define ADS1299_REG_LOFF_STATP_IN2P_OFF (1<<1)
#define ADS1299_REG_LOFF_STATP_IN1P_OFF (1<<0)

/* LOFF_STATN REGISTER ***************************************************************************************************************/

/**
 *  \brief Bit mask definitions for LOFF_STATN register (read-only).
 *
 * Consult the ADS1299 datasheet, page 45, for more information.
 */
#define ADS1299_REG_LOFF_STATN_IN8N_OFF (1<<7)
#define ADS1299_REG_LOFF_STATN_IN7N_OFF (1<<6)
#define ADS1299_REG_LOFF_STATN_IN6N_OFF (1<<5)
#define ADS1299_REG_LOFF_STATN_IN5N_OFF (1<<4)
#define ADS1299_REG_LOFF_STATN_IN4N_OFF (1<<3)
#define ADS1299_REG_LOFF_STATN_IN3N_OFF (1<<2)
#define ADS1299_REG_LOFF_STATN_IN2N_OFF (1<<1)
#define ADS1299_REG_LOFF_STATN_IN1N_OFF (1<<0)

/* GPIO REGISTER *********************************************************************************************************************/

/**
 *  \brief Bit mask definitions for GPIO.GPIODn (GPIO direction bits).
 *
 * The ADS1299 has 4 GPIO pins that can be manipulated via the SPI bus if there are not enough
 * GPIO pins available on the host.
 * GPIOD[4:1] controls the logic levels on GPIO pins 4:1.
 *
 * Consult the ADS1299 datasheet, page 46, for more information.
 */
#define ADS1299_REG_GPIO_GPIOD4_LOW         (0<<7)
#define ADS1299_REG_GPIO_GPIOD4_HIGH        (1<<7)
#define ADS1299_REG_GPIO_GPIOD3_LOW         (0<<6)
#define ADS1299_REG_GPIO_GPIOD3_HIGH        (1<<6)
#define ADS1299_REG_GPIO_GPIOD2_LOW         (0<<5)
#define ADS1299_REG_GPIO_GPIOD2_HIGH        (1<<5)
#define ADS1299_REG_GPIO_GPIOD1_LOW         (0<<4)
#define ADS1299_REG_GPIO_GPIOD1_HIGH        (1<<4)

/**
 *  \brief Bit mask definitions for GPIO.GPIOCn (GPIO level).
 *
 * The ADS1299 has 4 GPIO pins that can be manipulated via the SPI bus if there are not enough
 * GPIO pins available on the host.
 * GPIOC[4:1] controls the pin direction on GPIO pins 4:1.
 *
 * Consult the ADS1299 datasheet, page 46, for more information.
 */
#define ADS1299_REG_GPIO_GPIOC4_OUTPUT      (0<<3)
#define ADS1299_REG_GPIO_GPIOC4_INPUT       (1<<3)
#define ADS1299_REG_GPIO_GPIOC3_OUTPUT      (0<<2)
#define ADS1299_REG_GPIO_GPIOC3_INPUT       (1<<2)
#define ADS1299_REG_GPIO_GPIOC2_OUTPUT      (0<<1)
#define ADS1299_REG_GPIO_GPIOC2_INPUT       (1<<1)
#define ADS1299_REG_GPIO_GPIOC1_OUTPUT      (0<<0)
#define ADS1299_REG_GPIO_GPIOC1_INPUT       (1<<0)

/**
 *  \brief Combined value of reserved bits in GPIO register.
 *
 */
#define ADS1299_REG_GPIO_RESERVED_VALUE             0

/* MISC1 REGISTER ********************************************************************************************************************/

/**
 *  \brief Bit mask definitions for MISC1.SRB1 (SRB1 internal connection).
 */
#define ADS1299_REG_MISC1_SRB1_OFF      (0<<5)          ///< Stim/ref/bias 1 turned off.
#define ADS1299_REG_MISC1_SRB1_ON       (1<<5)          ///< Stim/ref/bias 1 connected to all channel inverting inputs.

/**
 *  \brief Combined value of reserved bits in MISC1 register.
 *
 */
```

```
#define ADS1299_REG_MISC1_RESERVED_VALUE                0


/* MISC2 REGISTER ******************************************************************************************************************/

/**
 *  \brief Combined value of reserved bits in MISC2 register.
 *
 * MISC2 don't do nothin' right now!
 * Consult the ADS1299 user's guide, page 46, for more information.
 */
#define ADS1299_REG_MISC2_RESERVED_VALUE                0



const byte WAKEUP = 0b00000010;     // Wake-up from standby mode
const byte STANDBY = 0b00000100;   // Enter Standby mode
const byte RESET = 0b00000110;   // Reset the device
const byte START = 0b00001000;   // Start and restart (synchronize) conversions
const byte STOP = 0b00001010;   // Stop conversion
const byte RDATAC = 0b00010000;   // Enable Read Data Continuous mode (default mode at power-up)
const byte SDATAC = 0b00010001;   // Stop Read Data Continuous mode
const byte RDATA = 0b00010010;   // Read data by command; supports multiple read back

//Register Read Commands
const byte RREG = 0b00100000;
const byte WREG = 0b01000000;

const byte CH1 = 0x05;
const byte CH2 = 0x06;
const byte CH3 = 0x07;
const byte CH4 = 0x08;
const byte CH5 = 0x09;
const byte CH6 = 0x0A;
const byte CH7 = 0x0B;
const byte CH8 = 0x0C;
const byte CHn = 0xFF;
```

Pin_Table_Defs.h

```
// ###################### SAMD21 Pin Table ####################

#define PA02 15 // =========== NXR1: A0 - EDA Signal
#define PA04 18 // =========== NXR1: LED Red
#define PA05 19 // =========== NXR1: A4 - VBATT Level
#define PA06 20 // =========== NXR1: D8
#define PA07 21 // =========== NXR1: ADS1299 #0 - CLK
#define PB09 32 // =========== NXR1: A2
#define PB10 4  // =========== NXR1: DRDY
#define PA16 8  // =========== NXR1: ADS1299 #0/#1 - SPI MOSI
#define PA17 9  // =========== NXR1: ADS1299 #0/#1 - SPI SCK
#define PA18 24 // =========== NXR1: USB OTG sense
#define PA19 10 // =========== NXR1: ADS1299 #0/#1 - SPI MISO
#define PA20 6  // =========== NXR1: ADS1299 #1 - SPI CS2
#define PA21 7  // =========== NXR1: ADS1299 #0 - SPI CS1
#define PA22 0  // =========== NXR1: ADS1299 #0 - ext CLK trigger
#define PA23 1  // =========== NXR1: ADS1299 #0 - Reset
#define PB11 5  // =========== NXR1: CLK_SEL

#define PB08 31 // =========== NXR1: uBlox RST
#define PA08 11 // =========== NXR1: uBlox & IMU & Crypto - I2C
#define PA09 12 // =========== NXR1: uBlox & IMU & Crypto - I2C
#define PA12 26 // =========== NXR1: uBlox - TX_MOSI
#define PA13 27 // =========== NXR1: uBlox - RX_MISO
#define PA14 28 // =========== NXR1: uBlox - RTS_CS
#define PA15 29 // =========== NXR1: uBlox - CTS_SCK
#define PA27 30 // =========== NXR1: uBlox - GPIO0
#define PA28 35 // =========== NXR1: uBlox - ACK
#define PB22 14 // =========== NXR1: uBlox - TX
#define PB23 13 // =========== NXR1: uBlox - RX
#define PA10 2

#define PA11 3
#define PB02 16
#define PB03 17
#define PB10 4
#define PA03 25



// // ###################### UBLOX Pin Table ####################



//   //---- MKR1010 / NovaXR v1 --------
//   #define LED_GREEN 25 // =========== MKR1010: uBlox
//   #define LED_BLUE  27 // =========== MKR1010: uBlox
//   #define LED_RED   26 // =========== MKR1010: uBlox

//   //---- NovaXR V1 --------
//  // #define LED_BLUE  18 // =========== NXR: uBlox
//  // #define LED_RED   17 // =========== NXR: uBlox
// // ################### SAMD21 ADC Definitions #############
// #define ADC_READS_SHIFT      8
// #define ADC_READS_COUNT      (1 << ADC_READS_SHIFT)
// #define ADC_MIN_GAIN         0x0400
// #define ADC_UNITY_GAIN       0x0800
// #define ADC_MAX_GAIN         (0x1000 - 1)
// #define ADC_RESOLUTION_BITS  12
// #define ADC_RANGE            (1 << ADC_RESOLUTION_BITS)
// #define ADC_TOP_VALUE        (ADC_RANGE - 1)
// #define MAX_TOP_VALUE_READS  10


// // ################## ADS1299 Definitions #################
// #define ADS1299_ID  0x1E
// #define MASKADC_ADR 0x1F
```

WiFi_Settings.h

```
#include <WiFiNINA.h>
#include <WiFiUdp.h>
#include <Ethernet.h>

// #define SECRET_SSID "raspi_wifi"
// #define SECRET_PASS "fluidfluid"

//#define SECRET_SSID "gitgudbruh"
//#define SECRET_PASS "giganticorchestra203"

#define SECRET_SSID "#8103"
#define SECRET_PASS "1423qweasd"
#define HOST_ID     "10.0.0.74"
#define PORT_NUM    8899

#define SEND_SIZE 22
// 1 for serial count,
// 1 for valid array for the data packet (1 maps to data means invalid)
// 8 for ADS data, ( - status bits, incorporated into valid array)
// 6 for IMU data,
// 1 for EDA
// 1 for temperature
// 1 for PPG
#define NUM_ELEMENTS 17
#define ELEM_SIZE 4
#define PACKET_SIZE (ELEM_SIZE*NUM_ELEMENTS)

// INDICES of PACKET WHERE EACH CATEGORY STARTS
#define i_VALID 1
#define i_ADS i_VALID + 1
#define i_IMU i_ADS + 8
#define i_EDA i_IMU + (3) /* IMU is 6 points, each of which is only 2 bytes */
#define i_TEM i_EDA + 1
#define i_PPG i_TEM + 1
#define i_TIM i_PPG + 1

// Indices of the valid bit for each of these data
// (this is not the same as the indices above due to the size
//  difference between the data-- the IMU data is only 2 bytes,
//  but are 6 differernt data points)
#define v_ADS 2
#define v_IMU v_ADS + 8
#define v_EDA v_IMU + 6
#define v_TEM v_EDA + 1
#define v_PPG v_TEM + 1
#define v_TIM v_PPG + 1
```

Fascia_collect_sensor_data.ino

```
// library includes
#include "SAMD_AnalogCorrection.h"
#include <SPI.h>
#include "wiring_private.h"
//#include "I2Cdev.h"//
#include "MPU6050.h"
#include "Wire.h"//
#include "MAX30105.h"
// header files
#include "Pin_Table_Defs.h"
#include "WiFi_Settings.h"
#include "ads1299.h"

// define enum for boards and data retrieval types
enum board_types {NOVA_XR_V1, NOVA_XR_V2_SISTER, NOVA_XR_V2_MAIN, FASCIA_V0_0, FASCIA_V0_1};
enum data_mode_t {RDATA_CC_MODE, RDATA_SS_MODE};
enum run_mode_t  {GEN_TEST_SIGNAL, NORMAL_ELECTRODES};

// settings
#define CONNECT_WIFI 1
```

```
#define BOARD_V FASCIA_V0_0
#define DATA_MODE RDATA_SS_MODE
#define RUN_MODE NORMAL_ELECTRODES
// v for verbose: lots of prints
#define v 0
// debug: serial reads and writes
#define debug 0
// REMEMBER: comment out lines in setup() and loop() for the sensors you do not have.

// Setup for SPI communications
SPIClass mySPI (&sercom1, PA19, PA17, PA16, SPI_PAD_0_SCK_1, SERCOM_RX_PAD_3);
const int SPI_CLK = 4*pow(10,6) ; //4MHz
// Setup for I2C communications
// PA08: SDA
// PA09: SCL
// MPU6050 I2C address: 0x110100X where X is the logic level in pin AD0
// and last bit is r/w
#define MPU_ADDR 0b11010000

// define pins depending on boards
int pLED;
int pBAT;
// ADS1299 ADC pins
const int pRESET = PA23;    // reset pin
int pCS;               // chip select pin
int pDRDY;             // data ready pin
// EDA
const int pEDA = PA02;
// MPU6050 IMU pins
MPU6050 accelgyro;
const int pMPUint = PB03;
// PPG & temperature pins
MAX30105 particleSensor;

// define variables
int print_ch = -1;
bool LEDval = LOW;

int cnt = 0;

// ADS channel gains
byte ADS_CHANNEL_GAINS[8] = {/*chan 1 EMG Gain 4  */ ADS1299_REG_CHNSET_GAIN_4,
                             /*chan 2 EMG Gain 4  */ ADS1299_REG_CHNSET_GAIN_4,
                             /*chan 3 EOG Gain 2  */ ADS1299_REG_CHNSET_GAIN_4,
                             /*chan 4 EMG Gain 4  */ ADS1299_REG_CHNSET_GAIN_4,
                             /*chan 5 EEG Gain 12 */ ADS1299_REG_CHNSET_GAIN_12, // Passive Electrode
                             /*chan 6 EEG Gain 12 */ ADS1299_REG_CHNSET_GAIN_12, // Passive Electrode
                             /*chan 7 EEG Gain 1  */ ADS1299_REG_CHNSET_GAIN_1,  // Active  Electrode
                             /*chan 8 EEG Gain 1  */ ADS1299_REG_CHNSET_GAIN_1   // Active  Electrode
                             };

void select_board_pins(void) {
    switch (BOARD_V){
    case NOVA_XR_V1: {
        Serial.println("initializing pins for Nova XR V1");
        pCS = PA21;
        pDRDY = PA10;
        break;
    }
    case NOVA_XR_V2_MAIN: {
        Serial.println("initializing pins for Nova XR V2 main board");
        pCS = PA21;
        pDRDY = PB10;
        // disable other ADS
        pinMode(PA20, OUTPUT);
        digitalWrite(PA20, HIGH);
        break;
    }
    case NOVA_XR_V2_SISTER: {
      Serial.println("initializing pins for Nova XR V2 sister board");
      pCS = PA20;
      pDRDY = PB02;
      // disable other ADS
      pinMode(PA21, OUTPUT);
      digitalWrite(PA21, HIGH);
      break;
    }
    case FASCIA_V0_0: {
        Serial.println("initializing pins for Fascia V0.0");
        pCS = PA20;
        pDRDY = PA07;
        pLED = PA04;
        pBAT = PA05;
        break;
    }
    case FASCIA_V0_1: {
        Serial.println("initializing pins for Fascia V0.1");
        pCS = PA20;
        pDRDY = PB10;
        break;
    }
  }
}

void initialize_pin_modes(void) {
  pinMode(pCS, OUTPUT);
  pinMode(pDRDY, INPUT);
  pinMode(pRESET, OUTPUT);
  pinMode(pEDA, INPUT);
  pinMode(pLED, OUTPUT);
  pinMode(pBAT, INPUT);
  pinPeripheral(PA19, PIO_SERCOM);
  pinPeripheral(PA17, PIO_SERCOM);
  pinPeripheral(PA16, PIO_SERCOM);
  if (DATA_MODE == RDATA_CC_MODE) {
    // attachInterrupt(pDRDY, DRDY_ISR, FALLING); // TODO figure out interrupt with input. malloc maybe?
  }
  // attachInterrupt(pMPUint, get_gyro_data, MODEDEODEOD);
}

void ADS_connect(void) {
  //enable ADS
  digitalWrite(pRESET, HIGH);
  digitalWrite(pCS, HIGH);
  delay(500);
  // select and reset ADS
  digitalWrite(pCS, LOW);
    mySPI.beginTransaction(SPISettings(SPI_CLK, MSBFIRST, SPI_MODE1));
    mySPI.transfer(RESET);
    mySPI.endTransaction();
    delay(100);
  digitalWrite(pCS, HIGH);
  delay(50);
  // read and print ADS device ID to ensure connection
  byte idval = ADS_RREG(0x0,1);
  Serial.print("connected to ADS device id "); Serial.println(idval,BIN);
  if (idval != 0x3E) {
    Serial.println("ADS device not properly connected");
    Serial.println("If the ADS ID number is 0xFF, you likely have an issue with power");
    while(1) {    Serial.println("ADS device not properly connected");}
  }
}

void ADS_init(void) {
  // register map on page 44 of the data sheet;
  // pages expading on register descriptions follow in the next pages
  byte config2_data = ADS1299_REG_CONFIG2_RESERVED_VALUE | //0b11000000;
                      ADS1299_REG_CONFIG2_CAL_PULSE_FCLK_DIV_2_21;
  byte config3_data = ADS1299_REG_CONFIG3_REFBUF_ENABLED | //0b11101100;
                      ADS1299_REG_CONFIG3_RESERVED_VALUE |
                      ADS1299_REG_CONFIG3_BIAS_MEAS_DISABLED |
                      ADS1299_REG_CONFIG3_BIASREF_INT |
                      ADS1299_REG_CONFIG3_BIASBUF_ENABLED |
                      ADS1299_REG_CONFIG3_BIAS_LOFF_SENSE_DISABLED;
  byte channel_mode = ADS1299_REG_CHNSET_NORMAL_ELECTRODE;

  if (RUN_MODE == GEN_TEST_SIGNAL) {
    config2_data = ADS1299_REG_CONFIG2_RESERVED_VALUE | //0b11010000;
                   ADS1299_REG_CONFIG2_CAL_INT |
                   ADS1299_REG_CONFIG2_CAL_AMP_2VREF_DIV_2_4_MV |
                   ADS1299_REG_CONFIG2_CAL_PULSE_FCLK_DIV_2_21;
    config3_data = ADS1299_REG_CONFIG3_REFBUF_ENABLED | //0b11100000;
                   ADS1299_REG_CONFIG3_RESERVED_VALUE |
                   ADS1299_REG_CONFIG3_BIAS_MEAS_DISABLED |
                   ADS1299_REG_CONFIG3_BIASREF_EXT |
                   ADS1299_REG_CONFIG3_BIASBUF_DISABLED |
                   ADS1299_REG_CONFIG3_BIAS_LOFF_SENSE_DISABLED;
    channel_mode = ADS1299_REG_CHNSET_TEST_SIGNAL;
  }

  ADS_WREG(ADS1299_REGADDR_CONFIG1, ADS1299_REG_CONFIG1_RESERVED_VALUE |
                                    ADS1299_REG_CONFIG1_2kSPS); // last three bits is the data rate page 46 of data sheet
  ADS_WREG(ADS1299_REGADDR_CONFIG2, config2_data);
  ADS_WREG(ADS1299_REGADDR_CONFIG3, config3_data);
  ADS_WREG(ADS1299_REGADDR_CONFIG4, 0x00);//0b00001000);
  ADS_WREG(ADS1299_REGADDR_GPIO, 0x00);
  ADS_WREG(ADS1299_REGADDR_MISC1, 0x00);

  delay(10);
  ADS_WREG(ADS1299_REGADDR_CH1SET, ADS_CHANNEL_GAINS[0] |
                                   channel_mode |
                                   ADS1299_REG_CHNSET_CHANNEL_ON |
                                   ADS1299_REG_CHNSET_SRB2_DISCONNECTED);
  ADS_WREG(ADS1299_REGADDR_CH2SET, ADS_CHANNEL_GAINS[1] |
                                   channel_mode |
                                   ADS1299_REG_CHNSET_CHANNEL_ON |
                                   ADS1299_REG_CHNSET_SRB2_DISCONNECTED);
  ADS_WREG(ADS1299_REGADDR_CH3SET, ADS_CHANNEL_GAINS[2] |
                                   channel_mode |
                                   ADS1299_REG_CHNSET_CHANNEL_ON |
                                   ADS1299_REG_CHNSET_SRB2_DISCONNECTED);
  ADS_WREG(ADS1299_REGADDR_CH4SET, ADS_CHANNEL_GAINS[3] |
                                   channel_mode |
                                   ADS1299_REG_CHNSET_CHANNEL_ON |
                                   ADS1299_REG_CHNSET_SRB2_DISCONNECTED);
  ADS_WREG(ADS1299_REGADDR_CH5SET, ADS_CHANNEL_GAINS[4] |
                                   channel_mode |
                                   ADS1299_REG_CHNSET_CHANNEL_ON |
                                   ADS1299_REG_CHNSET_SRB2_CONNECTED);
  ADS_WREG(ADS1299_REGADDR_CH6SET, ADS_CHANNEL_GAINS[5] |
                                   channel_mode |
                                   ADS1299_REG_CHNSET_CHANNEL_ON |
                                   ADS1299_REG_CHNSET_SRB2_CONNECTED);
  ADS_WREG(ADS1299_REGADDR_CH7SET, ADS_CHANNEL_GAINS[6] |
                                   channel_mode |
                                   ADS1299_REG_CHNSET_CHANNEL_ON |
                                   ADS1299_REG_CHNSET_SRB2_CONNECTED);
  ADS_WREG(ADS1299_REGADDR_CH8SET, ADS_CHANNEL_GAINS[7] |
                                   channel_mode |
                                   ADS1299_REG_CHNSET_CHANNEL_ON |
                                   ADS1299_REG_CHNSET_SRB2_CONNECTED);

 // turn on bias for all EEG Channels (5-8)
 ADS_WREG(ADS1299_REGADDR_BIAS_SENSN,  ADS1299_REG_BIAS_SENSN_BIASN8 |
                                       ADS1299_REG_BIAS_SENSN_BIASN7 |
                                       ADS1299_REG_BIAS_SENSN_BIASN6 |
                                       ADS1299_REG_BIAS_SENSN_BIASN5 );

  // set up Lead-Off detection
  /* from page 63 in the manual:
   *   10.1.2.1 Lead-Off
   *   Sample code to set dc lead-off with pull-up and pull-down resistors on all channels.
   *     WREG LOFF       0x13  // Comparator threshold at 95% and 5%, pullup or pulldown resistor dc lead-off
   *     WREG CONFIG4    0x02  // Turn on dc lead-off comparators
   *     WREG LOFF_SENSP 0xFF  // Turn on the P-side of all channels for lead-off sensing
   *     WREG LOFF_SENSN 0xFF  // Turn on the N-side of all channels for lead-off sensing
   *   Observe the status bits of the output data stream to monitor lead-off status.
  */
//  ADS_WREG(ADS1299_REGADDR_LOFF,       0x13);
//  ADS_WREG(ADS1299_REGADDR_CONFIG4,    0x02);
//  ADS_WREG(ADS1299_REGADDR_LOFF_SENSP, 0x0F);
//  ADS_WREG(ADS1299_REGADDR_LOFF_SENSN, 0xFF);
//  // TODO make sure this below works
//  ADS_WREG(ADS1299_REGADDR_LOFF_FLIP,  0xF0); // flip the EEG channels since we are connecting them to the N end
}

void ADS_start(void) {
  digitalWrite(pCS, LOW);
  mySPI.beginTransaction(SPISettings(SPI_CLK, MSBFIRST, SPI_MODE1));
  mySPI.transfer(START);
  if (DATA_MODE == RDATA_SS_MODE) {
    mySPI.transfer(RDATA);
  } else if (DATA_MODE == RDATA_CC_MODE) {
    mySPI.transfer(RDATAC);
  }
}

void Arduino_ADC_setup() {
  //https://forum.arduino.cc/index.php?topic=443173.0
  //http://yaab-arduino.blogspot.com/2015/02/fast-sampling-from-analog-input.html
  //https://forum.arduino.cc/index.php?topic=6549.0

    // original SAMD bootloader code set to:
    //  ADC->CTRLB.reg = ADC_CTRLB_PRESCALER_DIV512 |    // Divide Clock by 512.
    //                   ADC_CTRLB_RESSEL_10BIT;         // 10 bits resolution as default
    ADC->CTRLB.reg = ADC_CTRLB_PRESCALER_DIV16 | ADC_CTRLB_RESSEL_12BIT;
    Serial.println("done setting up ADC with lower prescaler value and higher bit resolution");
}

void setup_MAX30105() {
  if (!particleSensor.begin(Wire, I2C_SPEED_FAST)) {//Use default I2C port, 400kHz speed
    while (1){Serial.println("MAX30105 was not found. Please check wiring/power. ");}
  }
  //The LEDs are very low power and won't affect the temp reading much but
  //you may want to turn off the LEDs to avoid any local heating
  // can try setting data output rate (currently (default) close to the slowest)
  particleSensor.setup(/*byte powerLevel = */0x1F, /*byte sampleAverage = */4, /*byte ledMode =*/ 3, /*int sampleRate =*/3200); //Configure sensor. Turn 
off LEDs
  // TODO increase sampling rate here!!!!

  //particleSensor.setup(); //Configure sensor. Use 25mA for LED drive
  //TODO seems like line below is not actually necessary
  particleSensor.enableDIETEMPRDY(); //Enable the temp ready interrupt. This is required.
  // particleSensor.setPulseAmplitudeGreen(0); //Turn off Green LED
  particleSensor.setPulseAmplitudeRed(0x0A); //Turn Red LED to low to indicate sensor is running
  Serial.println("done setting up MAX30105");
}

void setup_MPU6050() {
  accelgyro.initialize();
  // Test connection
  if (!accelgyro.testConnection()) {
    while (1){Serial.println("Failed to connect to MPU6050");}
  }

  // use the code below to print before/after and change accel/gyro offset values
  /*
  Serial.println("Updating internal sensor offsets...");
  accelgyro.setXGyroOffset(220);
  accelgyro.setYGyroOffset(76);
  accelgyro.setZGyroOffset(-85);
  */

  /*
  // data rate change?
  // SMPLRT_DIV register
  uint8_t getRate();
  void setRate(uint8_t rate);*/

  /*
    // Calibration Routines
    void CalibrateGyro(uint8_t Loops = 15); // Fine tune after setting offsets with less Loops.
    void CalibrateAccel(uint8_t Loops = 15);// Fine tune after setting offsets with less Loops.
    */
    Serial.println("Done setting up MPU6050");
}

void setup() {
  delay(1000);
  // initialize communications: spi, I2C, serial, and wifi if applicable
  mySPI.begin();
  Wire.begin();
  Serial.begin(115200);
  // set up indicator LED
  select_board_pins();
  initialize_pin_modes();
  digitalWrite(pLED, HIGH);

  #if (CONNECT_WIFI)
    setupWifi();
  #endif

  // initialize board and settings
  // make sure settings are compatible
  if (BOARD_V == FASCIA_V0_0 && DATA_MODE == RDATA_CC_MODE) {
    Serial.println("You cannot use Continuous Conversion mode on Fascia Version 0.0");
    Serial.println("The DRDY pin on this PCB (PA07) cannot be configured as an interrupt pin");
    while(1);
  }

  // speed up analog read speed
  Arduino_ADC_setup();

  // initialize MAX30105 PPG sensor (we will also be getting temperature data from it)
//  setup_MAX30105();

  // initialize the IMU MPU6050
  setup_MPU6050();

  // initialize ads1299
  ADS_connect();
  ADS_init();
  ADS_start();
  Serial.println("Done with setup.");
  digitalWrite(pLED, LOW);
  #if debug
    print_serial_instructions();
  #endif
}

void print_serial_instructions() {
  Serial.println("Type the channel number to print that channel's data [1-8] (and plot, if you switch to Serial Plotter)");
  Serial.println("Or type '0' to stop printing the data.");
  Serial.println("type BN#0 to deactivate biasN for channel # and BN#1 to activate it");
  Serial.println("type BP#0 to deactivate biasP for channel # and BP#1 to activate it");
  Serial.println("type S#0 to deactivate SRB2 for channel # and B#1 to activate it");
  Serial.println("type G#N to set the gain for channel # to N=0:1, N=1:2, N=2:4, N=3:6, N=4:8, N=5:12, N=6:24");
  Serial.println("type T#0 to toggle channel # off, and T#1 to toggle channel # on");
  Serial.println("type 'R' or 'r' to print the current register settings of the ADS1299");
  Serial.println("type 'P' or 'p' to print these instructions again");
}

void loop() {
  #if debug
    if(Serial.available()>1){
      parse_serial_input();
    }
  #endif

  long packet[NUM_ELEMENTS];
  packet[i_VALID] = 0;

  #if v
    Serial.println("packet #"+String(cnt));
  #endif

  #if DATA_MODE == RDATA_SS_MODE
    DRDY_ISR(packet);
  #endif
//  get_EDA_data(packet);
  if (!(cnt%10)) {
//    get_PPG_temp_data(packet);
    get_EDA_data(packet);
    get_IMU_data(packet);
  } else {
    packet[i_VALID] |= (1<<v_PPG);
    packet[i_VALID] |= (1<<v_TEM);
    packet[i_VALID] |= (1<<v_EDA);
    packet[i_VALID] |= (0b111111<<v_IMU);
  }

  #if CONNECT_WIFI
    pushToBuf((char *)packet);
    sendWiFiDataPacket();
  #endif
}

void DRDY_ISR(long* packet) {
  //get all data before sign extending etc
  digitalWrite(pCS, LOW);
  if (DATA_MODE == RDATA_SS_MODE) {
    while(digitalRead(pDRDY));
    // Serial.println("DRDY just went low!");
    mySPI.transfer(START);
    mySPI.transfer(RDATA);
  }

  // first, read status bytes
  byte b1 = mySPI.transfer(0x00);
  byte b2 = mySPI.transfer(0x00);
  byte b3 = mySPI.transfer(0x00);
//  Serial.println(b1,BIN);
//  Serial.println(b2,BIN);
//  Serial.println(b3,BIN);
  // figure out if the data is valid (if so, which)
  if ((b1 & 0xF0) == 0b11000000) {
    byte loff_p =  (b1<<4) | (b2>>4);
    byte loff_n =  (b2<<4) | (b3>>4);
//    Serial.print("loff_p: ");    Serial.println(loff_p,BIN);
//    Serial.print("loff_n: ");    Serial.println(loff_n,BIN);
    // TODO use loff-p and loff-n to figure out which channels might be invalid
    // the first 4 channels are EMG and EOG- they use both leads
    int i = 0;
    for (i; i < 4; i++) {
//      packet[i_VALID] |= (((loff_p | loff_n) >> i) & 1) << (v_ADS + i);
    }
//    Serial.print("packet[i_VALID]: ");    Serial.println(packet[i_VALID],BIN);
    // the last 4 channels are all EEG, they use only the N inputs
    for (i; i < 8; i++) {
    }
  } else {
    Serial.println("invalid ADS1299 packet");
    for (int i = 3; i < 27; i++){
      mySPI.transfer(0x00);
      packet[i_VALID] |= 1 << (i_ADS + (i/3-1));
    }
    return;
  }

  // read channel data and sign extend it if valid
  byte temp[4] = {0,0,0,0};
  for (int i = 3; i < 27; i++){
    temp[2-((i+3)%3)] = mySPI.transfer(0x00);//DOUT[i];
    if ((i+3)%3 == 2) {
      int32_t d = *((int32_t*)temp);
      int ed = SIGN_EXT_24(d);//SIGNEXTEND(d);
      float cd = convert_ADC_volts(ed, ADS_GAINS[((ADS_CHANNEL_GAINS[i/3-1])>>4)]);
//      Serial.println(String(i/3-1)+": "+String(ADS_GAINS[((ADS_CHANNEL_GAINS[i/3-1])>>4)]));
      packet[i_ADS + (i/3-1)] = *((long*)&cd);//ed;
      #if v
        Serial.print("ADS ");Serial.print(i/3);Serial.print(" ");Serial.println(ed);
//        Serial.print("ADS ");Serial.print(i/3);Serial.print(" ");Serial.println(packet[i_ADS + i/3]);
      #endif
      if (i/3 == (print_ch)) {Serial.println(ed);}
    }
  }
}

float convert_ADC_volts(int raw_data, int gain) {
  float vref = 4.5;
  float fs = vref / gain;
  float converted_data = fs * raw_data / (pow(2,23)-1);
  // Serial.print(raw_data);Serial.print(" -> ");Serial.println(converted_data);
  return converted_data;
}

// returns the last byte read
byte ADS_RREG(byte r , int n) {
  if (r + n - 1 > 24)
    n = 24 - r ;
  digitalWrite(pCS, LOW);
  // Serial.print("Register "); Serial.print(r, HEX); Serial.println(" Data");
  mySPI.beginTransaction(SPISettings(SPI_CLK, MSBFIRST, SPI_MODE1));
  mySPI.transfer(SDATAC);
  mySPI.transfer(RREG | r); //RREG
  mySPI.transfer(n-1); // 24 Registers
  byte to_ret;
  for (int i = 0; i < n; i++)
  { byte temp = mySPI.transfer(0x00);
//    Serial.println(temp, HEX);
    if ((n-i) == 1) to_ret = temp;
  }
  mySPI.endTransaction();
  digitalWrite(pCS, HIGH);
  return to_ret;
}

void ADS_WREG(byte r, byte d) {
  if (r == 0 || r == 18 || r == 19)
    Serial.println("Error: Read-Only Register");
  else
  { digitalWrite(pCS, LOW);
    mySPI.beginTransaction(SPISettings(SPI_CLK, MSBFIRST, SPI_MODE1));
    mySPI.transfer(SDATAC);
    mySPI.transfer(WREG | r); //RREG
    mySPI.transfer(0x00); // 24 Registers
    mySPI.transfer(d);
    mySPI.endTransaction();
    digitalWrite(pCS, HIGH);
    Serial.print("Wrote ");Serial.print(d, BIN); Serial.print(" to Register "); Serial.println(r, HEX);
  }
}

void parse_serial_input() {
  //TODO consider this
  // char* s = Serial.readStringUntil('\n');
  char c = Serial.read();
  char p;
  // if char is '0' - '8'
  if (c >= 0x30 && c <= 0x38) {
    print_ch = (c -0x30);
    //         ('#'-> #) -> #-1 to index channels
    Serial.print("changed printing channel to ");Serial.println(print_ch);
    return;
  }
  switch (c) {
  case 'B':
  case 'b':
    p = Serial.read();
    c = Serial.read();
    if (c >= 0x31 && c <= 0x38) {
      if (p == 'p' || p == 'P') change_channel_biasP(c-0x30-1);
      if (p == 'n' || p == 'N') change_channel_biasN(c-0x30-1);
    }
    break;
  case 'S':
  case 's':
    c = Serial.read();
    if (c >= 0x31 && c <= 0x38) {
      change_channel_SRB2(c-0x30-1);
    }
    break;
  case 'G':
  case 'g':
    c = Serial.read();
    if (c >= 0x31 && c <= 0x38) {
      change_channel_gain(c-0x30-1);
    }
    break;
  case 'T':
  case 't':
    c = Serial.read();
    if (c >= 0x31 && c <= 0x38) {
      toggle_channel(c-0x30-1);
    }
    break;
  case 'p':
  case 'P':
    print_serial_instructions();
    break;
  case 'r':
  case 'R':
    print_ADS_reg_settings();
    break;
  case 0xA:
    break;
  default:
    Serial.println(c, HEX);
    Serial.println("!!invalid input");
    break;
  }
}

void change_channel_SRB2(int chan){
  char c = Serial.read();
  Serial.print("CHANNELS[chan] ");Serial.println(CHANNELS[chan],HEX);
  byte old_val = ADS_RREG(CHANNELS[chan], 1);
  byte change = 0;
  byte new_val;
  if (c == '1') {
    change = ADS1299_REG_CHNSET_SRB2_CONNECTED;
    new_val = old_val | change;
  } else if (c == '0'){
    change = 0xFF ^ ADS1299_REG_CHNSET_SRB2_CONNECTED;
    new_val = old_val & change;
  } else {
    Serial.println("invalid input");return;
  }
  Serial.print("changing SRB2 of channel "); Serial.print(chan);Serial.print(" to be ");Serial.println(c);
  Serial.println(change,BIN);
  Serial.print(old_val, BIN);Serial.print(" -> ");Serial.println(new_val, BIN);
  ADS_WREG(CHANNELS[chan], new_val);
  // START CONVERSION AGAIN
  if (DATA_MODE == RDATA_CC_MODE) {
    digitalWrite(pCS, LOW);
    mySPI.transfer(START);
    mySPI.transfer(RDATAC);
  }
}

void change_channel_biasN(int chan){
  char c = Serial.read();
  // Serial.print("CHANNELS[chan] ");Serial.println(CHANNELS[chan],HEX);
  byte old_val = ADS_RREG(ADS1299_REGADDR_BIAS_SENSN, 1);
  byte change = 0;
  byte new_val;
  if (c == '1') {
    change = BIAS_SENSN[chan];
    new_val = old_val | change;
  } else if (c == '0'){
    change = 0xFF ^ BIAS_SENSN[chan];
    new_val = old_val & change;
  } else {
    Serial.println("invalid input");return;
  }
  Serial.print("changing biasN of channel "); Serial.print(chan);Serial.print(" to be ");Serial.println(c);
  Serial.println(change,BIN);
  Serial.print(old_val, BIN);Serial.print(" -> ");Serial.println(new_val, BIN);
  ADS_WREG(ADS1299_REGADDR_BIAS_SENSN, new_val);
  // START CONVERSION AGAIN
  if (DATA_MODE == RDATA_CC_MODE) {
    digitalWrite(pCS, LOW);
    mySPI.transfer(START);
    mySPI.transfer(RDATAC);
  }
}

void change_channel_biasP(int chan){
  char c = Serial.read();
  // Serial.print("CHANNELS[chan] ");Serial.println(CHANNELS[chan],HEX);
  byte old_val = ADS_RREG(ADS1299_REGADDR_BIAS_SENSP, 1);
  byte change = 0;
  byte new_val;
  if (c == '1') {
    change = BIAS_SENSP[chan];
    new_val = old_val | change;
  } else if (c == '0'){
    change = 0xFF ^ BIAS_SENSP[chan];
    new_val = old_val & change;
  } else {
    Serial.println("invalid input");return;
  }
  Serial.print("changing biasP of channel "); Serial.print(chan);Serial.print(" to be ");Serial.println(c);
  Serial.println(change,BIN);
  Serial.print(old_val, BIN);Serial.print(" -> ");Serial.println(new_val, BIN);
  ADS_WREG(ADS1299_REGADDR_BIAS_SENSP, new_val);
  // START CONVERSION AGAIN
  if (DATA_MODE == RDATA_CC_MODE) {
    digitalWrite(pCS, LOW);
    mySPI.transfer(START);
    mySPI.transfer(RDATAC);
  }
}

void change_channel_gain(int chan){
  Serial.print("CHANNELS[chan] ");Serial.println(CHANNELS[chan],HEX);
  char c = Serial.read();
  byte old_val = ADS_RREG(CHANNELS[chan], 1);
  byte gain = 0;
  byte new_val;
  if (c >= 0x30 && c <= 0x36) {
    gain = GAINS[c-0x30];
    new_val = (old_val & (~gain_mask)) | gain;
  } else {
    Serial.println("invalid input");return;
  }
  Serial.print("changing gain of channel "); Serial.print(chan);Serial.print(" to be ");Serial.println(c-0x30,BIN);
  Serial.println(gain,BIN);
  Serial.print(old_val, BIN);Serial.print(" -> ");Serial.println(new_val, BIN);
  ADS_WREG(CHANNELS[chan], new_val);
  // START CONVERSION AGAIN
  if (DATA_MODE == RDATA_CC_MODE) {
    digitalWrite(pCS, LOW);
    mySPI.transfer(START);
    mySPI.transfer(RDATAC);
  }
}

void toggle_channel(int chan){
  Serial.print("CHANNELS[chan] ");Serial.println(CHANNELS[chan],HEX);
  char c = Serial.read();
  byte old_val = ADS_RREG(CHANNELS[chan], 1);
  byte new_val;
  if (c == '1') {
    new_val = old_val & 0x7F;
  } else if (c == '0'){
    new_val = old_val | 0x80;
  } else {
    Serial.println("invalid input");return;
  }
  Serial.print("turning channel "); Serial.print(chan);Serial.print(" to be ");
  if(c=='1') Serial.println("on");
  else       Serial.println("off");
  Serial.print(old_val, BIN);Serial.print(" -> ");Serial.println(new_val, BIN);
  ADS_WREG(CHANNELS[chan], new_val);
  // START CONVERSION AGAIN
  if (DATA_MODE == RDATA_CC_MODE) {
    digitalWrite(pCS, LOW);
    mySPI.transfer(START);
    mySPI.transfer(RDATAC);
  }
}

void print_ADS_reg_settings() {
  for(uint8_t address =0; address<24; address++){
    Serial.print("Register Address: 0x"); Serial.print(address,HEX);
    Serial.print("\t");
    Serial.print(ADS_reg_names[address]);
    Serial.print("\t");
    if(!(address>12 && address <20)) Serial.print("\t");
    byte data = ADS_RREG(address,1);
    Serial.print("Register Data: 0x"); Serial.print(data, HEX);
    Serial.print("\t");
    Serial.print("0b"); Serial.print(data, BIN);
    Serial.println();
  }

//  // START CONVERSION AGAIN
//  if (DATA_MODE == RDATA_CC_MODE) {
//    digitalWrite(pCS, LOW);
//    mySPI.transfer(START);
//    mySPI.transfer(RDATAC);
//  }
}

inline void get_IMU_data(long* packet){
  int16_t ax, ay, az, gx, gy, gz;
  accelgyro.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
  // NEED to convert from raw data to some underestandable units

  // insert imy data into packet
  ((int16_t*)&(packet[i_IMU]))[0] = ax;
  ((int16_t*)&(packet[i_IMU]))[1] = ay;
  ((int16_t*)&(packet[i_IMU]))[2] = az;
  ((int16_t*)&(packet[i_IMU]))[3] = gx;
  ((int16_t*)&(packet[i_IMU]))[4] = gy;
  ((int16_t*)&(packet[i_IMU]))[5] = gz;

  #if v
    for (int i = 0; i < 6; i++) {
//      Serial.println("IMU "+String(i)+" "+String( ((int16_t*)packet)[(2*i_IMU)+i])); //wrong indexing
      Serial.println("IMU "+String(i)+" "+String( ((int16_t*)&(packet[i_IMU]))[i]));
    }
  #endif



  /*  From MPU6050 register maps (pg 31), Gyroscope:
   *
   * FS_SEL     Full Scale Range      LSB Sensitivity
   *    0           ± 250 °/s           131 LSB/°/s      [DEFAULT]
   *    1           ± 500 °/s           65.5 LSB/°/s
   *    2           ± 1000 °/s          32.8 LSB/°/s
   *    3           ± 2000 °/s          16.4 LSB/°/s
   */
  // conversion: g/131 = # °/s

  /*  From MPU6050 register maps (pg 29) , Accelerometer:
   *
   * FS_SEL     Full Scale Range      LSB Sensitivity
   *    0           ± 2 g               16384 LSB/g       [DEFAULT]
   *    1           ± 4 g               8192 LSB/g
   *    2           ± 8 g               4096 LSB/g
   *    3           ± 16 g              2048 LSB/g
   */
  // conversion: a/16384 = #g *9.81 = # m/s^2

/*
  // MOT_DETECT_STATUS register
  uint8_t getMotionStatus();
  bool getXNegMotionDetected();
  bool getXPosMotionDetected();
  bool getYNegMotionDetected();
  bool getYPosMotionDetected();
  bool getZNegMotionDetected();
  bool getZPosMotionDetected();
  bool getZeroMotionDetected();
*/
}

// number of eda data points to use to average
int eda_avg_size = 10;
// current index of the value being added
int eda_idx = 0;
// current total so far for the first eda_idx samples
int eda_total = 0;
//int eda_vals[eda_avg_size];

inline void get_EDA_data(long* packet) {
  int vEDA = analogRead(pEDA);
//  eda_vals[eda_idx] = vEDA;
  eda_total += vEDA;
  eda_idx = (eda_idx+1) % eda_avg_size;
  // if we have collected enough samples to average
  if ((eda_idx % eda_avg_size) == 0) {
//    for (int i = 0; i < eda_avg_size; i++)
    float avg_vEDA = eda_total / eda_avg_size;
    float Rskin = convert_eda_adc_to_Rskin(avg_vEDA);
    packet[i_EDA] = *((long*)(&Rskin));
    eda_total = 0;
  } else {
    // if not ready to average yet, mark EDA as invalid
    packet[i_VALID] |= (1<<v_EDA);
  }
//  Serial.println(String(eda_idx)+" , "+String(eda_total));

  #if v
//    Serial.print("EDA "); Serial.println(vEDA);
    Serial.print("EDA "); Serial.println(packet[i_EDA]);
  #endif
}

inline float convert_eda_adc_to_Rskin(int sensorValue) {
  float Vout = (sensorValue * 3.3)/4095;

  // these are constants- should not change between iterations
  // values are from the PCB layout/schematic in Fascia Physio Board V0
  const int Rref = 820000; // reference resistor between - opamp and gnd
  // Rref might actually be 2M or 820K -- undocumented board build value
  const float Vref = 3.3 * 20./(20.+140.); // voltage divider output (virtual gnd)
  const float i = (float)Vref / (float)Rref;

  float Rskin = (Vout - Vref) / i;
//  Serial.println(String(Vref)+", "+ String(Vout)+", "+String(Rskin));
  //float Cskin = 1./Rskin;

  return Rskin;
}

//inline void get_battery_v(long* packet) {
//    packet[i_BAT] = analogRead(pBAT);
//}

inline void get_PPG_temp_data(long* packet) {
  particleSensor.requestTemperature();
  long irValue = particleSensor.getIR(1); // ms to wait TODO figure out smallest good
                                          // TODO MATCH THIS WITH DATA SAMPLE RATE IN .SETUP()
  packet[i_PPG] = irValue;
//  Serial.println(irValue);
  if (irValue < 50000){
    #if v
      Serial.println("No contact with sensor "+String(irValue));
    #endif
    packet[i_VALID] |= (1<<v_PPG);
    packet[i_VALID] |= (1<<v_TEM); // TODO keep this here????
    if (irValue != 0) packet[i_VALID] |= (1<<v_TIM);
//    return;
  }

  float temperature = particleSensor.readTemperature();
  packet[i_TEM] = *(long*)(&temperature); //TODO make sure this casting works properly

  #if v
    Serial.print("TMP ");Serial.println(temperature);
  //  Serial.print("TMP ");Serial.println(packet[i_TEM]);

    Serial.print("PPG ");Serial.println(irValue);
  //  Serial.print("PPG ");Serial.println(packet[i_PPG]);
  #endif
}

/////////////////////////////////// WIFI STUFF //////////////////////////////////////
//int cnt = 0;

//There are two buffer used for wifi data sending.
char sendBuf[2][PACKET_SIZE*SEND_SIZE];

//Indicating which buffer is being written
int wBufIndex = 0;

//Indicates how much data packets are written into the current buffer
int wCount = 0;
bool isBufReady = false;

//For WIFI
int status = WL_IDLE_STATUS;
char ssid[] = SECRET_SSID;
char pass[] = SECRET_PASS;
unsigned int localPort = PORT_NUM;      // local port to listen on

WiFiUDP Udp;



inline char* getWriteBuf()
{
    return sendBuf[wBufIndex];
}

//Get the buffer for sending
//Have to be in non-interrupt context
inline char* getSendBuf()
{
    if(isBufReady == true)
    {
        isBufReady = false;
        return wBufIndex == 0 ? sendBuf[1] : sendBuf[0];
    }
    else
    {
        return 0;
    }
}

//Write one data packet into the buffer.
inline void pushToBuf(char* packet)
{
    ((int*)packet)[0] = cnt;
    ((int*)packet)[i_TIM] = millis();
//    Serial.println(((int*)packet)[i_TIM]);
    cnt++;
    //When current buffer is full
    if(wCount == SEND_SIZE)
    {
        //Switch buffer
        wBufIndex = (wBufIndex +1)%2;
        isBufReady = true;
        wCount = 0;
        //Serial.println("Switch Buffer");
    }
    //Write to buffer
    char* buf = getWriteBuf();
    memcpy(buf+wCount*PACKET_SIZE, packet, PACKET_SIZE);
    wCount++;

}

void printWiFiStatus() {
  // print the SSID of the network you're attached to:
  Serial.print("SSID: ");
  Serial.println(WiFi.SSID());

  // print your WiFi shield's IP address:
  IPAddress ip = WiFi.localIP();
  Serial.print("IP Address: ");
  Serial.println(ip);

  Serial.print("Data host IP: ");
  Serial.println(HOST_ID);

  // print the received signal strength:
  long rssi = WiFi.RSSI();
  Serial.print("signal strength (RSSI):");
  Serial.print(rssi);
  Serial.println(" dBm");
}

void setupWifi()
{
    //Setup the wifi
    // check for the WiFi module:
  if (WiFi.status() == WL_NO_MODULE) {
    Serial.println("Communication with WiFi module failed!");
    // don't continue
    while (true);
  }

  String fv = WiFi.firmwareVersion();
  if (fv < "1.0.0") {
    Serial.println("Please upgrade the firmware");
  }

  // attempt to connect to Wifi network:
  while (status != WL_CONNECTED) {
    Serial.print("Attempting to connect to SSID: ");
    Serial.println(ssid);
    // Connect to WPA/WPA2 network. Change this line if using open or WEP network:
    status = WiFi.begin(ssid, pass);

    // wait 10 seconds for connection:
    delay(5000);
  }
  Serial.println("Connected to wifi");
  printWiFiStatus();

  Serial.println("\nStarting connection to server...");
  // if you get a connection, report back via serial:
  Udp.begin(localPort);

}



inline void sendWiFiDataPacket() {
  byte* sBuf = (byte*)getSendBuf();

  if(sBuf != 0){
    Udp.beginPacket(HOST_ID, PORT_NUM);
    int nbytes = Udp.write(sBuf, PACKET_SIZE*SEND_SIZE);
    Udp.endPacket();
    if (nbytes != PACKET_SIZE*SEND_SIZE) {
      Serial.println("Failed to send packet. Sent "+String(nbytes)+" bytes only");
    } else {
      #if v
        Serial.println("Successfully sent full packet of "+String(nbytes)+" bytes");
      #endif
    }
  }
  LEDval = (cnt%10)? LEDval : !LEDval;
//  Serial.println("cnt = "+String(cnt)+" LEDVAL = "+String(LEDval));
  digitalWrite(pLED, LEDval);
}
```

APPENDIX C: PYTHON VISUALIZATION CODE

https://github.mit.edu/gbernal/Fascia_nucleus/tree/master/Fascia_dataViz/Fascia_sensor_data_plotter

CausalButter.py

```python
# "CausalButter.py" is the python class I wrote to implement the causal butterworth filter (same algorithm as Tianhe_filter but in python)

# reference here: http://www.exstrom.com/journal/sigproc/
#                 http://www.exstrom.com/journal/sigproc/bwbpf.c
import math

class CausalButter:
    # the default init method assumes Causal butter is a bandpass filter, and allows signal frequency from f_low to f_high to pass.
    # when bandstop == 1, the causal butter filter becomes bandstop and signal frequency from f_low to f_high will be filtered out
    def __init__(self,order,f_low,f_high,sampleRate,bandstop=0):
        if order%4 != 0:
            print ("the order of CausalButter has to be a multiple of 4")
            return
        self.f_high = f_high
        self.f_low = f_low
        self.sampleRate = sampleRate
        self.order = order
        self.bandstop = bandstop

        s = self.sampleRate
        a = math.cos(math.pi*(f_high+f_low)/s)/math.cos(math.pi*(f_high-f_low)/s)
        a2 = a**2
        b = math.tan(math.pi*(f_high-f_low)/s);
        b2 = b**2
        n = int(order/4)
        self.n = int(order/4)
        self.a = a
        self.a2 = a2
        self.b = b
        self.b2 = b2

        self.A = [0]*n
        self.d1 = [0]*n
        self.d2 = [0]*n
        self.d3 = [0]*n
        self.d4 = [0]*n
        self.w0 = [0]*n
        self.w1 = [0]*n
        self.w2 = [0]*n
        self.w3 = [0]*n
        self.w4 = [0]*n

        if self.bandstop ==0:
            for i in range(n):
                r = math.sin(math.pi*(2.0*i+1.0)/(4.0*n));
                s = b2 + 2.0*b*r + 1.0;
                self.A[i] = b2/s;
                self.d1[i] = 4.0*a*(1.0+b*r)/s;
                self.d2[i] = 2.0*(b2-2.0*a2-1.0)/s;
                self.d3[i] = 4.0*a*(1.0-b*r)/s;
                self.d4[i] = -(b2 - 2.0*b*r + 1.0)/s;
        else:
            for i in range(n):
                r = math.sin(math.pi*(2.0*i+1.0)/(4.0*n));
                s = b2 + 2.0*b*r + 1.0;
                self.A[i] = 1/s;
                self.d1[i] = 4.0*a*(1.0+b*r)/s;
                self.d2[i] = 2.0*(b2-2.0*a2-1.0)/s;
                self.d3[i] = 4.0*a*(1.0-b*r)/s;
                self.d4[i] = -(b2 - 2.0*b*r + 1.0)/s;
            self.r = 4.0*a;
            self.s = 4.0*a2+2.0;

    def inputData(self, raw_data):
        #BUGMAN 5/24/2017 modified the npts
        npts = len(raw_data);
        filtered_data = [None]*npts;

        # the default is to create a bandpass causal butter filter
        if self.bandstop ==0:
            for pnt in range(npts):
                x = raw_data[pnt]
                for i in range(self.n):
                    self.w0[i] = self.d1[i]*self.w1[i] + self.d2[i]*self.w2[i] + self.d3[i]*self.w3[i] + self.d4[i]*self.w4[i] + x;
                    x = self.A[i]*(self.w0[i] - 2.0*self.w2[i] + self.w4[i]);
                    self.w4[i] = self.w3[i];
                    self.w3[i] = self.w2[i];
                    self.w2[i] = self.w1[i];
                    self.w1[i] = self.w0[i];
                filtered_data[pnt] = x
        else:
            for pnt in range(npts):
                x = raw_data[pnt]
                for i in range(self.n):
                    self.w0[i] = self.d1[i]*self.w1[i] + self.d2[i]*self.w2[i] + self.d3[i]*self.w3[i] + self.d4[i]*self.w4[i] + x;
                    # bandstop method changed some coefficients here
                    x = self.A[i]*(self.w0[i] - self.r*self.w1[i] + self.s*self.w2[i] - self.r*self.w3[i]+ self.w4[i]);
                    self.w4[i] = self.w3[i];
                    self.w3[i] = self.w2[i];
                    self.w2[i] = self.w1[i];
                    self.w1[i] = self.w0[i];
                filtered_data[pnt] = x


        return filtered_data


    def printParams(self):
        print('self.f_high is ', self.f_high)
        print('self.f_low is ', self.f_low)
        print('self.sampleRate is ', self.sampleRate)
        print('self.order is ', self.order)
        print('self.n is ', self.n)
        print('self.a is ', self.a)
        print('self.a2 is ', self.a2)
        print('self.b is ', self.b)
        print('self.b2 is ', self.b2)
        print('self.A is ', self.A)
        print('self.d1 is ', self.d1)
        print('self.d2 is ', self.d2)
        print('self.d3 is ', self.d3)
        print('self.d4 is ', self.d4)
        print('self.w0 is ', self.w0)
        print('self.w1 is ', self.w1)
        print('self.w2 is ', self.w2)
        print('self.w3 is ', self.w3)
        print('self.w4 is ', self.w4)
```

floatingCurves.py

```
"""
This class is a widget it allows displaying multiple graphs and float each as a window if needed on real time.
Updating the graph will be done outside of this class.
"""

import sys
from PyQt5 import QtGui, QtCore, QtWidgets
import numpy as np
import pyqtgraph as pg



class floatingCurves_Max(QtWidgets.QMainWindow):
    def __init__(self, curve:pg.PlotDataItem, oldWidget:pg.PlotWidget ,parent=None):
        super(floatingCurves_Max, self).__init__(parent)
        self.oldWidget = oldWidget
        self.curve = curve
        plotWidget = pg.PlotWidget()
        plotWidget.addItem(curve)
        centralWidget = QtWidgets.QWidget(self)
        self.setCentralWidget(plotWidget)

    def closeEvent(self, a0):
        self.oldWidget.addItem(self.curve)

        return super().closeEvent(a0)


class floatingCurves(QtWidgets.QWidget):

    def __init__(self, channelNum, start_i, fft_chan):
        super(floatingCurves, self).__init__()
        self.curveList = list()
        self.plotWidgetList = list()

        #Perpare the layout
        self.layout = QtWidgets.QGridLayout()

        self.setLayout(self.layout)
        self.titles = ["packet number", "Valid array", "ADS 1: EMG 1/2 (volts)", "ADS 2: EMG 4/3 (volts)", "ADS 3: EOG 1/2 (volts)",
                       "ADS 4: EMG 7/8 (volts)", "ADS 5: EEG1 (PASSIVE) (volts)", "ADS 6: EEG2 (PASSIVE) (volts)", "ADS 7: EEG (ACTIVE) (volts)", "ADS 8:
EEG (ACTIVE) (volts)",
                       "IMU Ax", "IMU Ay","IMU Az", "IMU Gx","IMU Gy", "IMU Gz",#"IMU 7", "IMU 8","IMU 9",
                       "EDA: Rskin (Ohm)","temperature (C)", "PPG raw data", "FFT from ADS "+str(fft_chan)]#,"battery voltage level"]#"heart rate
arduino"]
        self.generateGraphsArray(channelNum, start_i)
        self.addText()

    def addText(self):
        PN = QtWidgets.QLabel()
        PN.setText("Packet #: ")
        self.layout.addWidget(PN)
        self.PN = PN
        PDR = QtWidgets.QLabel()
        PDR.setText("P Data Rate: ")
        self.layout.addWidget(PDR) #pg.TextItem("Data Rate: ")
        self.PDR = PDR
        ADR = QtWidgets.QLabel()
        ADR.setText("A Data Rate: ")
        self.layout.addWidget(ADR) #pg.TextItem("Data Rate: ")
        self.ADR = ADR
        HR = QtWidgets.QLabel()
        HR.setText("Heart Rate: ")
        self.layout.addWidget(HR)
        self.HR = HR

    def addCurve(self, newCurve:pg.PlotDataItem, x, y, t):
        self.curveList.append(newCurve)
        plotWidget = pg.PlotWidget(title=t)
        plotWidget.addItem(newCurve)
        self.plotWidgetList.append(plotWidget)
        self.layout.addWidget(plotWidget,x,y)
        #Add the button
        button = QtWidgets.QPushButton("+",plotWidget)
        button.resize(20,20)
        button.clicked.connect(self.make_btn_floatWnd(len(self.curveList)-1))

    def generateGraphsArray(self, channelNum, start_i):
        for i in range(channelNum):
            newCurve = pg.PlotDataItem()
            y = int(i/4)
            x = i%4
            self.addCurve(newCurve ,x,y, self.titles[i+start_i])

    def make_btn_floatWnd(self, index):
        def btn_floatWnd():
            newWnd = floatingCurves_Max(self.curveList[index], self.plotWidgetList[index], self)
            newWnd.show()
        return btn_floatWnd

    def updateCurve(self, index, data: list(), data_x: list() = []):
        if data_x != []:
            self.curveList[index].setData(x=data_x, y=data)
        else:
            self.curveList[index].setData(y=data)

#Used for testing

# win = pg.GraphicsWindow()
# win.addPlot()
```

BCI_Data_Receiver.py

```
import socket
import sys
import time
import numpy as np
import threading
import struct

import time

class BCI_Data_Receiver(object):

    def __init__(self, ip, port, data_plotting_widget):
        self.ip = ip
        self.port = port
        self.sock = None
        self.receiveBuff= bytes()
        self.dataBuff = []
        self.dataReadyCallback = None
        self.readingThread = None
        self.address = (self.ip, self.port)
        self.prev_time_stamp = 0
        self.prev_EDA_time_stamp = 0
        # self.prev_PPG_time_stamp = 0
        self.dataPlottingWidget = data_plotting_widget
        self.current_data_rate = 1

        self.prev_A_ts = 0;

    def startConnection(self):
        """Start the socket connection"""
        # self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        #For UDP
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.bind(self.address)
        print(self.address)
        # self.sock.connect(("192.168.1.10",35294))
        self.sock.settimeout(30.0)
        self.processStream()

    def asyncReceiveData(self, dataReadyCallback):
        """Receive the data from ADS1299 asynchronously."""
        if self.readingThread == None:
            self.dataReadyCallback = dataReadyCallback
            self.readingThread = threading.Thread(target = self.startConnection)
            self.readingThread.start()
        else:
            raise Exception("The reading thread is already running!")

    def processStream(self):
        data_names = ["PKT #", "VALID", "ADS 1", "ADS 2", "ADS 3","ADS 4", "ADS 5", "ADS 6", "ADS 7", "ADS 8",
                      "IMU 0", "IMU 1", "IMU 2", "IMU 3", "IMU 4", "IMU 5", #"IMU 7", "IMU 8", "IMU 9",
                      "EDA  ", "TEMP ", "PPG  ", "TIM"]#, "BTR "]#"HRT  "]
        num_elements = 17
        num_bytes = 4*num_elements
        num_packets = 22
        while True:
            #Receive data from sensor
            data, addr = self.sock.recvfrom(num_bytes*num_packets)
            cur_time_stamp = time.time()
            # print("data rate: "+str(int(num_packets/(cur_time_stamp-self.prev_time_stamp))) + " Hz")
            self.current_data_rate = int(num_packets/(cur_time_stamp-self.prev_time_stamp))
            t = "P Data Rate: "+str(self.current_data_rate)+" Hz"
            self.dataPlottingWidget.PDR.setText(t)
            self.prev_time_stamp = time.time()
            # print(data, addr)
            #self.receiveBuff = self.receiveBuff + self.sock.recv(40)
            self.receiveBuff = self.receiveBuff + data

            if(len(self.receiveBuff) >= num_bytes*num_packets):
                data = self.receiveBuff[0:num_bytes*num_packets]
                self.receiveBuff = self.receiveBuff[num_bytes*num_packets:]
                for i in range(num_packets):
                    unpacked_data = struct.unpack('i'+'i'+'f'*8+'h'*6+'f'+'f'+'f'+'ii', data[i*num_bytes: (i+1)*num_bytes])
                    # unpacked_data = struct.unpack('i'*num_elements, data[i*num_bytes: (i+1)*num_bytes])
                    #from manual For the 'f', 'd' and 'e' conversion codes, the packed representation uses the IEEE 754 binary32, binary64 or binary16
format (for 'f', 'd' or 'e' respectively), regardless of the floating-point format used by the platform.

                    # print(unpacked_data[19], self.prev_A_ts)
                    if unpacked_data[19] != self.prev_A_ts:
                        dr = 1000/(unpacked_data[19] - self.prev_A_ts)
                        # print(dr)
                        t = "A Data Rate: " + str(int(dr)) + " Hz"
                        self.dataPlottingWidget.ADR.setText(t)
                        self.prev_A_ts = unpacked_data[19] # milliseconds

                    # For Walaa: debug prints

                    # for j in range(num_elements):
                    #     print(data_names[j] + ' ' + str(unpacked_data[j]))

                    # if (unpacked_data[1] & (1<<16)) == 0:
                    #     # EDA is valid
                    #     print("EDA rate: " + str(int(1 / (cur_time_stamp - self.prev_EDA_time_stamp))) + " Hz")
                    #     self.prev_EDA_time_stamp = self.prev_time_stamp

                    self.dataReadyCallback(unpacked_data)
```

MainGUI.py

```
import os
os.environ['PYQTGRAPH_QT_LIB'] = 'PyQt5'
from PyQt5 import QtGui, QtCore, QtWidgets
import pyqtgraph as pg
import multiprocessing
import pandas as pd

from BCI_Data_Receiver  import *
import floatingCurves as fc
from CausalButter import *

import heartpy as hp
import math
import threading



i_CNT = 0
i_VAL = i_CNT + 1
i_ADS = i_VAL + 1
i_IMU = i_ADS + 8
i_EDA = i_IMU + 6
i_TEM = i_EDA + 1
i_PPG = i_TEM + 1
i_TIM = i_PPG + 1

class mainWindow(QtWidgets.QWidget):

    def __init__(self):

        #Init Data structures
        super(mainWindow,self).__init__()
        self.plotBufs = list()
        PLOTWNDSIZE = 2000
        self.PLOTWNDSIZE = PLOTWNDSIZE

        # for plotting
        self.start_idx = 2 #TODO: make this 0 if you want to graph all the packet data
        self.n_plots = 19
        self.fft_idx = self.n_plots-self.start_idx

        # for FFT
        self.graph_fft = 1 #TODO: change this to 1 if you dont want FFT graph
        self.FFT_CHANNEL = 4 #TODO: make sure this is the channel you want the FFT for (0 indexed)
        self.fft_lock = threading.Lock();
        # self.fft_thread = threading.Thread(target = self.fft_calc, args = (self.FFT_CHANNEL,) );
        self.moreData = False;
        # self.fft_thread.start()

        # for initial plotting
        for i in range(self.n_plots-self.start_idx + self.graph_fft):
            self.plotBufs.append(np.zeros(PLOTWNDSIZE))

        # for data Recording
        self.isRecording = False
        self.recordingBuf = list()

        # for filters
        data_rate = 1000 #TODO: make sure this matches the data rate of the ADS1299 in the firmware
        self.data_rate = data_rate
        num_ADS_plots = 8
        # Init/store all the required filters
        # Do the bandpass filters
        self.BPfilters = []
        # for i in range(0,num_ADS_plots):
        self.BPfilters.append(CausalButter(4, 10, 500, data_rate, 0))  # EMG 1/2
        self.BPfilters.append(CausalButter(4, 10, 500, data_rate, 0))  # EMG 4/3
        self.BPfilters.append(CausalButter(4, 10, 500, data_rate, 0))  # EOG 1/2
        self.BPfilters.append(CausalButter(4, 10, 500, data_rate, 0))  # EMG 5/6
        self.BPfilters.append(CausalButter(4, 10, 500, data_rate, 0))  # EMG 7/8
        self.BPfilters.append(CausalButter(8,  5,  50, data_rate, 0))  # EEG 1
        self.BPfilters.append(CausalButter(8,  5,  50, data_rate, 0))  # EEG 2
        self.BPfilters.append(CausalButter(8,  5,  50, data_rate, 0))  # EEG 3
        self.BPfilters.append(CausalButter(8,  5,  50, data_rate, 0))  # EEG 4
        # and  the bandstop filters
        self.BSfilters = []
        for i in range(0,num_ADS_plots):
            self.BSfilters.append(CausalButter(8, 55, 65, data_rate, 1))

        # for heart rate measuring algorithm
        self.heart_sig_arr  = []
        self.heartbeat_ts   = []
        self.heartrate_avg  = []

        # initialize the UI
        self.title = "Fascia Sensor Data"
        self.initUI()

        self.timer = QtCore.QTimer()
        self.timer.timeout.connect(self.updateGUI)

        # The ip of user's machine
        self.ip = '10.0.0.74' #TODO make sure this matches intet in en0 in ifconfig
        self.port_number = 8899

        self.Data_receiver = BCI_Data_Receiver(self.ip, self.port_number, self.dataPlottingWidget)
        self.Data_receiver.asyncReceiveData(self.dataReadyCallback)



    def initUI(self):
        self.setWindowTitle(self.title)
        self.setGeometry(100,100,1024,768)

        hbox = QtWidgets.QHBoxLayout()
        self.setLayout(hbox)



        #Add the graph arrays

        #Perpare the array
        self.dataPlottingWidget = fc.floatingCurves(self.n_plots-self.start_idx+self.graph_fft, self.start_idx, self.FFT_CHANNEL)

        hbox.addWidget(self.dataPlottingWidget)

        # #Add the button panel
        # self.button_panel = QtWidgets.QWidget(self)
        # self.button_panel_layout = QtWidgets.QVBoxLayout()
        # self.button_panel.setLayout(self.button_panel_layout)
        # hbox.addWidget(self.button_panel)
        # #Add all the buttons
        # self.ICA_btn = QtWidgets.QPushButton("ICA")
        # self.record_btn = QtWidgets.QPushButton("Record")
        # self.filter_btn = QtWidgets.QPushButton("Filters")
        #
        # self.button_panel_layout.addWidget(self.ICA_btn)
        # self.button_panel_layout.addWidget(self.record_btn)
        # self.button_panel_layout.addWidget(self.filter_btn)
        #
        # #Connect all button functions
        # self.linkBtnFunctions()

        self.show()

    def linkBtnFunctions(self):
        self.record_btn.clicked.connect(self.onRecordBtnClicked)



    def dataReadyCallback(self, newData):
        d = list()
```

```
temp = np.zeros(len(newData))
invalid_arr = newData[1];
# t = "Packet #: " + str(newData[i_CNT])
# self.dataPlottingWidget.PN.setText(t)

# # for recording purposes
# for i in range(0, self.start_idx):
#     temp[i] = newData[i]

if self.graph_fft:
    self.fft_thread = threading.Thread(target = self.fft_calc, args = (self.FFT_CHANNEL,) );
    self.fft_thread.start()



for i in range(self.start_idx, self.n_plots):
    #apply filters to newData
    # temp[2+i] = newData[2+i]
    # temp[2+i] = self.BPfilters[i].inputData([newData[2+i]])[0]
    # temp[2+i] = self.BSfilters[i].inputData([temp[2+i]])[0]
    if (i >1 and i<10):
        temp[i] = self.BPfilters[i-i_ADS].inputData([newData[i]])[0]
        temp[i] = self.BSfilters[i-i_ADS].inputData([temp[i]])[0]
    else:
        temp[i] = newData[i]

    # temp[2+i] = self.HPfilters[i].inputData([convert_ADC_volts(newData[2+i])])[0]
    # temp[2+i] = convert_ADC_volts(newData[2+i]);
    # print(i)
    d.append([temp[i]])

    if ((invalid_arr>>i) & 1):
        # print("invalid data at "+str(i))
        continue

    #For plotting
    idx = i - self.start_idx
    self.plotBufs[idx] = self.plotBufs[idx][1:]
    self.plotBufs[idx] = np.append(self.plotBufs[idx],d[idx][0])

# calculate heart rate
# me: if it drops 400 counts in 5 samples -> heart beat
# if not ((invalid_arr >> 18) & 1):
#     try:
#         working_data, measures = hp.process(self.plotBufs[18], self.data_rate/10);
#         heart_rate = measures['bpm']
#         if not math.isnan(heart_rate):
#             print("heart rate: ", int(heart_rate)," bpm")
#         else:
#             print("hp nan")
#     except hp.exceptions.BadSignalWarning:
#         print("hp exception")
#         pass

# below would probably be good if the data is invalid due to lack of connection or incorrect data, instead of
# the sampling rate issue where i am only sampling the PPG data once every 10 packets.
# if ((invalid_arr >> i_PPG) & 1):
#     self.heart_sig_arr = []
#     self.heartbeat_ts = []
# else:
if not ((invalid_arr >> i_PPG) & 1):
    ppg_sig = newData[i_PPG]
    l = len(self.heart_sig_arr);
    for i in range(min(10, l)):
        if self.heart_sig_arr[l-1-i] - ppg_sig >= 100 and self.heart_sig_arr[l-1-i] - ppg_sig < 700 and l>20:
            print("heart beat!",newData[i_TIM])
            self.heartbeat_ts.append(newData[i_TIM])
            # self.plotBufs[i_PPG - self.start_idx][-1] *=-1 #mark spot in graph where heartbeat detected
            # calculate heart rate
            if len(self.heartbeat_ts) > 1:
                # delta_ts = time in ms difference between the current and most recent heart beat
                delta_ts = self.heartbeat_ts[-1] - self.heartbeat_ts[len(self.heartbeat_ts)-2]
                delta_sec = delta_ts / 1000
                bpm = 1/(delta_sec/60)
                print("local heart rate: "+str(int(bpm)))
                self.heartrate_avg.append(bpm)
                bpm = np.average(self.heartrate_avg)
                print("heart rate:",int(bpm), "bpm, ",len(self.heartbeat_ts)," / ",delta_ts)
                t = "Heart Rate: " + str(int(bpm)) + " BPM"
                self.dataPlottingWidget.HR.setText(t)
            self.heart_sig_arr = []
            break
    self.heart_sig_arr.append(ppg_sig)
    # trim arrays to max lengths
    if len(self.heart_sig_arr) > 25:
        self.heart_sig_arr = self.heart_sig_arr[1:]
    if len(self.heartbeat_ts) > 100:
        self.heartbeat_ts = self.heartbeat_ts[1:]
    if len(self.heartrate_avg) > 100:
        self.heartrate_avg = self.heartrate_avg[1:]
elif ((invalid_arr >> i_TIM) & 1):
    print("no contact with ppg sensor")
    self.heart_sig_arr = []
    self.heartbeat_ts = []
    self.heartrate_avg = []
# else:
#     print("invalid heart and temp data")
# if(self.isRecording == True):
#     #save new data to the recordingBuf
#     self.recordingBuf.append(temp)
```

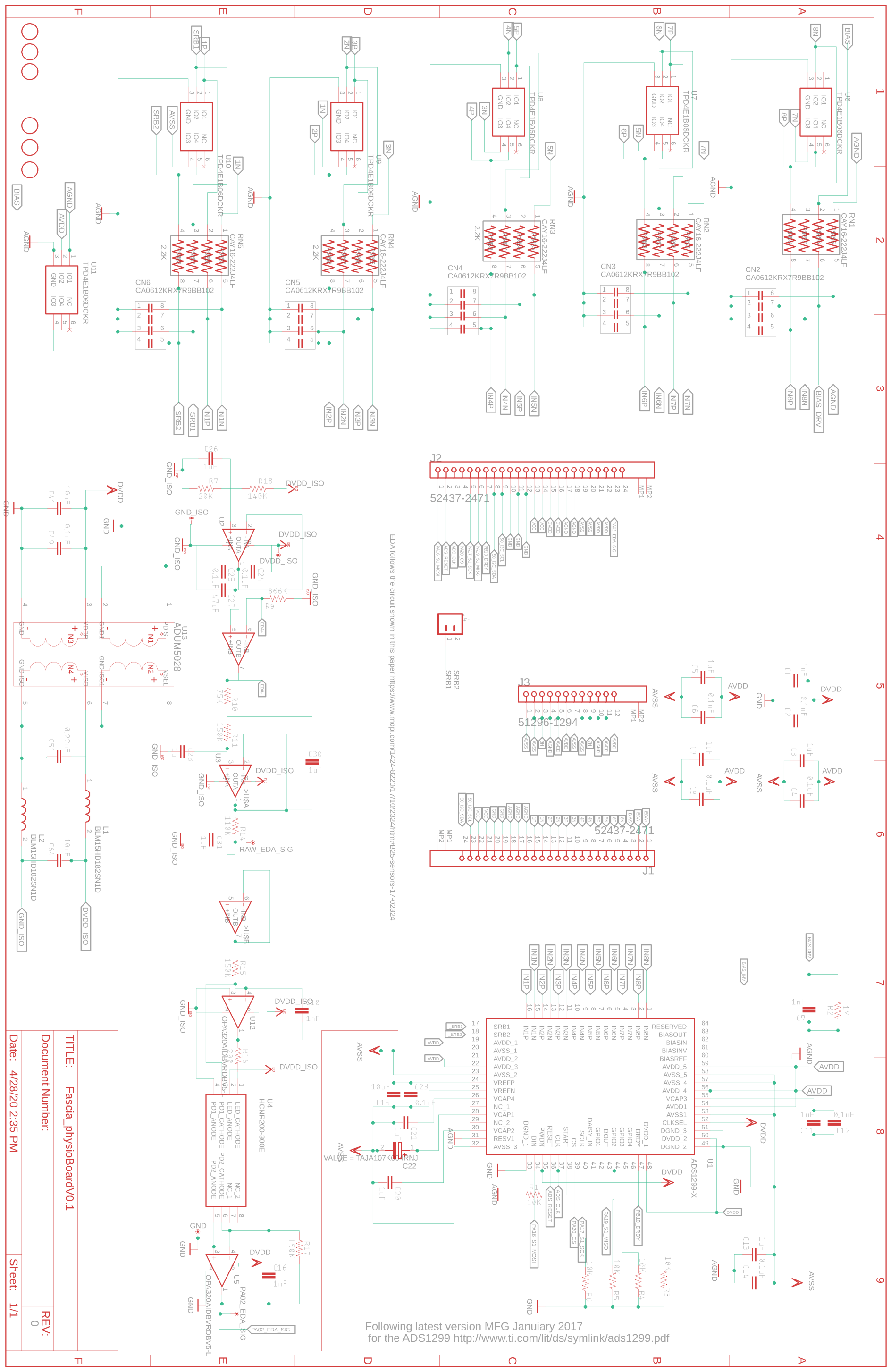
```
            self.moreData = True


        def fft_calc(self, channel):
            # while True:
            #     if self.moreData:
            #         self.moreData = False;
            #FFT stuff here
            # FFT_CHANNEL = 1;
            # self.fft_lock.acquire()
            channel_idx = i_ADS + channel - self.start_idx
            # print(channel_idx)
            bins = np.fft.rfft(self.plotBufs[channel_idx])
            size = len(self.plotBufs[channel_idx])
            # self.fft_lock.release()
            bins = np.abs(bins)#[np.abs(v) for v in bins]
            bins[0] = 0 # first element is DC element
            timestep = 1/self.data_rate
            freq = np.fft.rfftfreq(self.PLOTWNDSIZE, timestep);
            self.plotBufs[self.fft_idx] = bins,freq;
            # print("fft freq", freq)
            # print("fft bins", bins)
            # sys.stdout.flush()
            # print(len(bins), len(freq))

        def start(self):
            self.timer.start(1)

        def updateGUI(self):
            if self.moreData:
                self.moreData = False
                for i in range(self.n_plots-self.start_idx):
                    # print(i,type(self.plotBufs[i]),self.plotBufs[i])
                    self.dataPlottingWidget.updateCurve(i,self.plotBufs[i])
                if self.graph_fft:
                    self.dataPlottingWidget.updateCurve(self.fft_idx,self.plotBufs[self.fft_idx][0],self.plotBufs[self.fft_idx][1])

        def keyPressEvent(self, e):
            """
            press "r" key to record the data
            """
            if e.isAutoRepeat():
                return
            if( e.key() == QtCore.Qt.Key_R and self.isRecording == False):
                self.onRecordBtnClicked()
            return super().keyPressEvent(e)

        def keyReleaseEvent(self, e):
            """
            release "r" key to stop recording data
            """
            if e.isAutoRepeat():
                return
            if( e.key() == QtCore.Qt.Key_R and self.isRecording == True):
                self.onRecordBtnClicked()

            return super().keyReleaseEvent(e)

        def onRecordBtnClicked(self):
            """
            The callback function for the click event of record button
            """
            if(self.isRecording == False):
                self.isRecording = True
                #Start Record
                self.record_btn.setText("stop")
                #Push data to the buffer
                #This step is done in the data ready callback

            else:
                self.record_btn.setText("Record")
                self.isRecording = False
                #Open the save file dialog
                name = QtWidgets.QFileDialog.getSaveFileName(self, "Save File", "0.csv", "CSV(*.csv)")
                if(name[0] != ""):
                    df = pd.DataFrame(self.recordingBuf)
                    df.to_csv(name[0], encoding='utf-8', sep="\t", index=False)



                #Clear the recording buffer
                self.recordingBuf.clear()




        def onICABtnClicked(self):
            """
            Callback function for the click event of the ICA button
            """
            pass

    def convert_ADC_volts(raw_data, gain = 1): #gain was = 24
        # LSB = (2 x VREF)/ Gain / (2^24 - 1)
        vref = 4.5
        fs = 2*vref / gain
        converted_data = fs * raw_data / (2**(24)-1)
        # print(raw_data, " -> ", converted_data)
        return converted_data
```

```python
if __name__ == "__main__":
    # fs = 4.5/24
    # print(convert_ADC_volts(0x000000) == 0)
    # print(convert_ADC_volts(0x000001) == fs/(2**23 -1))  #+1
    # print(convert_ADC_volts(0x7FFFFF) >= fs) #MAX POS NUM
    # # print(convert_ADC_volts(0xFFFFFF) == -fs/(2**23 -1)) #-1
    # print(convert_ADC_volts(-1) == -fs/(2**23 -1)) #-1
    # # print(convert_ADC_volts(0x800000) <= -fs * 2**23/(2**23 -1)) #MAX NEG NUM
    # print(convert_ADC_volts(-8388608) <= -fs * 2**23/(2**23 -1)) #MAX NEG NUM
    app = QtWidgets.QApplication([])
    ex = mainWindow()
    ex.start()
    sys.exit(app.exec_())
```

APPENDIX D: CIRCUIT LAYOUT

Physiological Board

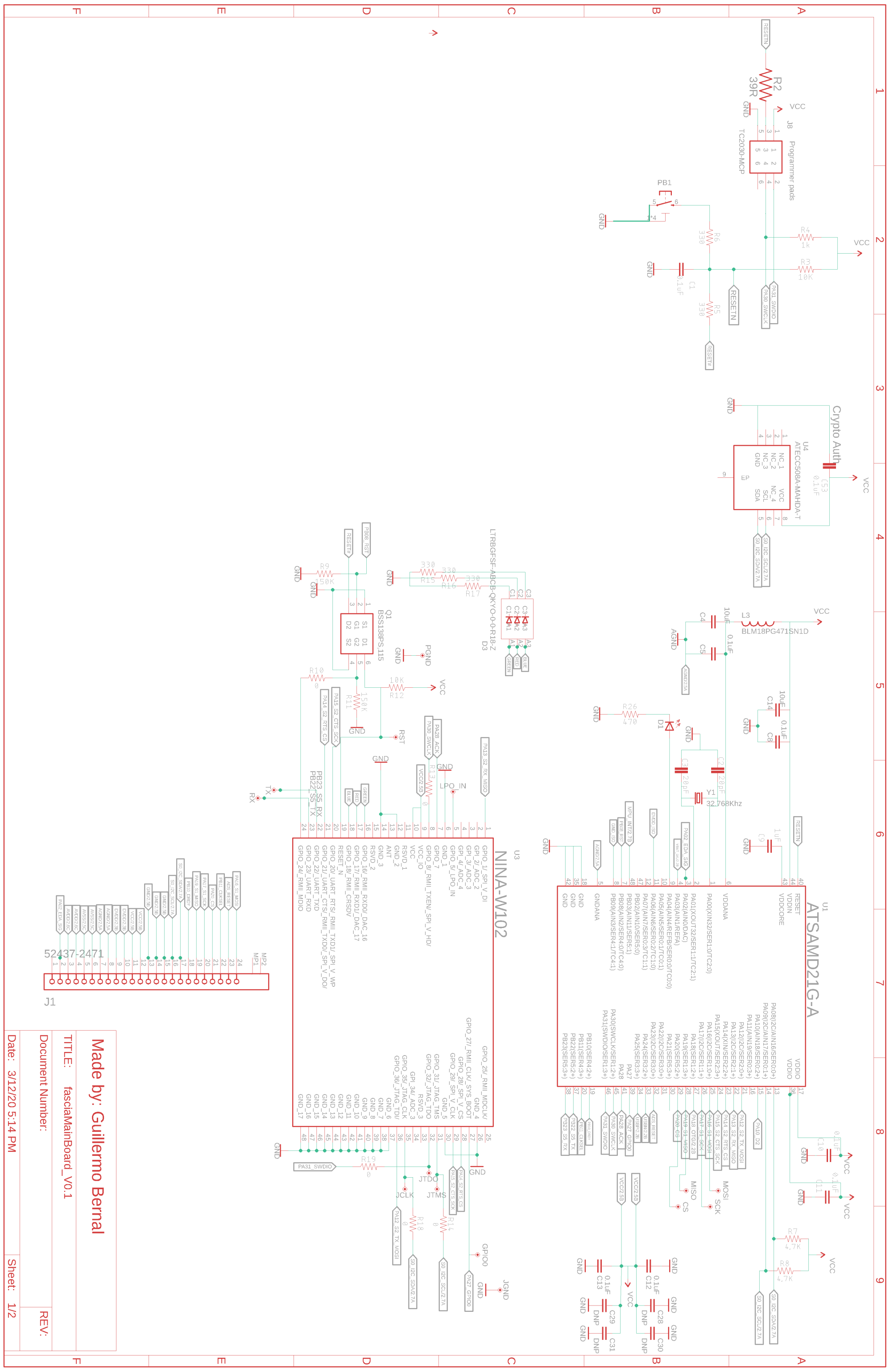
https://github.mit.edu/gbernal/Fascia_nucleus/tree/master/Fascia_physioBoard/fascia_physioBoardV0.1

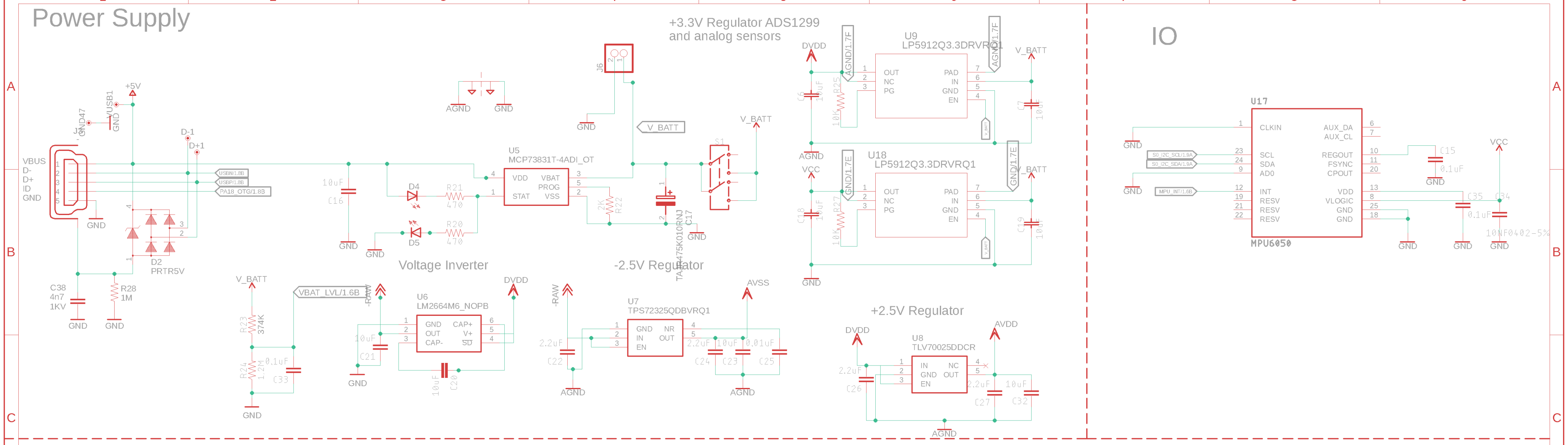


TITLE: Fascia_physioBoardV0.1
Document Number:
REV: 0
Date: 4/28/20 2:35 PM
Sheet: 1/1
ADS1299-X
ADUM5028
52437-2471
51296-1294
EDA follows the circuit shown in this paper: https://www.mdpi.com/1424-8220/17/10/2324/htm#B25-sensors-17-02324
Following latest version MFG January 2017 for the ADS1299 http://www.ti.com/lit/ds/symlink/ads1299.pdf

Main Board

https://github.mit.edu/gbernal/Fascia_nucleus/tree/master/Fascia_mainBoard/fasciaMainBoard_V0.1





Made by: Guillermo Bernal

TITLE: fasciaMainBoard_V0.1

Document Number:

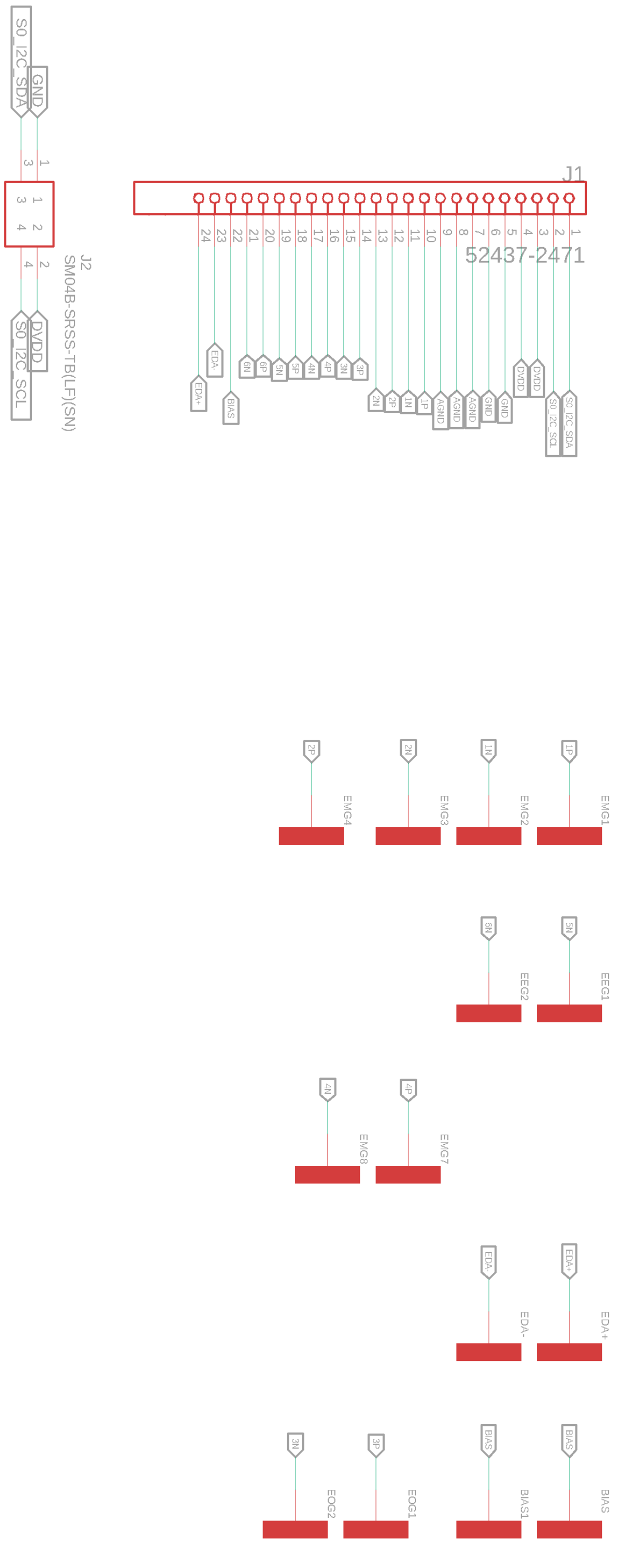
REV: 0

Date: 3/12/20 5:14 PM

Sheet: 2/2

Face Interface Board

```
https://github.mit.edu/gbernal/Fascia_nucleus/tree/master/Fascia_f
aceInterface/faceInterfaceV0.1
```



APPENDIX E: PCB DESIGN

Physiological Board

https://github.mit.edu/gbernal/Fascia_nucleus/tree/master/Fascia_physioBoard/fascia_physioBoardV0.1

Main Board

https://github.mit.edu/gbernal/Fascia_nucleus/tree/master/Fascia_mainBoard/fasciaMainBoard_V0.1

Face Interface Board

https://github.mit.edu/gbernal/Fascia_nucleus/tree/master/Fascia_faceInterface/faceInterfaceV0.1