

ABSTRACT

CHAUDHARI, KAUSTUBH ONKAR. Closed-Loop Non-Invasive Brain Stimulation and Recording Solution. (Under the direction of Dr. Flavio Frohlich and Dr. Alper Bozkurt).

Electroencephalography (EEG) is a monitoring method to record spontaneous electrical activity in the brain. Even though it has a low spatial resolution, it is used to diagnose brain disorders like sleep disorders, coma, encephalopathies and brain death and most often used to diagnose epilepsy which causes abnormalities in EEG readings. The hardware costs for EEG are significantly lower than those of most other techniques, so it provides a pretty good solution in high traffic hospitals and is required to diagnose disorders like epilepsy which are mostly present in developing countries. Many past and current studies have tried to perfect ways to detect, diagnose or monitor Epileptic EEG in patients. Although, there are many EEG devices available in the market, most of them are bulky or they are not built to record continuously for a long duration of time. So, for prolonged research, treatment and diagnosis it is necessary to develop new EEG devices which can be used to monitor the EEG from patients or users for a long time and which can be easily manufactured on a large scale for hospitals and areas where it is needed the most. This thesis tries to tackle this requirement by providing an EEG monitoring solution which fulfills these requirements while consuming minimal resources.

This thesis also contributes to the field of closed loop brain stimulation by modifying a transcranial current stimulation device to move on from the traditional direct current and alternating current stimulation to performing stimulation based on different fundamental parameters of the stimulated region by using a custom-made waveform with a trigger from an external system to complete the closed loop system.

© Copyright 2019 by Kaustubh Onkar Chaudhari

All Rights Reserved

Closed-Loop Non-Invasive Brain Stimulation and Recording Solution

by
Kaustubh Onkar Chaudhari

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina
2019

APPROVED BY:

Dr. Alper Bozkurt
Co-Chair of Advisory Committee

Dr. Flavio Frohlich
Co-Chair of Advisory Committee

Dr. Alexander Dean

BIOGRAPHY

Kaustubh Onkar Chaudhari received the B.E. degree in Electronics and communication Engineering from Shri Ramdeobaba College of Engineering and Management in 2016. In the fall of 2017, he joined the graduate program in Electrical and Computer Engineering department to pursue M.S. in Electrical and Computer Engineering specializing in VLSI design and embedded systems with a focus in medical device design. From the spring 2018 he began working towards developing and advancing devices in Neuro-tech as a research assistant in Dr. Flavio Frohlich's lab. He also worked as an FPGA design Intern developing part of a brain implant during his Masters at Wyss Center for Bio and Neuroengineering in Geneva, Switzerland.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. Flavio Frohlich for providing me the opportunity to work with him and the Frohlich Lab. He has provided me support and guidance as I progressed towards the completion of my Master's degree. I would also like to thank the Frohlich Lab members, especially Sangtae Ahn who I got to work with and get valuable advice from. I would like to thank the committee members, Dr. Alper Bozkurt and Dr. Alexander Dean for their time, valuable feedbacks and comments. I would like to give a special thanks to Dr. Alper Bozkurt for showing me the right way and for the support along the way without which this thesis wouldn't be possible.

TABLE OF CONTENTS

LIST OF TABLES	vi
LIST OF FIGURES	vii
Chapter 1 INTRODUCTION	1
1.1 Motivation.....	1
1.2 Objective	5
Chapter 2 EEG MONITORING DEVICE DEVELOPMENT	6
2.1 Device Components	6
2.1.1 EEG Sampling Frontend.....	8
2.1.2 ‘EASE’ Module	15
2.1.3 Docking Station	16
2.2 System Architectures	16
2.2.1 ‘EASE’ Module Architecture	20
2.2.2 Docking Station Architecture	24
2.3 Hardware Interfacing and Design	29
2.3.1 Interfacing and Design Choices	29
2.3.2 Circuit Description.....	32
2.4 Result	35
Chapter 3 NON-INVASIVE BRAIN STIMULATION DEVICE DEVELOPMENT ..	47
3.1 Device Components.....	47
3.2 Device Architecture	49
3.3 Result	67
REFERENCES	70
APPENDIX	76
Appendix A Programs Used	77
A.1 Program for ATMEGA328PB on the ‘EASE’ Module	77
A.1.1 ‘main.ino’	77
A.1.2 ‘gpio.h’	80
A.2 Programs for Raspberry Pi on the Docking Station.....	81
A.2.1 ‘main.py’	81
A.2.2 ‘./rtc/settime.py’	84
A.2.3 ‘./sdsremod/main.c’	86
A.2.4 ‘./sdsrmod/mmcbb.c’	89
A.3 Programs used for analysis of the Recorded Data	90
A.3.1 ‘multipleanalysis.m’	90
A.3.2 ‘analyze.m’	93

Appendix B PCB Layouts	95
B.1 ‘EASE’ Module layout	95
B.2 Docking Station header layout.....	97

LIST OF TABLES

Table 2.1	TGAM Specifications	9
Table 3.1	XCSITE 100 change analysis relevant configuration	68

LIST OF FIGURES

Figure 1.1	Changes in EEG signal during onset of Epileptic seizure	2
Figure 1.2	Alpha-Stim custom stimulation waveform	4
Figure 2.1	Block diagram of the EEG monitoring system	7
Figure 2.2	Think Gear ASIC Module Front and Back	8
Figure 2.3	TGAM frequency response for input 30sec, 2mV peak to peak signals	11
Figure 2.4	TGAM recorded signal in time, 30sec 2Hz	11
Figure 2.5	Records of the Berger rhythm made with pad electrodes on the head (vertex and occiput), with the subjects sitting with the eyes closed	12
Figure 2.6	Frequency response for 30 sec eyes closed occipital lobe data using TGAM.....	13
Figure 2.7	Recorded, low pass filtered signal at 50Hz for eyes closed occipital lobe data using TGAM	13
Figure 2.8	Frequency response of TGAM over 0.1Hz – 140Hz input.....	14
Figure 2.9	EEG System Architecture	17
Figure 2.10	EEG System Flowchart.....	19
Figure 2.11	EASE module flowchart	23
Figure 2.12	Docking Station flowchart	28
Figure 2.13	EASE module GPIO and support circuitry	32
Figure 2.14	EASE module peripherals, Real Time Clock, Level Converter and SD housing...	33
Figure 2.15	EASE module connector for headset and Docking Station	35
Figure 2.16	EASE module front view	37
Figure 2.17	EASE module back view	37
Figure 2.18	Docking Station with the 14-pin header	38
Figure 2.19	EEG wet, golden cup electrode connected to occipital lobe.....	38

Figure 2.20	Reference and ground electrodes connected to the ear with a clip	39
Figure 2.21	Modified NeuroSky's Mindwave headset to use the TGAM directly	39
Figure 2.22	EEG system's response with applied signal at 7Hz frequency at $\pm 1mVp$	40
Figure 2.23	EEG system's response with applied signal at different frequencies at $\pm 500uVp$.	41
Figure 2.24	EEG system's maximum error response for signals applied at $\pm 10uVp$	41
Figure 2.25	EEG system's maximum error response for signals applied at $\pm 50uVp$	42
Figure 2.26	EEG system's maximum error response for signals applied at $\pm 100uVp$	42
Figure 2.27	EEG system's response for unfiltered eyes closed occipital lobe data	43
Figure 2.28	EEG system's current consumption over time.....	44
Figure 3.1	XCSITE 100, by Pulvinar Neuro	48
Figure 3.2	XCSITE 100 System Architecture.....	51
Figure 3.3	XCSITE 100 software flow	53
Figure 3.4	XCSITE 100 main state machine.....	56
Figure 3.5	Applied custom waveform.....	67
Figure 3.6	Output custom waveform result.....	68
Figure 3.7	Output custom waveform with a different time-base	69
Figure B.1	EASE Module Front Layout	95
Figure B.2	EASE Module Back Layout.....	96
Figure B.3	Docking Station Header Front Layout	97
Figure B.4	Docking Station Header Back Layout	98

CHAPTER 1

INTRODUCTION

1.1. Motivation

Electroencephalography (EEG) is a monitoring method to record electrical activity in the brain. EEG is a record of spontaneous activity in the brain over a period of time, which cannot be localized to a few neurons [INTR1] [INTR2]. Even though it has a low spatial resolution and it cannot precisely measure neural activity, it is used to diagnose brain disorders like sleep disorders, coma, encephalopathies, brain death and most often used to diagnose epilepsy which causes abnormalities in EEG readings [INTR3]. Epilepsy is a common, deadly brain disorder; for example, as of 2015 about 39 million people had epilepsy [INTR5], and it resulted in around 125,000 deaths [INTR6]. The hardware costs for EEG are significantly lower than those of most other techniques [INTR4], because of which it provides a cost-effective solution in high traffic hospitals. EEGs are also having very high temporal resolution because of relatively higher sampling frequencies for noninvasive monitoring techniques [INTR7]. EEG is also relatively tolerant of movement, although it does cause artifacts, they can be minimized using various methods available. Because of various advantages EEG is an effective multifaceted, versatile solution to diagnose patients with brain disorders. Especially when considering 80% of the Epilepsy cases occur in developing countries [INTR8], a cost-effective method like EEG to monitor or diagnose a brain disorder like Epilepsy would prove very beneficial.

There are many studies that have been done and many that are being done for perfecting ways to either detect or diagnose or monitor Epileptic EEG in patients, for example [INT10]

[INT11] [INT12]. And there are new techniques emerging which are making use of artificial intelligence and machine learning for the same cause [INTR9] [INT13]. Fig 1.1 shows an Epilepsy seizure marker where we can see how the EEG suddenly changes as soon as there is an onset of seizure [INTR9].

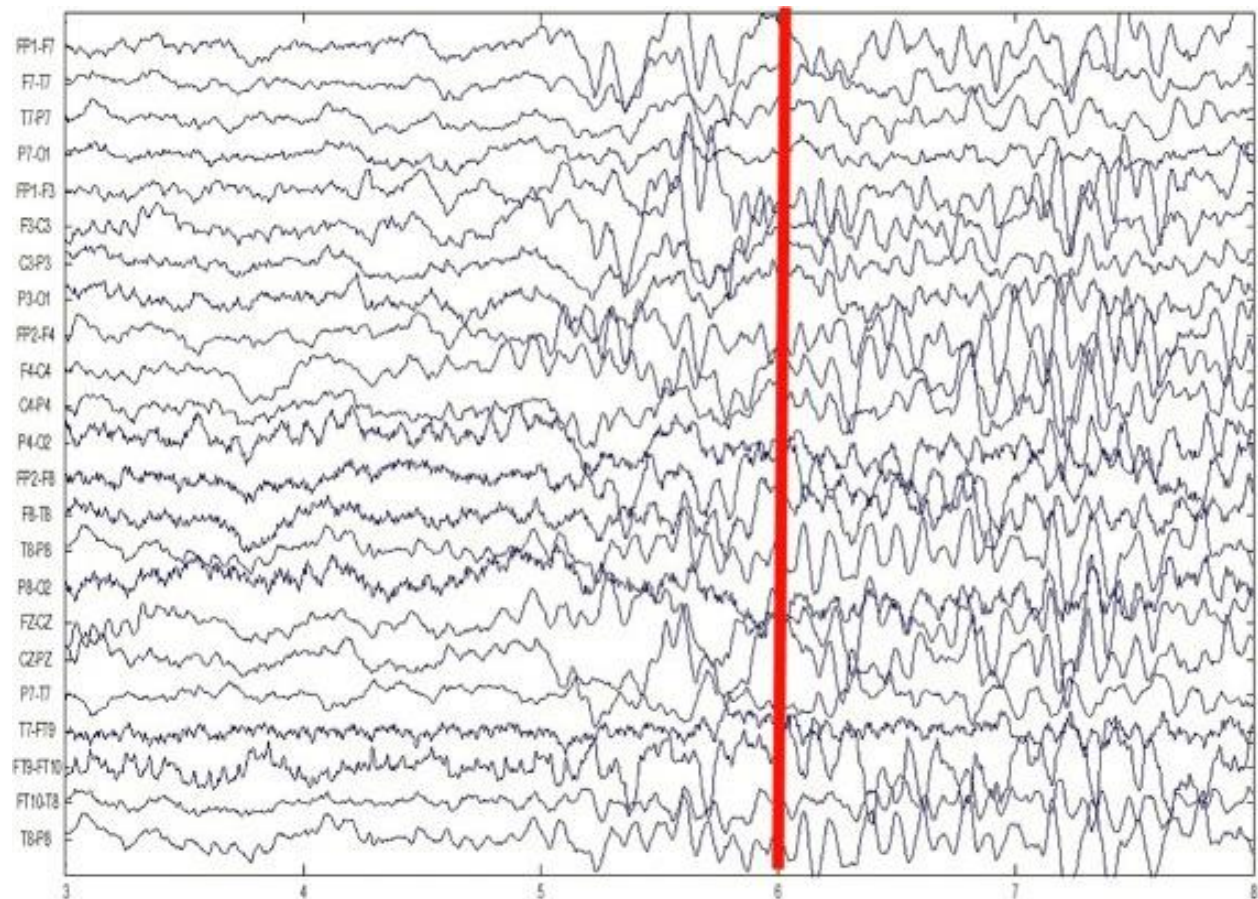


Figure 1.1 Changes in EEG signal during onset of Epileptic seizure [INTR9].

Similarly, while there are many EEG devices in the market, most of them are either bulky EEG devices or the devices which are not exactly built to record for a long duration of time continuously. For example, let's consider devices which could be used for conducting a big study which would mean the devices have to be comparatively cheaper and be able to provide research grade EEG data and record it at the same time. One example would be NeuroSky's Mindwave

[INT14]. It requires a single AAA battery, but runs for 8 hours at a time. Moreover, the module cannot store data locally, it can simply acquire the EEG data and transmit it via Bluetooth. The output signal is of research grade quality, but in order to record and analyze the data it requires mindset research tools which cost more than 10 times the device itself [INT15]. Another such device is EMOTIV Insight 5 [INT16]. This device samples at 128 sample per second, with a 14-bit resolution which is sufficient getting research grade EEG data. The frequency response is 0.5-43Hz which is in a good range considering brainwave analysis. Built in sinc filter, input range goes to 8400uV_{pp}. The module again only transmits the data at 2.4GHz band or with Bluetooth. It has an inertial measurement unit for user motion tracking which is also a great option for tracking user movements during the recording. But it can again run only for 4 hours only on Bluetooth and 8 hours on 2.4GHz transmitter with a 480mAh built in lithium polymer battery. So, this device would need external components again to record and store the data.

So, if there was a research study which needed to monitor patient/subject EEG for a long duration of time going on for days after a new treatment or for diagnosing, the patient or the subject would need to switch batteries or charge the devices often and that would mean extra care is required for the device along with intermittent data collection which wouldn't be such a great experience. Along with that they might need another device or media to record the EEG data since most of the devices available are simply EEG acquisition devices and need external recording equipment like a Bluetooth connected device. So, it appears there is a need for devices which can acquire and store the data while keeping the power consumption under control so they can be used for a long time without needing a battery change.

Closed-loop brain monitoring systems also involve brain stimulation techniques such as Transcranial Direct Current Stimulation or tDCS, a form of non-invasive neural stimulation which

uses constant, low direct current delivered via electrodes on the head for brain stimulation [INT17]. tDCS works by changing the neuron's resting membrane potential to depolarize or hyperpolarize based on whether a positive or negative stimulation is applied causing the neurons to either fire spontaneously or reduce the spontaneity respectively. Similarly, transcranial alternating current stimulation or tACS is a form of neurostimulation that delivers a small, pulsed, alternating current via electrodes on the head [INT18]. Both of these techniques have been promising for treatment of some brain disorders. But there are some concerns coming up about the adverse effects of these techniques. As [INT19] notes, the persistent events associated with tCS consist of skin lesions similar to burns, which can arise even in healthy subjects, and mania or hypomania in patients with depression. One paper discussed in the review reported a pediatric patient presenting with seizure after tDCS, although the connection between stimulation and seizure hasn't been made for this particular case. But rarely devices or research talk about stimulating with custom waveform different from the normal DC or AC stimulation waveforms. One such device is Alpha Stim [INT20] which shows promise for pain relief and depression treatment.

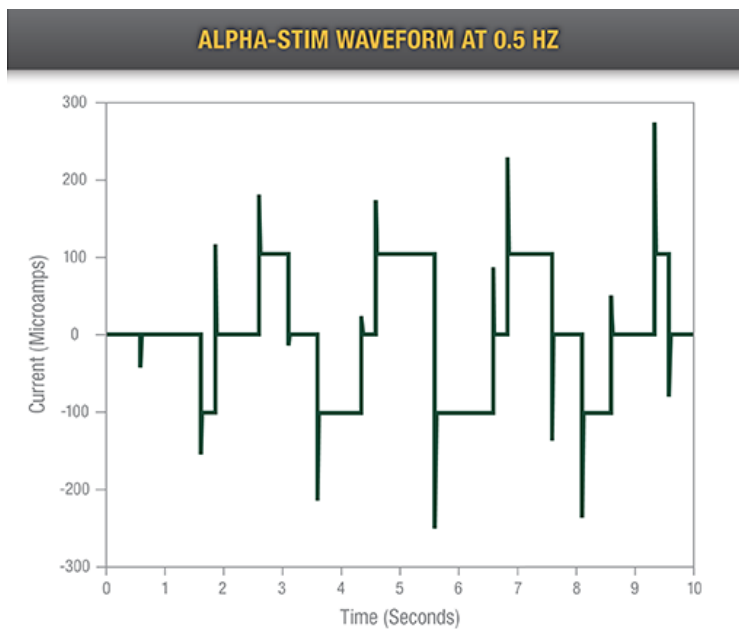


Figure 1.2 Alpha-Stim custom stimulation waveform [INTR9].

1.2. Objective

The devices discussed in this thesis mainly targets the points presented above while trying to keep the system open ended so it can be integrated with other system at the same time. So, the main objectives for the EEG monitoring system are:

1. The device should be able to perform long term EEG monitoring with research grade EEG.
2. The device should be cost effective, or at least be able to be easily built and distributed.
3. The device should be portable and less bulky.
4. The device should be easy to set up and use and shouldn't require any expert knowledge.

Designing a system with these criteria would enable us to achieve goals such as getting a large amount of EEG user data which can facilitate algorithm development and testing; long term monitoring of clinical trial participants with an experimental treatment; long-term monitoring or diagnosing patients with brain disorders and studying effects of certain techniques for teaching or studying by monitoring data from a lot of users. These goals and objectives for the EEG monitoring system are visited in the chapter 2.

The brain stimulation platform being developed and modified here is on already present tACS/tDCS hardware, so the main objective regarding this device is to first get the required custom waveform stimulation output and keep a hardware trigger so that either the EEG monitoring system or any other system can be used to trigger the stimulation platform based on its own signal processing technique/hardware. The non-invasive brain stimulation platform is visited in the chapter 3.

CHAPTER 2

EEG MONITORING DEVICE DEVELOPMENT

2.1. Device Components

The main functions of the device designed here is to obtain user EEG, record it and store it or send it to a common storage location. Since this device is different from the stimulation platform, and only deals with the frontend of EEG storage, it should obtain reliable EEG data with low error, while keeping the power consumption, data overhead, user intervention low for continuous usage.

The device consists of two main parts which combine together with the “Docking Station” (DS) to form the final working system. The two main parts are the “EEG Acquisition and Storage Equipment” (EASE) unit and the headset unit which contains the “ThinkGear ASIC Module” (TGAM). The EASE unit is the offline data recorder for user EEG which can be battery operated and which the user can carry around with them. The headset unit is the frontend of EASE unit and forms the EEG acquisition equipment as it is used to sample and digitize the EEG signal. The Docking Station unit is used to parse the recorded data and store it in MATLAB MAT file format to be easily analyzed and stored/uploaded to the main server. The Docking Station has been implemented until the analysis and storage so that the server upload can be integrated with a system already present for the stimulation equipment.

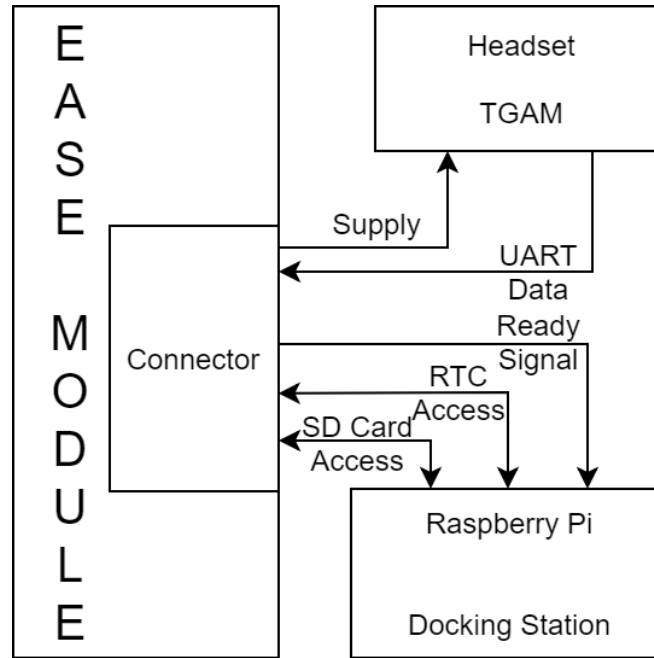


Figure 2.1 Block diagram of the EEG monitoring system.

Fig 2.1 shows the block diagram for the system, with the interconnections between the EASE module, TGAM unit and the Docking Station. A brief description for the parts of this system shown in the diagram, the EASE module consists of the battery-operated device which also powers the headset. The headset containing the TGAM unit samples the user EEG and performs some operations on it and sends the data out via Universal Asynchronous Receiver/Transmitter (UART) protocol. The storage unit or the EASE module consists of a micro-controller which obtains the EEG data from the serial output from the TGAM module and stores it to a micro-SD card. After that, at the end of the storage cycle or when the user wants all the locally recorded data to be parsed and stored, the EASE module is connected to the Docking Station and the data is parsed and stored in the MAT format and then can be uploaded to the main server. More details about the device components and their implementations are given in the coming points.

2.1.1. EEG Sampling Frontend

The goal of the EEG sampling frontend was to obtain reliable research grade with easy to wear electrodes and as low development time as possible. With this goal in mind the ThinkGear ASIC Module was selected as the sampling frontend. The TGAM is NeuroSky's primary brainwave sensor ASIC module designed for mass market applications. The TGAM processes and outputs EEG frequency spectrums according to established brainwave bands, EEG signal quality, raw EEG, and three NeuroSky eSense meters attention, meditation, and eyeblinks. The module works well with simple dry electrodes, and has a low rated power consumption, which is suitable for portable battery-driven applications like the system planned here [TGAM1]. The EEG sampling frontend or the headset consists of the TGAM module which performs the analog sampling of the EEG data based on the placement of EEG electrode, processes it and forms it into a packet based on the configuration of the device and sends it as serial UART data.

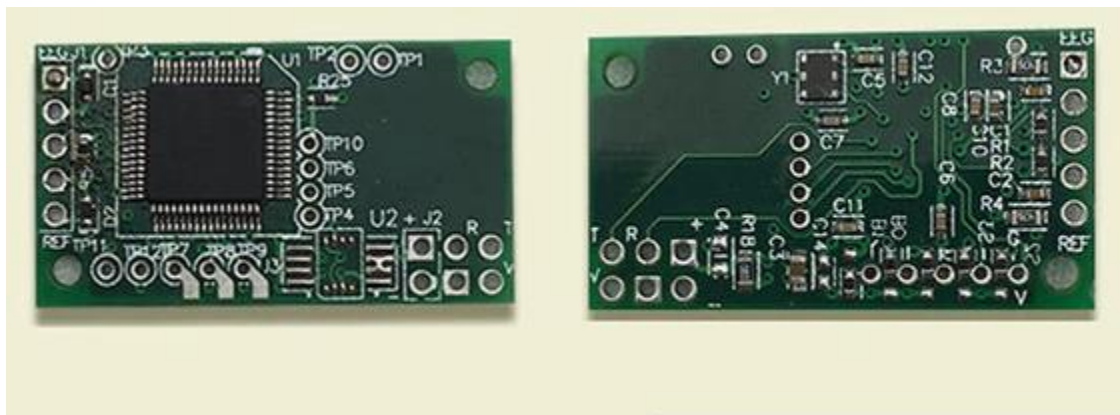


Figure 2.2 ThinkGear ASIC Module Front and Back.

Fig 2.2 shows the TGAM unit. Some of the important specifications of the TGAM unit are shown in Table 2.1 [TGAM2].

Table 2.1 TGAM Specifications.

Classification	Specification
Module Dimensions (Max)	1.1in x 0.6in x 0.1in (LxWxH)
Module Weight (Max)	0.0045 ounces
Operating Voltage	2.97V – 3.63V
Maximum Input Voltage Noise	10mV Peak to Peak
Maximum Power Consumption	15mA at 3.3V
ESD Protection	4kV Contact Discharge 8kV Air Discharge
Output Interface Standard	UART (Serial)
Output Baud Rate	1200, 9600, 57600
EEG Channels	1

The TGAM unit is small, light-weight, operates at the SD Card voltage range (SD card for storage), and outputs data via UART which are great common characteristics for mobile device and for easy interfacing. This is accompanied by the already present ESD protection which makes it easier to integrate as a wearable device. The TGAM’s power consumption is low enough to be able to use it as a battery-operated unit. Moreover, the input voltage is safe for EEG capture and the single channel EEG input is sufficient for the long-term monitoring goals required. According to the prescribed guidelines by the American Clinical Neurophysiology Society [TGAM3], acquisition of EEG data onto a digital storage medium should occur at a minimum sampling rate three times the high-frequency filter setting, to be sufficient to prevent aliasing. Digitization should use a resolution of at least 11 bits per sample including any sign bit. A resolution of 12 or more

bits is preferable, since the recording should be able to resolve EEG down to 0.5uV and record potentials within the range of several millivolts without clipping. The output presented by TGAM is at 512 samples per second, which meets our frequency needs till at-least 150 Hz where the inbuilt low pass filter is present. The maximum input noise range is set at 10mV peak to peak, but the input sampled is allowed to be at 2mV peak to peak, which combined with 12-bit RAW EEG resolution, gives us the ability to resolve the EEG down to 0.488uV per value.

So, as we can see the TGAM meets all the specifications required for clinical EEG monitoring. But all this wouldn't be useful if the device isn't able to record the signals as expected from the input supplied. So, the device was tested with known signals from the function generator and from occipital lobe of a user to test the expected response. Fig 2.3 shows the response given by TGAM for a 30 second, 2 mV peak to peak sinusoidal input signal from a function generator. The graph shows the relative amplitude at each frequency when compared with the frequency component on other frequencies, out of which the dominating amplitude becomes the dominant frequency which is found to be very close to the applied input. The MATLAB [ASIS1] program used to analyze the data is included in the appendix section A.3.2. For the analysis in frequency domain, Fast Fourier Transform (FFT) is applied to the input samples in time domain. Then the result is normalized by dividing it with the length of the signal and wrapped up to half of the result since the FFT result is symmetric. Fig 2.4 shows the time domain signal recorded which again shows the same close response to the applied signal.

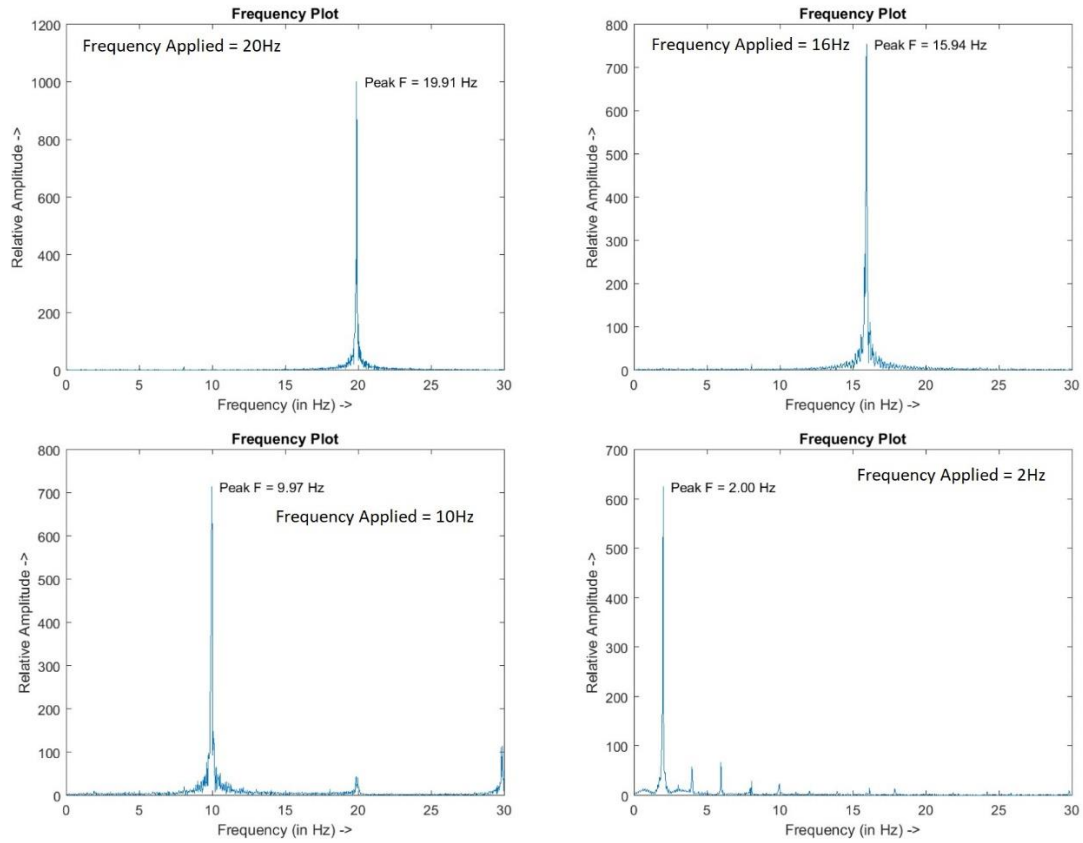


Figure 2.3 TGAM frequency response for input 30sec, 2mV peak to peak signals.

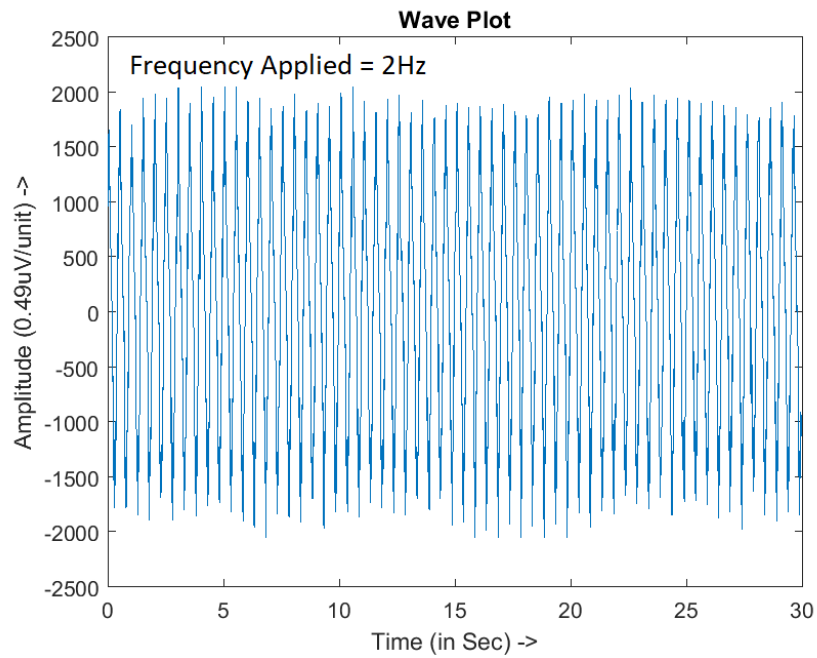


Figure 2.4 TGAM recorded signal in time, 30sec 2Hz.

After confirming the correct signal response from function generator, a known response from a user is checked. The known response selected here is the Berger rhythm or Alpha waves (Alpha waves have dominating frequency component in the range of 7.5Hz - 13Hz) present in the occipital lobe as illustrated by past studies. First demonstrated in 1929 by Hans Berger [ASIS2], then also observed in [ASIS3] and reiterated in various other studies, the alpha waves can be observed easily when the eyes are closed. One of the early examples of this is shown in Fig 2.5 from [ASIS3] where the recording was done with 6000Hz sampling frequency in (A) and 40 Hz in (B) and (C), where all of the subjects had eyes closed. The resulting dominant frequency in (A) was 9.5Hz, (B) was 9Hz and in (C) was 10.5Hz, all in alpha wave range.

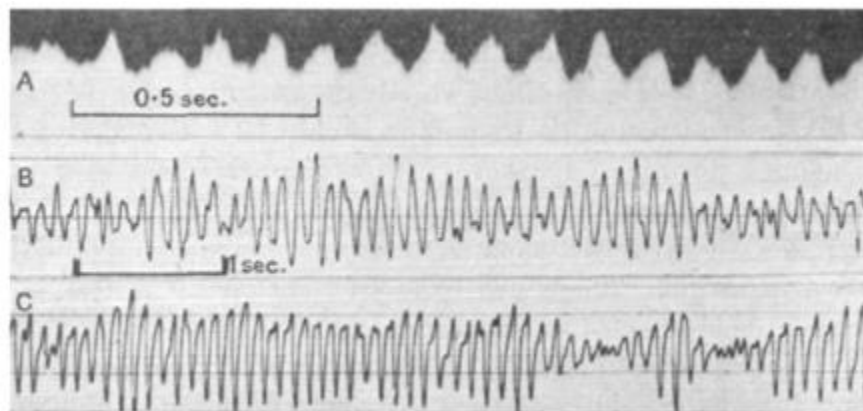


Figure 2.5 Records of the Berger rhythm made with pad electrodes on the head (vertex and occiput), with the subjects sitting with the eyes closed. [ASIS3]

Fig 2.6 shows the frequency response from the TGAM which is obtained after connecting the EEG electrode to the occipital lobe, while the reference and ground electrodes are connected to the ear lobe of the subject and the data is recorded for 30 sec duration. Fig 2.7 shows the output in time domain after a 50Hz 8th order Chebyshev type 1 low pass filter is applied to the captured samples. From this response we can see that there is a reliable signal present which we can confirm to be having the correct form as expected.

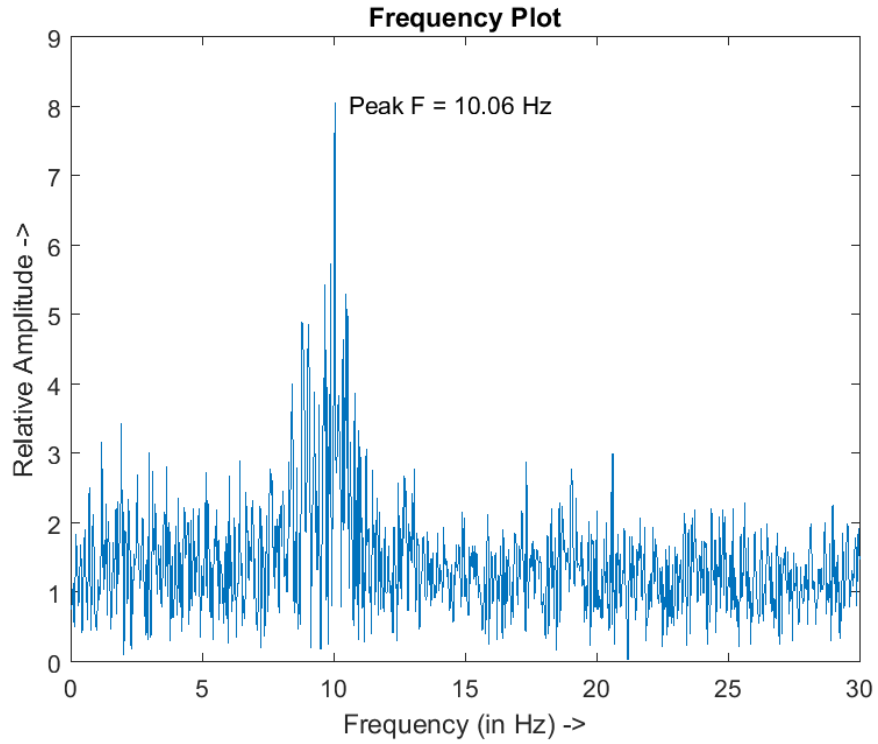


Figure 2.6 Frequency response for 30 sec eyes closed occipital lobe data using TGAM.

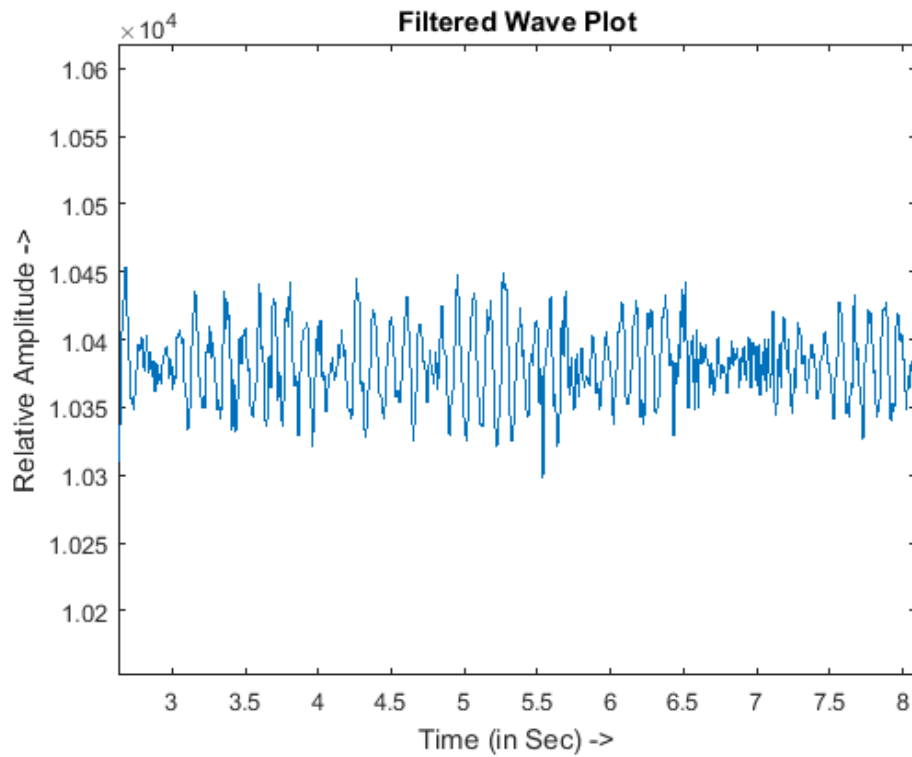


Figure 2.7 Recorded, low pass filtered signal at 50Hz for eyes closed occipital lobe data using TGAM.

The TGAM also has inbuilt fixed band pass (or a combination of low pass and high pass filters) and a configurable notch filter which was observed while using the device. This might present a limitation if the fixed filters were masking a considerable amount of signal which we need. According to the American Clinical Neurophysiology Society (ACNS) guidelines, for standard recordings, the low-frequency filter should be no higher than 1Hz and the high-frequency filter should be no lower than 70Hz. A low-frequency filter setting higher than 1Hz should not be used because vital information may be lost when pathologic activity in the delta range (which has a dominating frequency component till around 3Hz) is present. Similarly, a setting lower than 70 Hz for the high-frequency filters can distort or attenuate spikes and other pathologic discharges into unrecognizable forms and can cause muscle artifact to resemble spikes [TGAM4]. For this a frequency sweep across 0.1Hz – 140Hz resulted in the response shown in Fig 2.8 for TGAM.

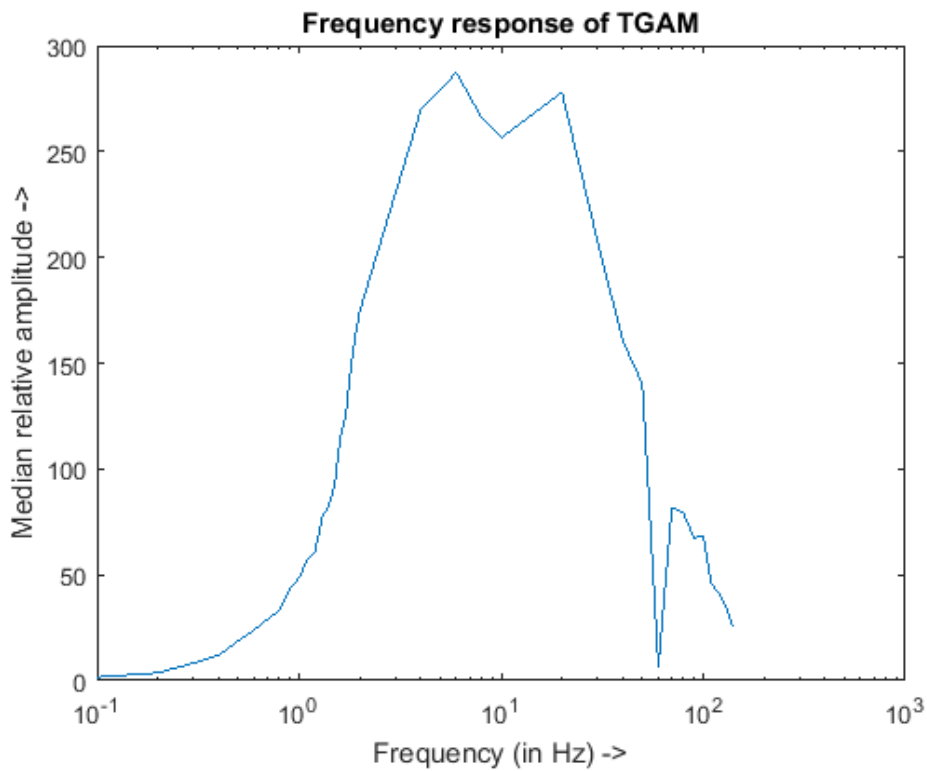


Figure 2.8 Frequency response of TGAM over 0.1Hz – 140Hz input.

As we can see from Fig 2.8, there appears to be a low pass filter around 100Hz applied frequency and a high pass filter around 1Hz frequency, along with a notch filter at 60Hz (the TGAM has a hardware option to change this notch filter's frequency to 50Hz, to adjust for the power line noise depending on the country). The frequencies applied for analysis were 0.1Hz, 0.2Hz, 0.3Hz, ..., 1.7Hz, 1.8Hz, 1.9Hz, 2Hz, 4Hz, 6Hz, 8Hz, 10Hz, 20Hz, 30Hz, ..., 140Hz at 100uV using a function generator. Based on the frequency response it seems that the response given by TGAM falls under the guidelines for clinical EEG monitoring, which is why it was chosen as the EEG sampling frontend.

2.1.2. *'EASE' Module*

Since the frontend is dealing with the sampling of EEG signals and digitization, the primary goal of the next component is to provide local offline storage for the sampled data coming from the TGAM. The secondary goal is to keep track of real time which would help to track abnormalities in EEG data or to timestamp certain events according to when they happened or form a correlation between them. The final goal for this stage is interfacing the EASE module with the Docking Station to provide it access to the local storage and update the time if needed. So, keeping these goals in mind the EASE module or the storage unit needs to have a controller that carries out the following tasks: store the data coming from UART to a widely available local medium like a SD Card, and timestamping the data based on when the data is being stored. So, for simplicity of the design and easy interfacing, the ATMEGA328PB controller [EASE1] is present on the EASE module, along with a Real Time Clock (RTC) which works on Inter-Integrated Circuit communication (I2C) [EASE2], a SD Card as the storage medium which works with SPI [EASE3] communication and a level converter is used for the communication between 5V and

3.3V voltage levels between the microcontroller and other peripherals. The EASE module is the main board that the user handles and to keep the user intervention minimum there is a single switch on the EASE module which the user has to handle along with connecting the headset to the EASE module or connecting the EASE module to the Docking Station at the beginning of EEG capture and at the end when the data has to be parsed and stored respectively.

2.1.3. Docking Station

The Docking Station comprises the next step where the data from the local offline storage on the EASE module has to be downloaded, parsed and stored in an easily usable format like MAT format [DOCK1] which can be accessed at a later stage. Taking this into consideration, the Raspberry Pi is kept as the brain of the docking station to enable easy processing of the data and easy upload capabilities to the common cloud storage. The Raspberry Pi in the Docking Station connects to the Real Time Clock on the EASE module and updates it using internet time (GMT [DOCK2]) whenever the EASE module is first connected to the Docking Station. After that it connects to the storage element on the EASE module and downloads the recorded data stored on it. The Docking Station is powered by a micro USB power supply which are easily available and connects to the internet using wired connection from the modem/router.

2.2. System Architectures

The system architecture/block diagram combining the headset, EASE module and the Docking Station are shown in Fig 2.9. The signals forming the communication between the devices are shown in the figure along with their direction where they originate and where they reach.

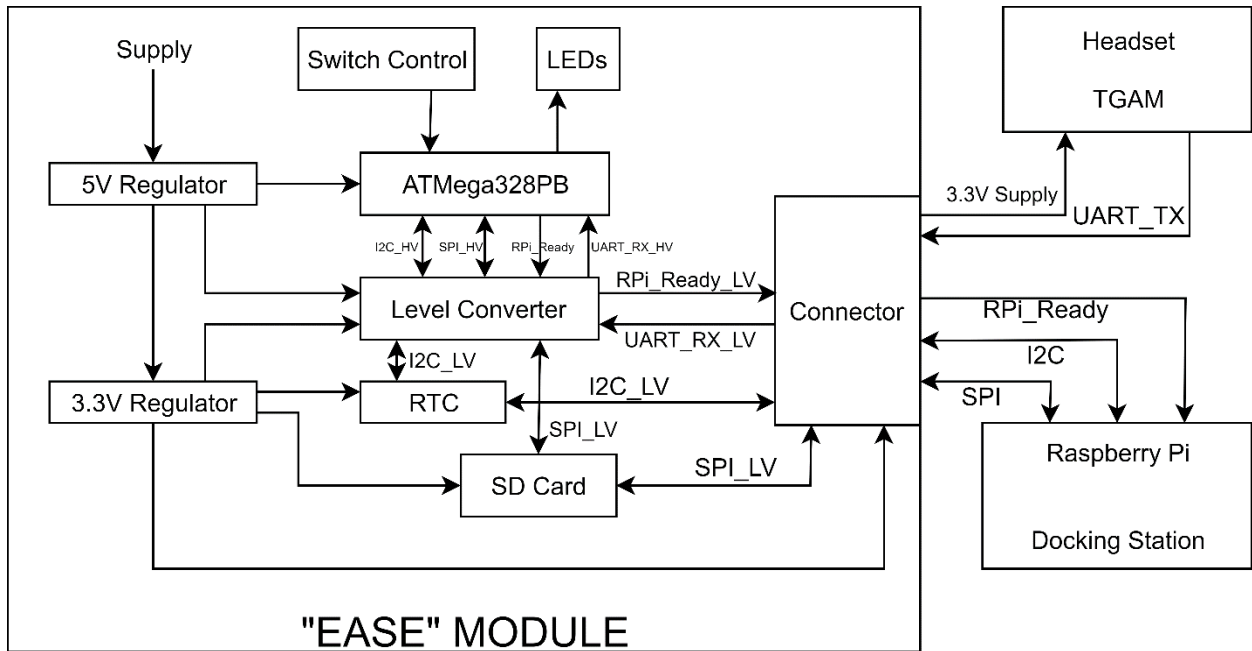


Figure 2.9 EEG System Architecture.

As shown in the figure, the heart of the EASE module is the ATMEGA328PB microcontroller which handles the EASE module and its peripherals primarily. The signals coming from this microcontroller are level translated for the Real Time Clock, SD Card and Raspberry Pi signaling whose control also goes to the connector for the Docking Station. The incoming UART signal from the headset connected to the connector is also level translated before sending it to the microcontroller and the headset is also powered using the supply pins from the connector. More hardware and interfacing descriptions are given in the next sub-chapter. The user can use the system according to the flowchart in Fig 2.10 and as described below.

1. The battery or supply to the EASE module is turned ON and the switch on the EASE module is turned OFF at the beginning.
2. The EASE module connector is then connected to the Docking Station connector, this enables the Docking Station to synchronize the time on Real Time Clock to the current GMT. This step can be ignored under these conditions:

- a. If the timestamping is not necessary for the data capture.
 - b. If the Real Time Clock is powered by another battery like a coin cell.
 - c. If the current battery hadn't been disconnected and the Real Time Clock has been synchronized with internet time before.
3. Now, the EASE module is disconnected from the Docking Station and its connector is connected to the headset connector and then user puts the headset on. Once the headset is properly connected by the user, the switch on the EASE module can be turned ON. After this switch is turned ON, the controller on the EASE module will first try to connect to the SD card which is shown by the GREEN LED turning ON. If the connection to the SD card fails the GREEN LED will be ON showing the microcontroller is still trying to connect to the SD card in which case the user has to turn the switch OFF and recheck if the SD card and the headset connector is properly connected or not then turn the switch back ON. When the GREEN LED turns OFF and YELLOW LED turns ON, the microcontroller starts storing the data coming from headset to the SD card.
4. The switch is kept ON till the user needs to record the data and turned OFF when the data recording has to be finished for further processing.
5. Now, the EASE module connector is disconnected from the headset and connected to the Docking Station connector. The Docking Station then again synchronizes the time for the Real Time Clock on the EASE module to the current GMT, and begins trying to connect to the SD card on the EASE module. Again, in this case, the GREEN LED on the Docking Station connector will turn ON, and if it is ON for more than a short period of time it indicates a connection problem with the SD card in which case the EASE module connector should be reconnected to the Docking Station or the SD card connection should be checked

after disconnecting EASE module from the Docking Station. Once the GREEN LED turns OFF and YELLOW LED turns ON, it indicates the data download from the SD card has been initiated. When the download has been finished, the YELLOW LED will turn OFF and the EASE module can be disconnected from the Docking Station, and also that the Docking Station has started parsing the data and storing it in MAT format which can be later uploaded to a server. Then the flow goes back to step 3 after the EASE module is disconnected from the Docking Station.

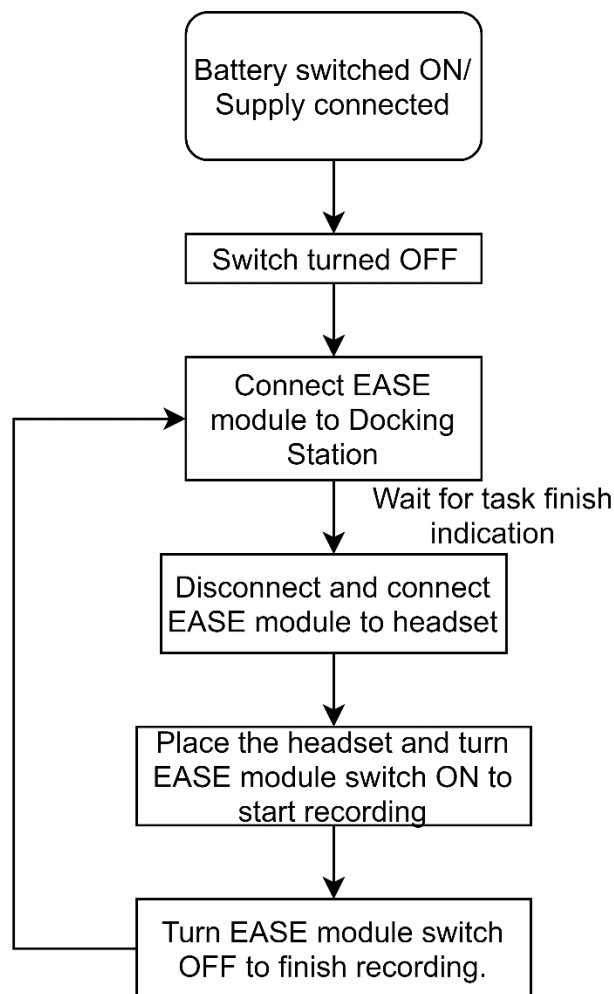


Figure 2.10 EEG System Flowchart.

2.2.1. 'EASE' Module Architecture

The development of the software for the EASE module was started with the Arduino platform and Arduino IDE [EASE4]. Later when the hardware for the first prototype was designed, the development stayed with Arduino IDE but the hardware changed, for better integration. The final design interfaces with the hardware blocks on the EASE module as shown in Fig 2.9, along with the communication taking place with these peripherals by the microcontroller. From the task assigned to the EASE module, the primary task that the microcontroller needs to handle is to capture the data with the least packet loss possible while storing that data in the SD card at the same time. The secondary tasks include getting the user input from the switch control, interfacing with Docking Station, controlling the division of storage files according to recorded time and getting timestamps from the Real Time Clock.

The program makes use of SD library which is a wrapper for the SdFat library which provides read/write access to FAT16/FAT32 file systems (so the maximum possible size is 32GB) on SD/SDHC flash cards, SPI library for communication with the SD card and Wire library by Arduino for I2C peripheral access for the communication with Real Time Clock chip PCF8523 [EASE6]. The files to be recorded on the SD card are recorded till a predefined duration to properly divide the files for easier analysis. The recording time can be supplied to the program before compilation, it is provided in seconds and the required variables used by the program related to it are calculated at the start. So, the program keeps track of data to be downloaded based on the duration and the bytes of data from UART which we expect to receive. According to the communication protocol for the TGAM [TGAM5], for the 57600 Baud UART Tx option, the device transmits 2 packets: the RAW EEG waveform and the other packet contains the calculated parameters related to the EEG wave. The RAW EEG waveform packets are sent 512 times per

second, according to the sampling rate and the bytes they take for each packet are: 2 (Sync Bytes) + 1 (Packet Size) + 1 (Type of Data) + 1 (Size of Data, since RAW value is multiple bytes) + 2 (RAW EEG value) + 1 (Checksum) = 8. The other packet which is sent contains the signal quality measure, different EEG bands, eSense measures developed by NeuroSky, and its total size comes out to be 36 bytes and this packet is sent once every second. According to this, the total bytes we receive each second are $(512*8) + 36 = 4132$. The UART data is captured and written to the SD card in a block of 256 bytes at a time which can be used to measure the time passed after recording the data. So, before compilation the number of blocks to be captured are calculated according to the supplied file duration and the remaining bytes are captured before closing the open file. The way these blocks are calculated is simply by multiplying the bytes per second that we receive from the device by the capture duration and dividing it by 256 which is bytes per block in the buffer which we store to the SD card, which will give us an integer value of blocks to calculate. After that, we can measure how many bytes will be remaining after capturing these blocks and store that value. For example, consider the capture duration to be 45 seconds. So, the total bytes to be captured becomes $45*4132 = 185940$, which means total blocks to be captured initially becomes $185940/256 = 726$. So, the remaining number of bytes to be captured becomes $185940 - (256 * 726) = 84$. So, the program captures 726 blocks of data measuring with a counter and then at the end it captures 84 bytes before closing the file.

The high-level flow of the software/program used for the microcontroller on the EASE module is shown in the flowchart in Fig 2.11 and described below:

1. At the beginning, all the inputs/outputs are configured and the initial output configuration is set. The RPi_READY signal is turned OFF to show that the Docking Station can use the peripherals. Then all the LEDs except the RED LED are turned OFF. The RED LED shows

that the device is either connected to the Docking Station or it is ready to capture EEG data from the headset if connected. Then, the program starts polling the switch to see if it is ON or OFF with a blocking delay of 20mS.

2. When the switch is turned ON, the program checks for a switch debounce by checking the switch after a small delay and if it is still ON, the program proceeds to next step. So, when the switch is confirmed to be ON, the RED LED is turned OFF and the GREEN LED is turned ON to show that the microcontroller has started trying to initialize the SD card. If the microcontroller wasn't able to initialize the SD card and the switch is still ON, it again tries to initialize the SD card. During this stage, the GREEN LED might be ON for a long time from conditions such as the user not connecting the SD card, or a contact problem between the SD card and the SD card housing, or a corrupt SD card, or if the EASE module connector wasn't connected to the headset connector. So, the user has to check for these problems and once finished the SD card will be initialized properly.
3. After the SD card is initialized properly, the GREEN LED is turned OFF and the YELLOW LED is turned ON indicating the start of data recording and the program moves to next state. At the beginning the microcontroller reads the time from the Real Time Clock and uses it to create a filename of the format “‘day’‘hrs’‘mins’‘sec’.txt” where day, hrs, mins and sec each take 2 characters. From this we get the final filename of 8.3 format confirming to the FAT file system naming standards [EASE5]. The controller then opens a file with that name for writing on the SD card. The timestamp at the time of creation along with the format of timestamp is stored in the file at the start and the program moves to the next state.
4. Now, the microcontroller starts capturing data. The data is captured in blocks as mentioned earlier and the switch is also verified at the same time to be ON. If the switch stays ON,

the data is captured for the capture duration limit specified at the beginning and the file is closed to open a new file after reading the current datetime from the Real Time Clock and the capture resumes. Whereas if the switch is turned OFF during recording, the current block of captured data is stored in the SD card then the currently open file is closed and the program moves to next state.

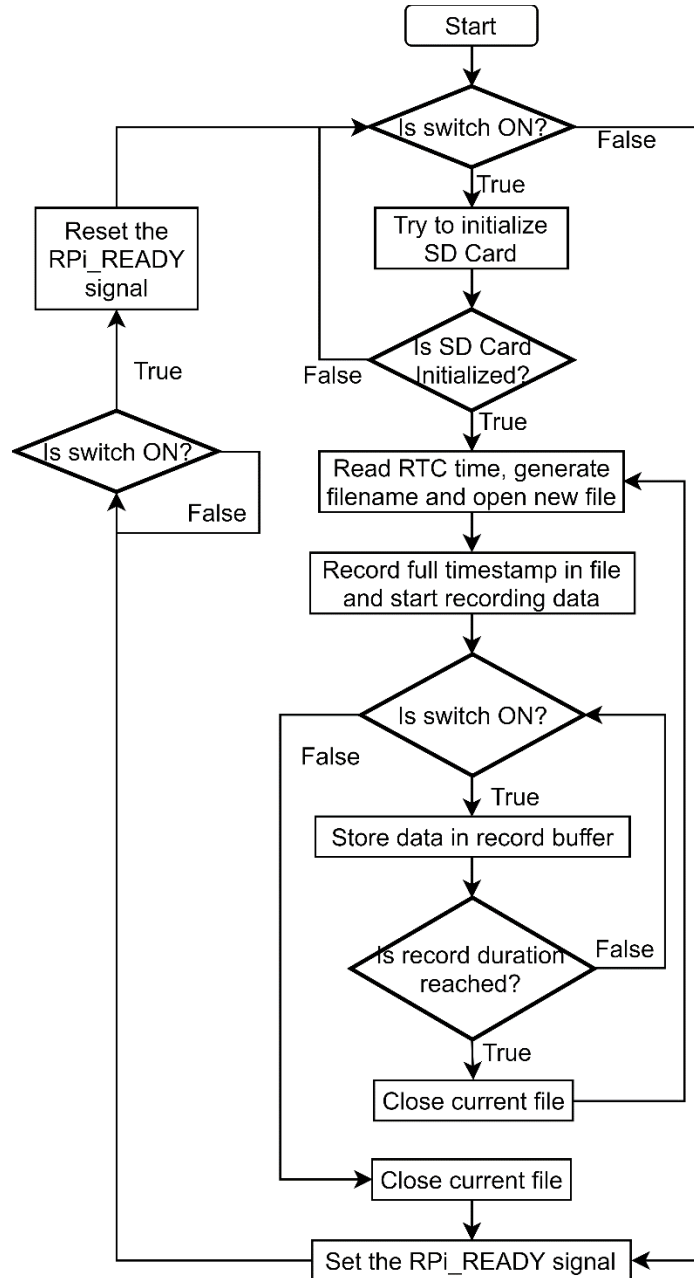


Figure 2.11 EASE module flowchart.

5. Once the capture is finished, the YELLOW LED is turned OFF and RED LED turns ON and the RPi_READY signal is set giving control of the peripherals to the Docking Station. The program again resumes polling the switch to be turned ON with a blocking delay of 20mS in between. When the switch is pressed, the program goes back to step 2.

The program for the EASE module is included in the Appendix section A.1.

2.2.2. Docking Station Architecture

The Docking Station consists of the Raspberry Pi 3 Model B, a very small sized computer on board and a small connector extension module which takes care of the connection between the Raspberry Pi GPIO and the connector pins. The primary task of the Docking Station is to download the data from the SD card on the EASE module, update the Real Time Clock on the EASE module, parse the downloaded record data and store it in MAT format to be uploaded or to access it later. The development of the Docking Station system is divided into 2 parts:

1. The main python program which handles data parsing, Real Time Clock updating, some of the LED signaling for the user, and normal GPIO operations.
2. A C program for faster operation which handles all the communication and procedures related to the SD card and the remaining LED signaling.

Since the data recorded by the EASE module is direct input from the UART signal, it has to be parsed before it can be used for anything. So, the parsing performed on the Docking Station takes care of that using the details of packet given in the TGAM communications protocol document [TGAM5]. All the packets contain 2 sync bytes of value '0xAA' at the start. After that, for the RAW EEG waveform samples, the length of payload is 4 bytes which is the next byte that

comes. Then the next 4 bytes contain the payload which has the type of data coming in ('0x80' for RAW EEG waveform), the size of data (which is 2 bytes for RAW EEG, since the value sent is a 12-bit signed value, so '0x02') and 2 bytes for the RAW EEG waveform. After the payload comes the checksum byte for the payload is sent and the packet finishes. The 2 bytes for the RAW EEG waveform contain the bytes in 2's Complement Signed format with the incoming data being in a big-endian format. So, the format for the packet becomes, "'0xAA' '0xAA' '0x04' '0x80' '0x02' D1 D2 C0" where $(D1 \ll 8 + D2)$ form the value of the current EEG sample and C0 is the checksum. The checksum can be calculated by summing all the bytes of the packet's data payload, taking the lowest 8 bits of the sum and performing the bit inverse (one's complement inverse) on them. Using this packet information, the python program parses the recorded data for all the samples it can find. For storing the data parsed in MAT format the program uses scientific python library, SciPy [DOCK3] and for Raspberry Pi GPIO access the RPi.GPIO library is used which are both inbuilt with stock Raspberry Pi 3 Raspbian [DOCK4] installation. The python program handling the main control flow of the system on Docking Station is given in appendix A.2.1, the program controlling the Real Time Clock handling is given in appendix A.2.2. The program handling the main control flow of the Docking Station runs with the Raspberry Pi at the bootup, so the Docking Station doesn't need to be always ON, it can simply be turned ON when the processed data has to be stored to a common media or uploaded.

The SD card interfacing program written here makes use of the FatFs library [DOCK5]. FatFs is a generic FAT/exFAT filesystem module for small embedded systems. It is separated from the disk I/O layer, so it is platform independent. The SPI pin access to the FatFs library is given with software SPI on the GPIO pins with low level GPIO access by Broadcom library. The C program working with the SD interfacing is included in appendix A.2.3 and the parts of the

program defining pin controls for the software SPI to use along with FatFs library using the Broadcom BCM2835 library (pin mapping to the register addresses) are given in appendix A.2.4.

The flow of the main program controlling the execution of tasks primarily along with the supporting SD card handling program for the Docking Station is shown in the flowchart in the Fig 2.12 and described below.

1. When the program starts, all the GPIO pins are initialized, and all the LEDs except RED are turned OFF indicating that the device is ready to be connected. Then the program goes in wait state, waiting for the RPi_READY pin to go low. When there is a falling edge on this pin, the pin is again checked for being pulled to ground after a small delay in case there was some error. If it still is pulled to ground, that means the EASE module is connected to the Docking Station and we can go to the next stage.
2. The peripherals on the EASE module are ready to be accessed by the Docking Station. At the start the current GMT is taken from the internet and the Real Time Clock is updated to this time. The RED LED is turned OFF and the GREEN LED is turned ON showing that the program is moving to try to connect to the SD card and initialize it and the program moves to next stage.
3. The C program tries to initialize the SD card now. If the initialization fails because of no input coming from the SD card, the SPI speed is reduced and the program tries to reinitialize the SD card if the SD card supports slower speeds. In case nothing works till the speed is slowed down by a lot, the SD card and the connection between EASE module and Docking Station has to be rechecked and the EASE module reconnected. Once the connection is successful, if the speed was reduced to initialize the SD card, the program tries to reconnect with faster speed in case it was just a communication problem and then

moves on with the selected speed. So, once the SD card initialization has been finished, the GREEN LED turns OFF and the YELLOW LED turns ON indicating that the data download is going on.

4. After establishing the connection successfully, the program takes a list of files present on the SD card, checks for the files with extension '.txt' and starts downloading them one by one. After a file is downloaded, it is removed from the SD card since it has been stored. In case there is a problem with the connection with SD card, the program will retry slowing down the speed and blink GREEN LED each time it does that. In such a case that the GREEN LED keeps blinking it shows that there is a problem with the connection with the SD card and the download is failing. So, the process has to be restarted after checking connection with the SD card, and the connectors and the files which were downloaded before wouldn't need to be downloaded again, only the ones that haven't been finished will be taken care of. After the complete download finishes, the YELLOW LED is turned OFF and the BLUE LED is turned ON to signify that the data is being parsed now. Once the YELLOW LED is turned OFF, it is safe to disconnect the EASE module from the Docking Station and can be used right after connecting the headset to it.
5. The Docking Station starts parsing the data one by one according to all the downloaded files it can find and stores it in MAT format. After each recorded file is parsed and stored in MAT format, its supporting files other than the final MAT file are removed to store space and clear unrequired clutter and so that those files don't get processed again. After the parsing finishes, the BLUE LED is turned OFF and the RED LED turns back ON signaling that the Docking Station is ready for connection again.

6. At the end of finishing all tasks, the program again waits for the RPi_READY signal to go back to HIGH state. This is done since if the EASE module was still connected to the Docking Station in OFF mode after the Docking Station finished its tasks, the RPi_READY will be continuously pulled to HIGH and would trigger another connection procedure which is not right. So, after waiting for the EASE module to get disconnected now the program goes back to the top to check if the EASE module is connected by checking the RPi_READY signal going LOW after which the program goes back to step 2.

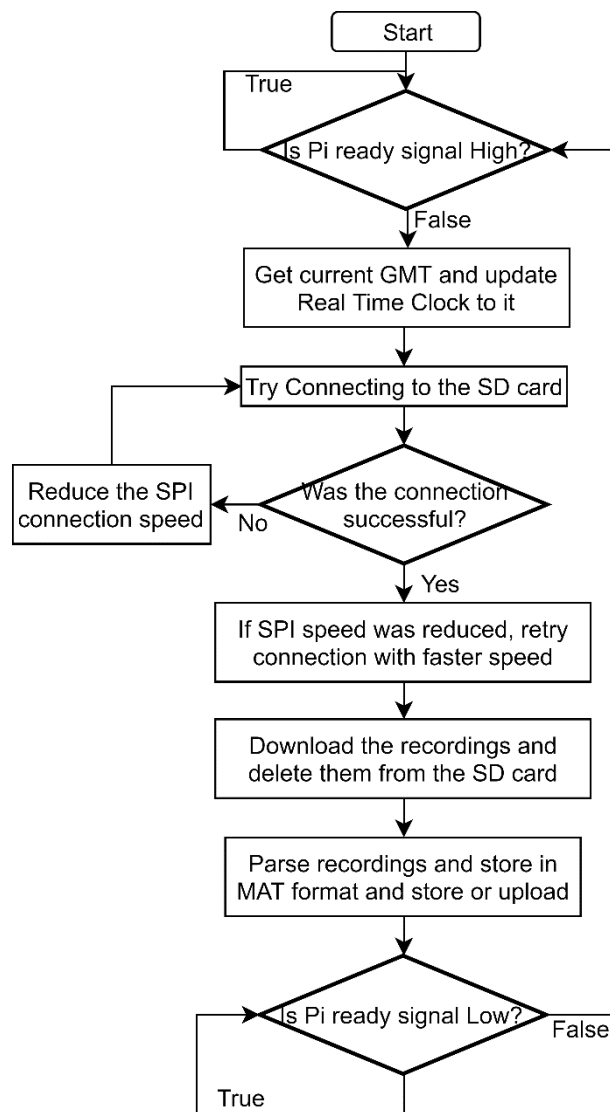


Figure 2.12 Docking Station flowchart.

2.3. Hardware Interfacing and Design

2.3.1. Interfacing and Design Choices

Similar to the programming choices, the main goal of the hardware design and interfacing was to get the required components and program flow working with each other with as less components as possible. The main interfacing is taken care of using only the EASE module so that the headset and Docking Station do not need to have a big external circuit with them. Initially, the prototypes for the EASE module were prepared using the Dual-Inline Package ATmega328P microcontroller IC, and the final prototype makes use of the ATmega328PB with TQFP packaging. The voltage regulators used on the EASE module are both having very low quiescent current and dropout voltage so that the applied input voltage range is maximized and can be started close to the regulated levels for lower battery usage. There are two voltage regulators used, first one at 5V, LP38691-5V [HARD1] which has a dropout voltage of 45mV at 100mA current output and a quiescent current of 55uA and it is used in TO-252 packaging to overcome any potential heating problems there might be from continuous device operation, the second regulator is supplied by the output from the first regulator since its input voltage range is selected lower and it gives the output at 3.3V, ADP121-3.3V [HARD2] which has a dropout voltage of around 35mV at 50mA current output and a quiescent current of around 22uA but these parameters do not matter for the 3.3V regulator since it receives a stable supply from the 5V regulator so the current is forwarded to the 5V regulator, with the dropped voltage dissipating as heat across the 3.3V regulator based on the current consumed by peripherals connected at 3.3V.

For the ATmega328PB microcontroller on the EASE module, the basic components required for interfacing the IC for proper working have been used such as connecting all the supply pins to the voltage regulator, connecting the 16MHz crystal oscillator along with the proper load

capacitances, RESET pin in pull-up configuration for continuous operation, GPIO connections for the switch and LEDs and SPI and I2C signals and the pinouts for ISP [HARD3] programming for the microcontroller. Now, according to the technical standards for the safety of medical devices [HARD4], there should be an isolation between the wearable medical device and the mains power supply. Since the only places where any part of this system touches the mains power supply is at the supply for Docking Station and supply for the battery charger, it can be completely isolated by isolating the connection of the EASE module with the Docking Station to be only done when the user is not using the device. In this way the possible case of electrostatic discharge through the mains supply is removed, and for the ICs and peripherals on the device we already have some or the other ESD protection present so they will not be damaged when the device is connected to the Docking Station or battery charger by itself. So, the easiest way to do this without adding additional components to the design is to keep the connector for the headset and the Docking Station connection to be the same one. This means the EASE module can only be connected to the Docking Station when the user takes off the headset.

Now, the connector has to connect to both the headset, where it has to supply power to the headset (2 pins - +3.3V, GND) and take a pin for UART reception (1 pin – UART RX), and to the Docking Station where it has to give the direct connection to the SD card using SPI pins (4 pins – CS, SCK, MISO, MOSI), give the I2C pins (2 pins – SDA, SCL), the common ground (can be combined with headset supply's ground) and the RPi_READY signal (1 pin), so the total number of pins come out to be 10. The ATmega328PB working at 16MHz can only operate at that frequency if it is supplied 4.5-5.5V [EASE1], which is why there is a 5V regulator used for the controller and the SD card, the Raspberry Pi GPIO signals and the TGAM can only work at 3.3V by default so there is the 3.3V regulator whereas the Real Time Clock can work at both voltages,

so it is kept at 3.3V for matching between the peripherals and for future provision for the Real Time Clock to have an independent battery. This means the communication signals from the ATmega328PB need to be translated before they reach the SD card or Raspberry Pi. The incoming or input signals work fine even if they are at 3.3V since it is within the allowed lower limit of HIGH-level input for ATmega328PB which is given by $0.6 \cdot V_{cc}$ which comes out to $0.6 \cdot 5V = 3V$ but it is still very close to the allowed input limit. So, a level translator is used in order to take care of communication between 5V peripherals and 3.3V peripherals, both input and output. Now, the system controlled by EASE module controller and the system controlled by the Docking Station controller both need to be able to access same main components i.e. the Real Time Clock and the SD card. Since the Real Time Clock uses the I2C line which is bidirectional, there is no problem but same cannot be said for the SPI line where the SD card is accessed. So, if we want to keep the ATmega continuously connected to the SD card, there might come a condition that the user forgets to turn OFF the ease module in which case the ATmega might still be communicating with the SD card and if in this case the EASE module is connected to the Docking Station there can be a short circuit, but other than that the SD card and Real Time Clock access are mutually exclusive for the EASE module and Docking Station if correct procedure is followed. For this, there can be solutions to avert it using extra hardware switching between the controllers for the SD card or software precautions for not letting this happen. So, instead of this, the low voltage signals coming from the level converter from the ATmega can be simply disconnected when the user switches from the headset to the Docking Station, since this is a task which cannot be skipped since the connector for both of them is taken to be same. So, the only control from the pins directly connected to the SD card are going to the connector, and from there when the headset is connected the connection from ATmega is completed and when the Docking Station is connected the control

goes to it, and we can avoid changes to the hardware or software which would take care of the same condition.

The hardware design choices explained here can be seen on the circuit from the schematics in the next section.

2.3.2. Circuit Description:

The EASE module and the Docking Station header circuits were designed for creating a custom PCB for easy connection and communication.

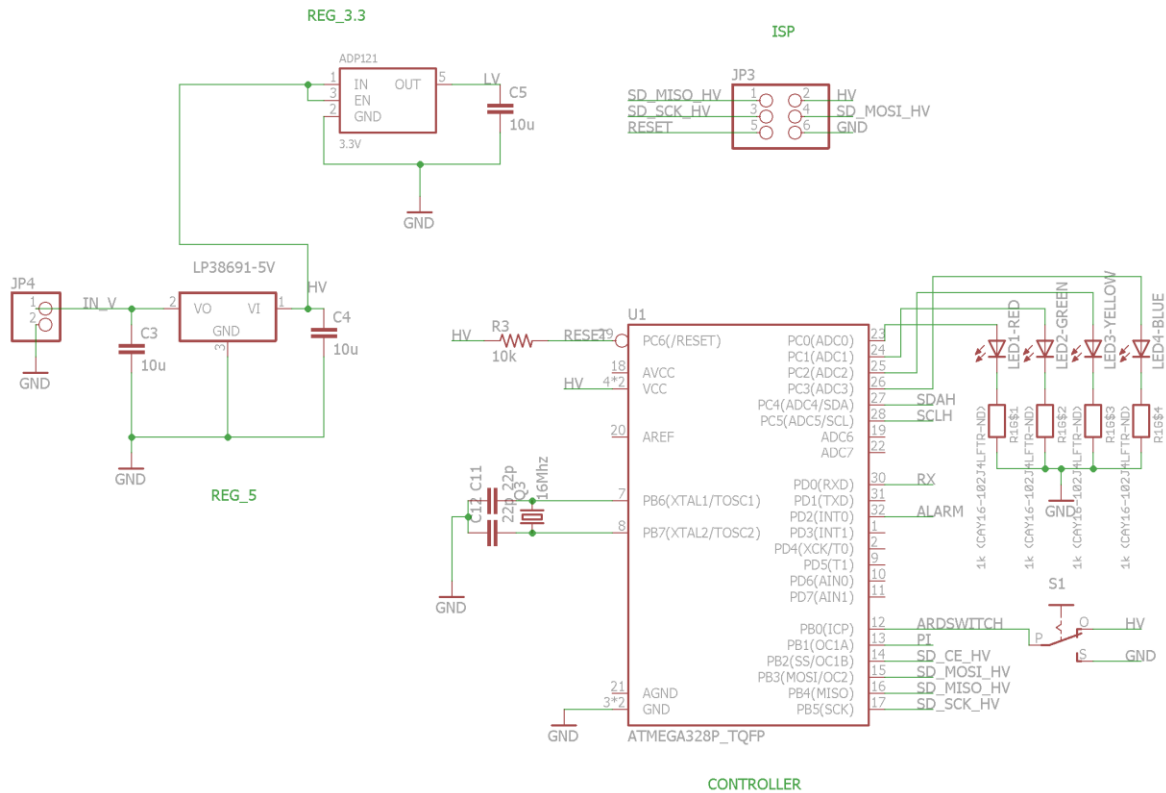


Figure 2.13 EASE module GPIO and support circuitry.

capacitance [HARD5], same as the default load capacitance set by the Real Time Clock chip so the clock is matched. The battery supply pin is left open so that with future hardware updates a local battery can be used for the Real Time Clock chip considering its low power usage, and the alarm signal is sent to the microcontroller as an interrupt from the Real Time Clock chip in case the day/hour/minute alarm pin interrupt functionality is used in the future software updates (The alarm functionality can be used for deciding the record time as we saw before. The minimum resolution is in minutes, so only a minute alarm clock can be set using the Real Time Clock whereas with current software the record time is selected in seconds). For the SD card housing, only the supply is directly connected, its SPI pins go to the connector. All the pins connected on the high voltage side for the level converter go to or come from the microcontroller, whereas the signals on the low voltage side go to or come from the connector (SPI signals, I2C lines, TX is the signal coming from UART TX of TGAM, PI_INT is the RPi_READY signal going to raspberry pi), also the I2C lines on the low voltage side go to the Real Time Clock too.

The common connector on the EASE module discussed according to the section 2.3.1 contains the signals shown in Fig 2.15. The signals with ‘SD_CE_LV’, ‘SD_MOSI_LV’, ‘SD_MISO_LV’, ‘SD_SCK_LV’, ‘PI_INT’, ‘SDAL’, ‘SCLL’ come from the level converter. The ‘SD_CE’, ‘SD_MOSI’, ‘SD_MISO’, ‘SD_SCK’ go directly to the SD card. So, for the connector configuration when the headset connector is connected, the TGAM is supplied by the LV and GND pins and its UART TX goes to the TX pin. The headset connector also connects the LV signals to the SD card signals so that the microcontroller can access the SD card, i.e. ‘SD_CE_LV’ is connected to ‘SD_CE’, ‘SD_MOSI_LV’ is connected to ‘SD_MOSI’, ‘SD_MISO_LV’ is connected to ‘SD_MISO’, ‘SD_SCK_LV’ is connected to ‘SD_SCK’ and the other pins are left untouched. The Docking Station connector uses the PI_INT signal, the GND, the I2C signals and

the direct connections to the SD card, so when the Docking Station connector is connected only it has the access to the SD card.

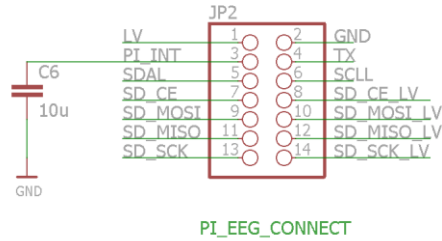


Figure 2.15 EASE module connector for headset and Docking Station.

2.4. Results

The system was tested both using a predefined input signal from a function generator and also with the eyes closed occipital lobe data of a test subject. The test protocol that was followed for testing the prototypes is as follows:

1. Setup input known signal for the EEG electrode. Provide supply for the EASE module.
2. Verify that the EASE module is OFF – its RED LED should be ON, then connect it to Docking Station when its RED LED is ON to synchronize time for the first time. Then disconnect the Docking Station when the Docking Station light is RED again.
3. Place the headset on the head and make sure that the electrode/gel touches the skin. Alternatively, for testing connect the known signal from function generator to the headset's EEG electrode and apply the ground signal from the function generator to the reference and ground electrodes. Connect the headset to the EASE module.

4. Turn the EASE module ON. The GREEN LED should light up for a bit then it should turn OFF and YELLOW LED will light up indicating that data record has begun. If the module is stuck on GREEN LED, turn it off and make sure the headset is connected properly or the SD card is properly inserted then try again.
5. Turn OFF the EASE module when the data record has to be finished, by default the files will be divided into 30 Sec long files, changing the value in source code will change the record duration.
6. Disconnect the headset and connect the EASE module to the Docking Station. The GREEN LED should light up for a bit then it should turn OFF and YELLOW LED will light up indicating that data download has begun. If the Docking Station is still stuck on GREEN LED, disconnect the EASE module and make sure that the SD card is properly placed in its socket and the connection between the EASE module and Docking Station is good then connect them again.
7. Docking Station updates the Real Time Clock time, then connects to the SD card (GREEN LED), then downloads data from the SD card (YELLOW LED) and then it parses sample data and saves it in MAT file format (BLUE LED) ready to be analyzed.

Fig 2.16, Fig 2.17 shows the EASE module; Fig 2.18 shows the Docking Station (Raspberry Pi with the docking station header for easy connection to the connector); Fig 2.19, Fig 2.20 shows the connection configuration while testing occipital lobe eyes closed data; Fig 2.21 shows the modifications to NeuroSky's Mindwave headset to use the TGAM module directly.

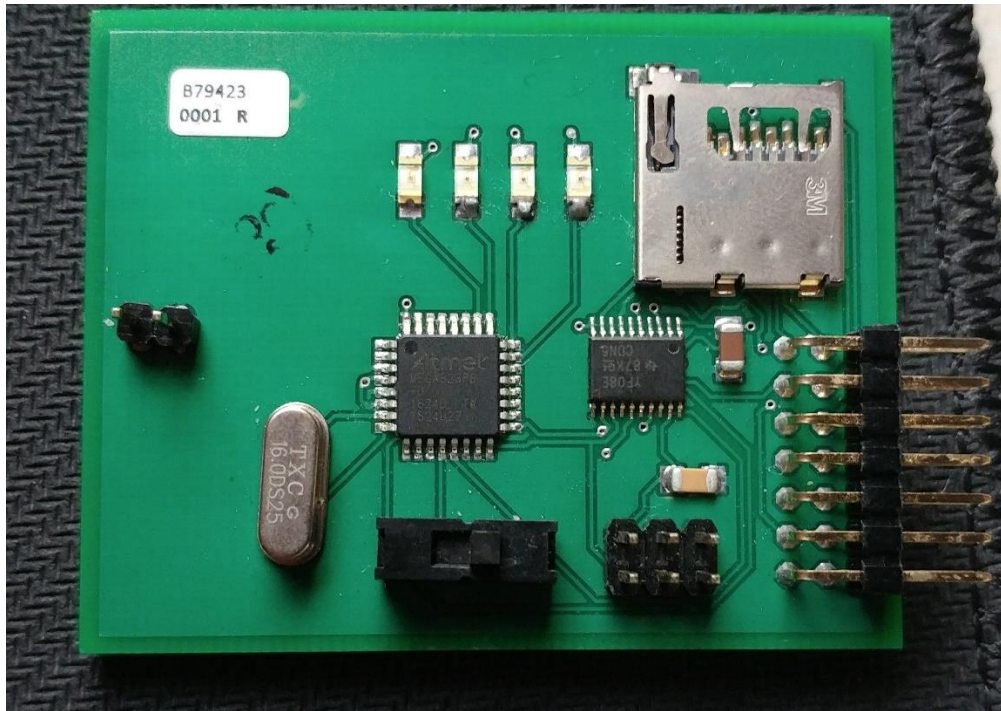


Figure 2.16 EASE module front view.

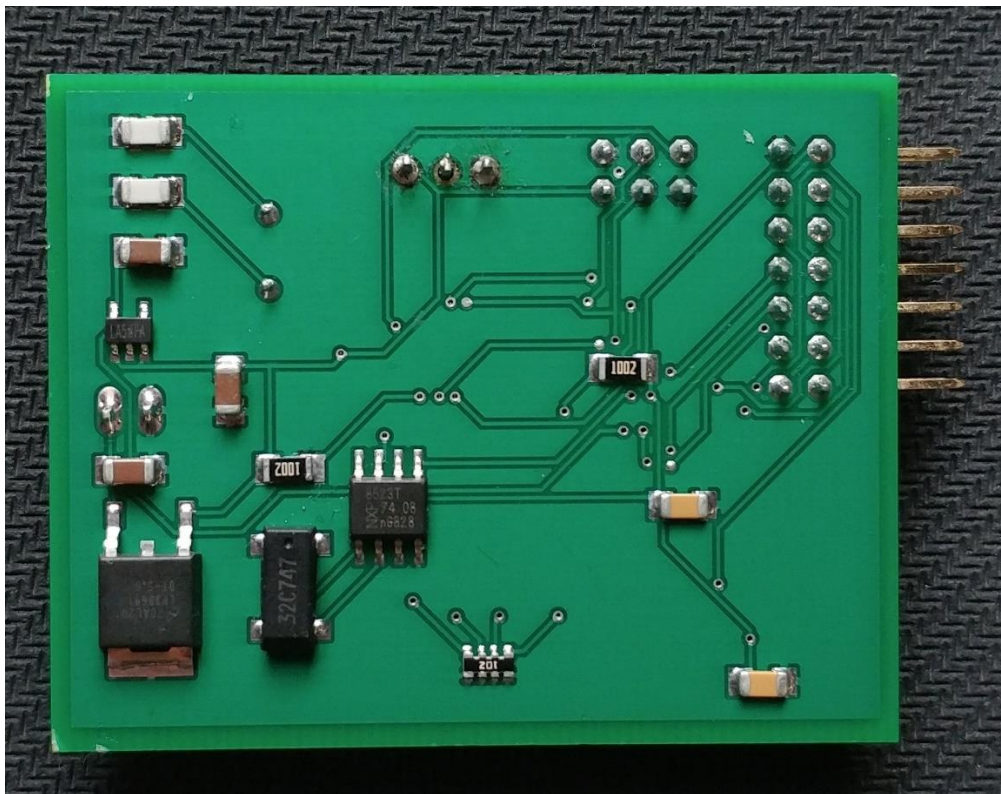


Figure 2.17 EASE module back view.

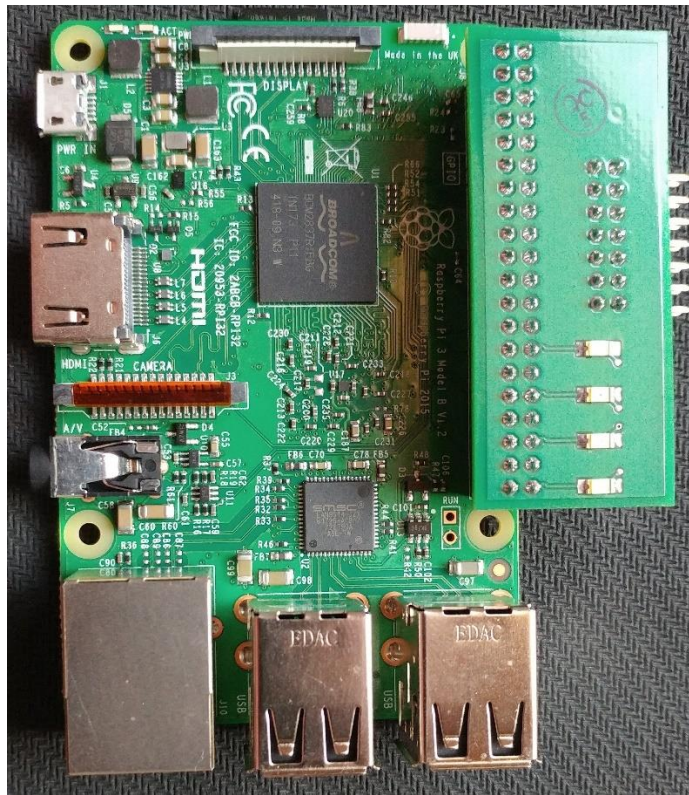


Figure 2.18 Docking Station with the 14-pin header.



Figure 2.19 EEG wet, golden cup electrode connected to occipital lobe.

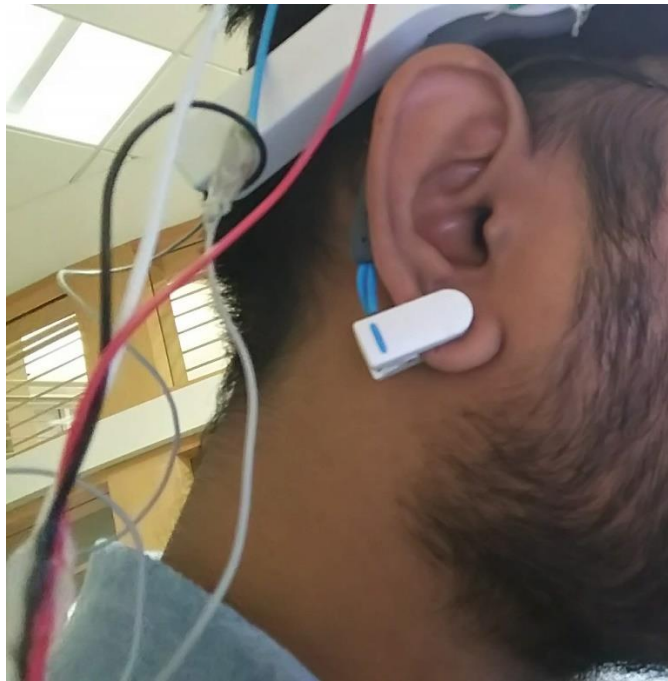


Figure 2.20 Reference and ground electrodes connected to the ear with a clip.

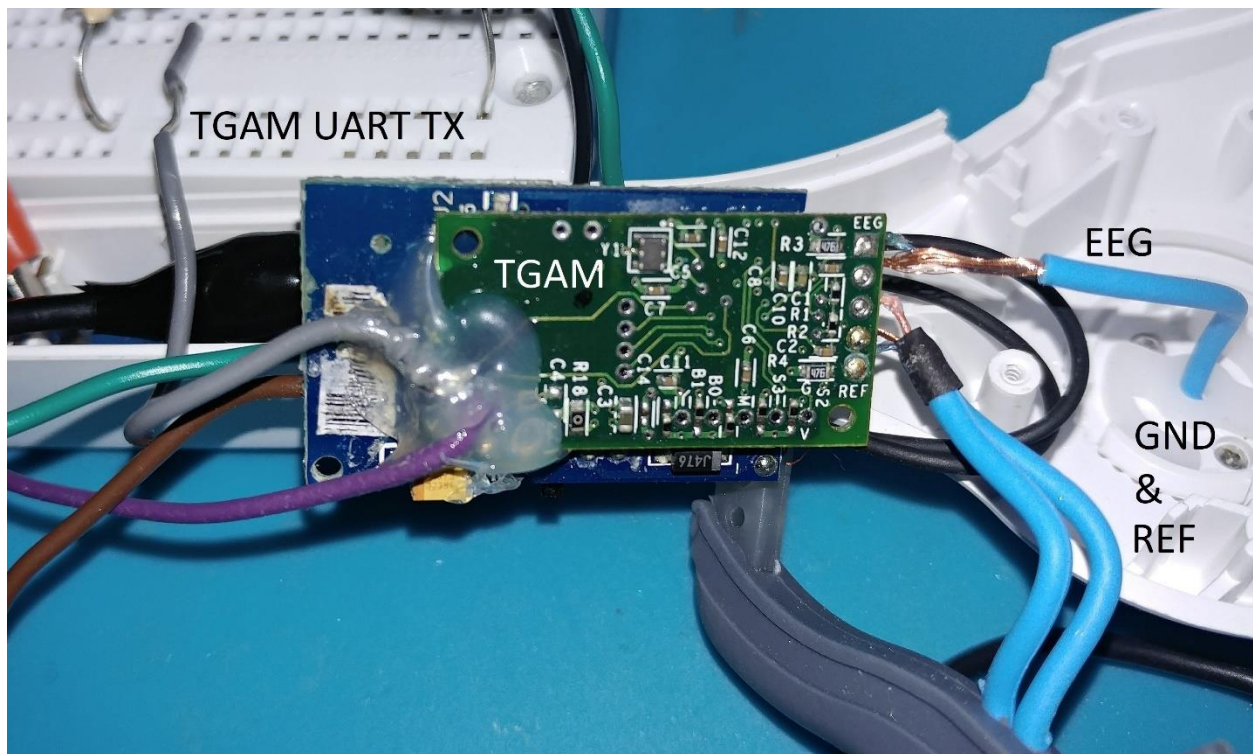


Figure 2.21 Modified NeuroSky's Mindwave headset to use the TGAM directly.

Testing the built system initially for errors found in the system's recorded output. Fig 2.22 and Fig 2.23 show result for some of the input signal applied. Fig 2.22 contains input sinusoidal signal with frequency 7Hz applied at $\pm 1mV_p$ and Fig 2.23 contains input sinusoidal signal with different frequencies at $\pm 500uV_p$. Other than this the system's error response was tested by applying different frequencies 2Hz, 4Hz, 6Hz, 8Hz, ..., 16Hz, 18Hz, 20Hz at $\pm 10uV_p$, $\pm 50uV_p$ and $\pm 100uV_p$ and taking the maximum error found for those signals as shown in Fig 2.24, 2.25 and 2.26 respectively.

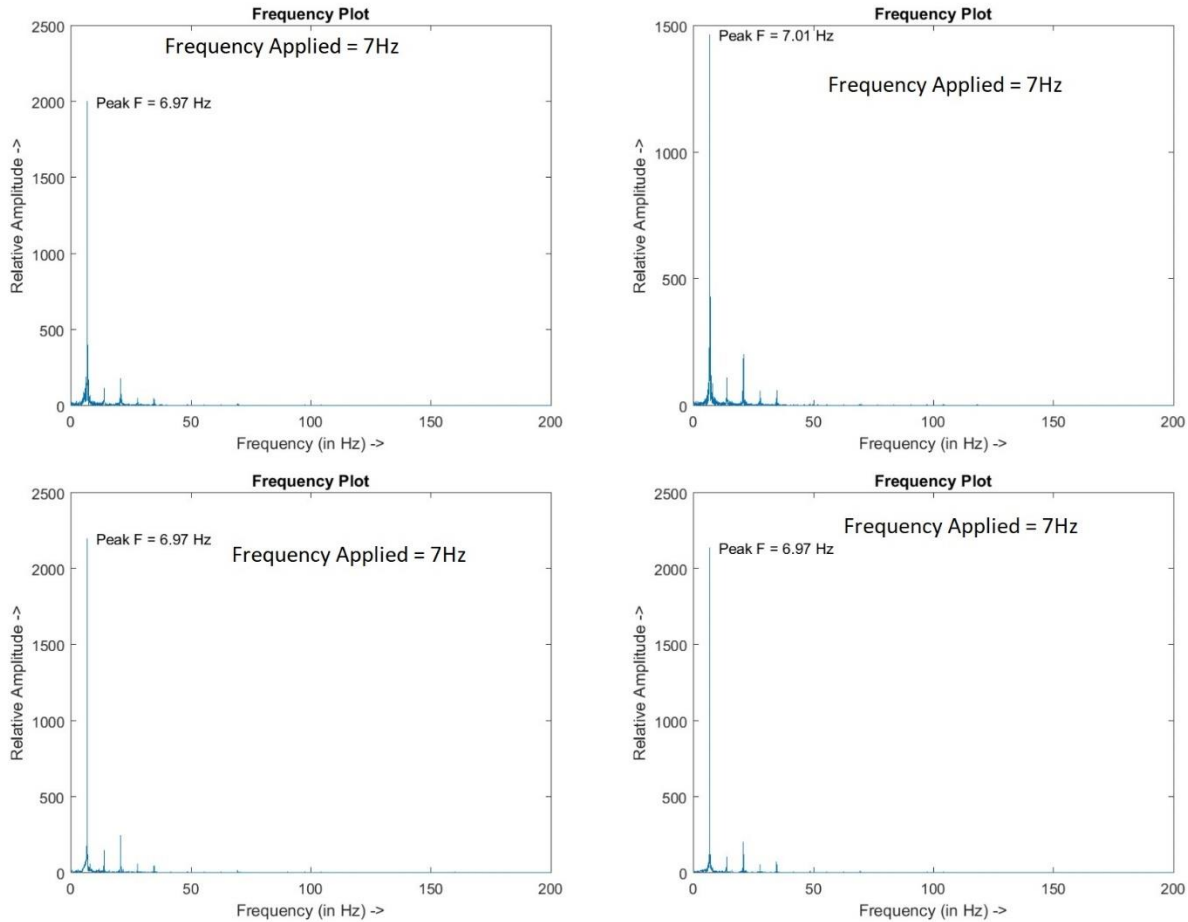


Figure 2.22 EEG system's response with applied signal at 7Hz frequency at $\pm 1mV_p$.

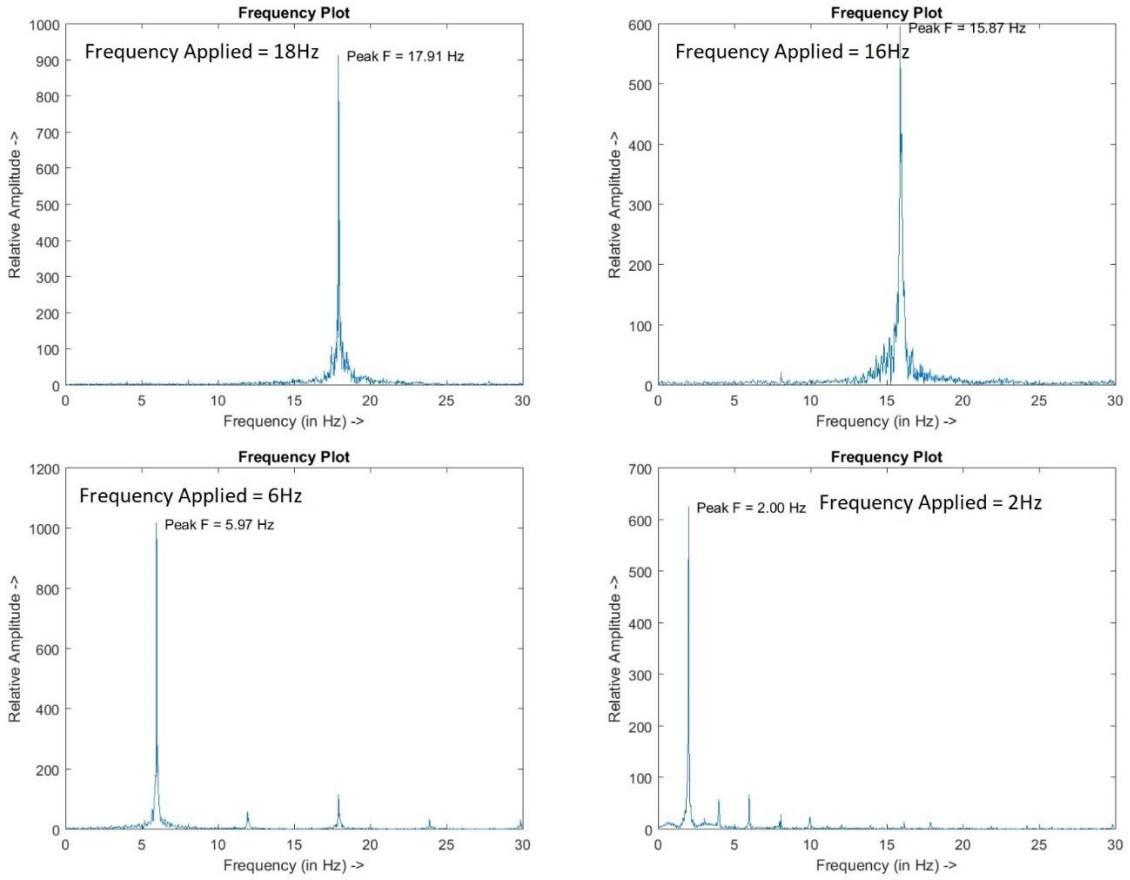


Figure 2.23 EEG system's response with applied signal at different frequencies at $\pm 500\mu V_p$.

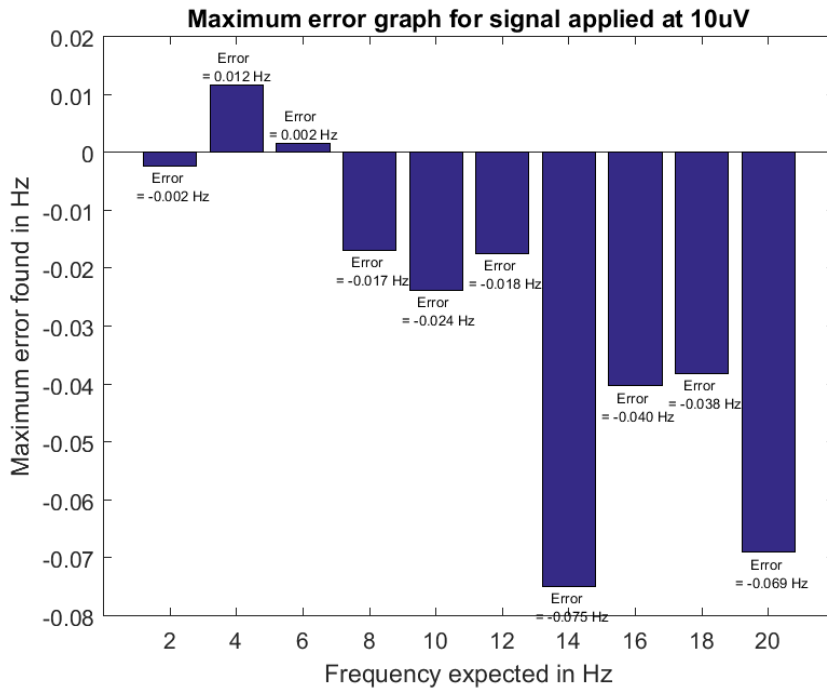


Figure 2.24 EEG system's maximum error response for signals applied at $\pm 10\mu V_p$.

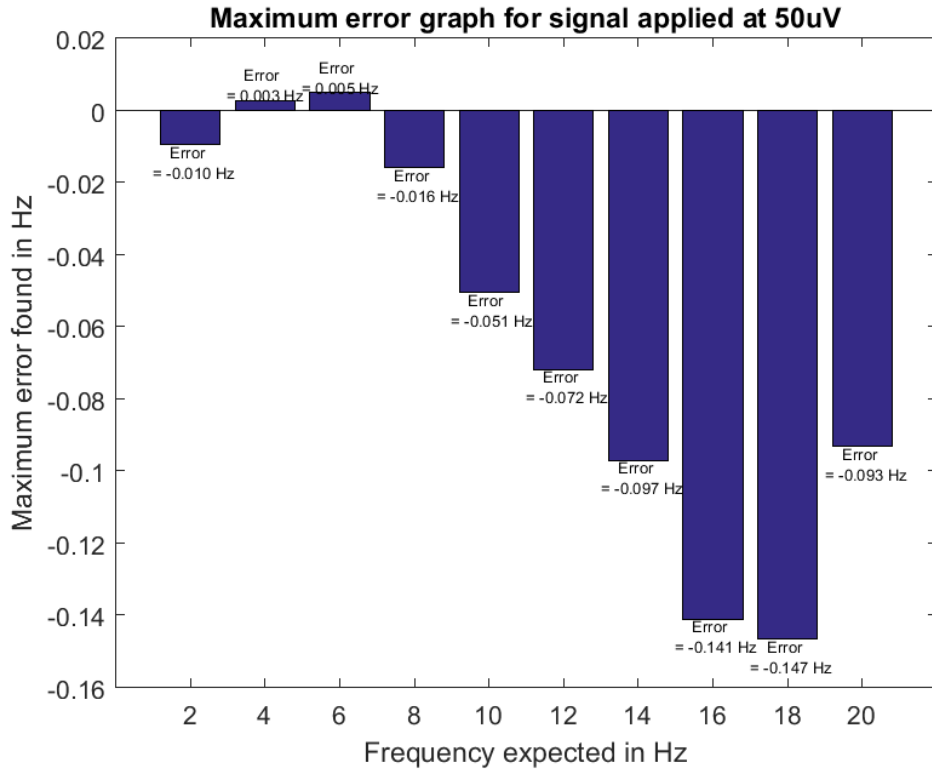


Figure 2.25 EEG system's maximum error response for signals applied at $\pm 50\mu\text{V}_p$.

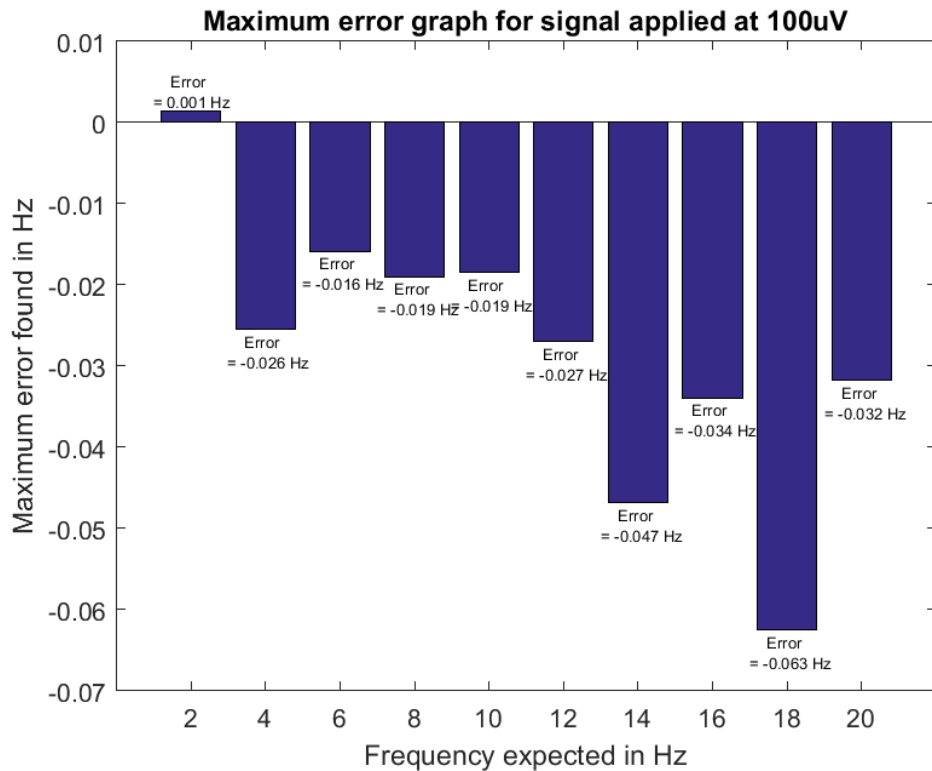


Figure 2.26 EEG system's maximum error response for signals applied at $\pm 100\mu\text{V}_p$.

From the maximum error responses found, it appears that the maximum error percentages found at $\pm 10\mu V_p$ was 0.536% at 14Hz, at $\pm 50\mu V_p$ was 0.881% at 16Hz, and at $\pm 100\mu V_p$ was 0.65% at 4Hz. From this it appears that there is not a direct relation between the applied voltage and the maximum error found in data, so it can also be attributed to other causes. So, typical maximum error percentage comes out to be average of 0.536%,0.881%,0.65% which is 0.689%.

Some of the responses for eyes closed occipital lobe data without any kind of external filtering are shown in Fig 2.27.

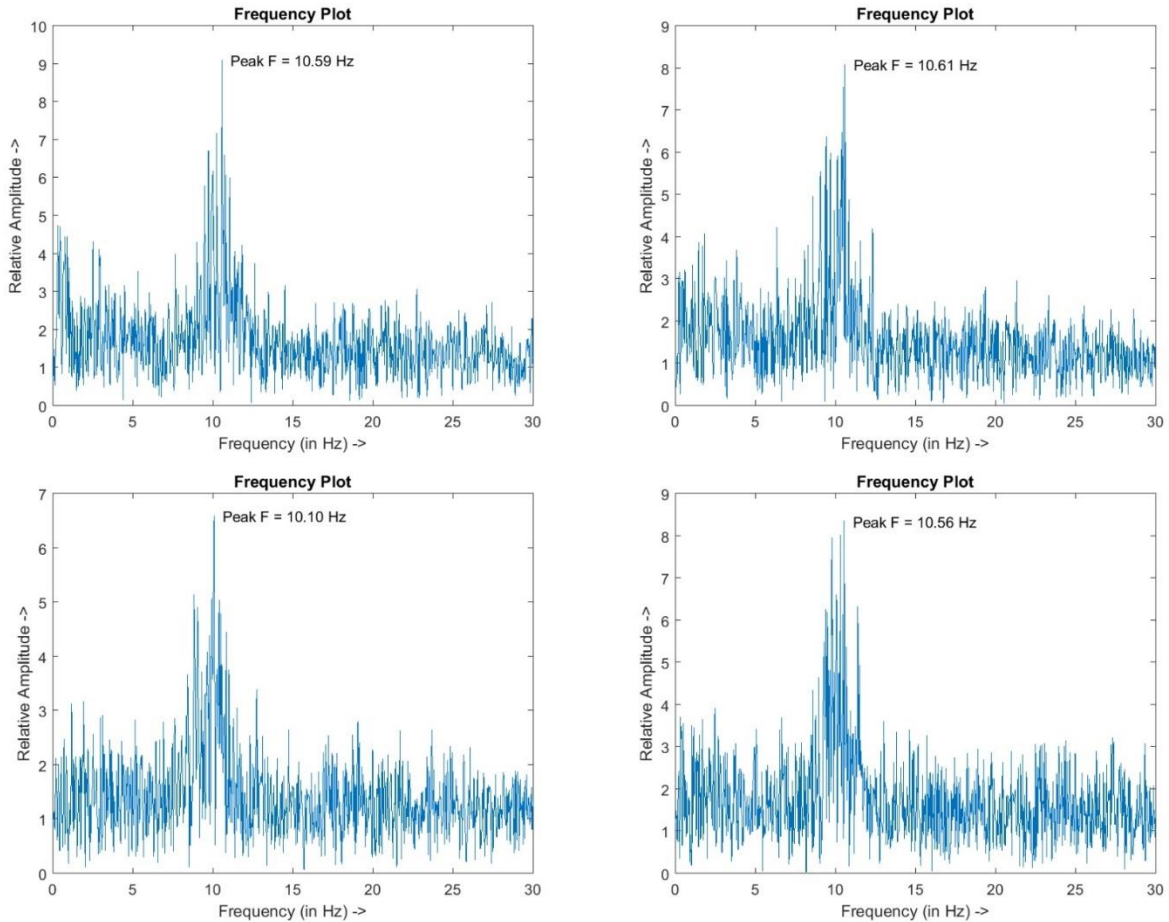


Figure 2.27 EEG system's response for unfiltered eyes closed occipital lobe data.

The system was also tested for power consumption over time. The setup conditions were that the device was connected to the headset which was receiving random noise signal and the data record was continuously going on, and the supplied supply voltage to the device was at 5.05V which is more than the minimum limit allowed by the 5V regulator. A source meter was used to calculate and record the current consumption over time for the device at the frequency of 28 samples per second. Some part of the response found from the system has been plotted in Fig 2.28 for around 435 seconds of data. By looking at the data it doesn't appear to have any correlation in the current consumed over time, except for few peaks which were coming after some period of time. Other than that, the base power consumption appears to be around 42mA and there are spikes going up to maximum 88.3mA which could be attributed to changing the banks inside the SD card while writing to it. After a certain amount of time the average current consumption didn't appear to be changing that much, and came out to be 45.869mA. So, we can say that the device consumes 45.869mAH battery on average.

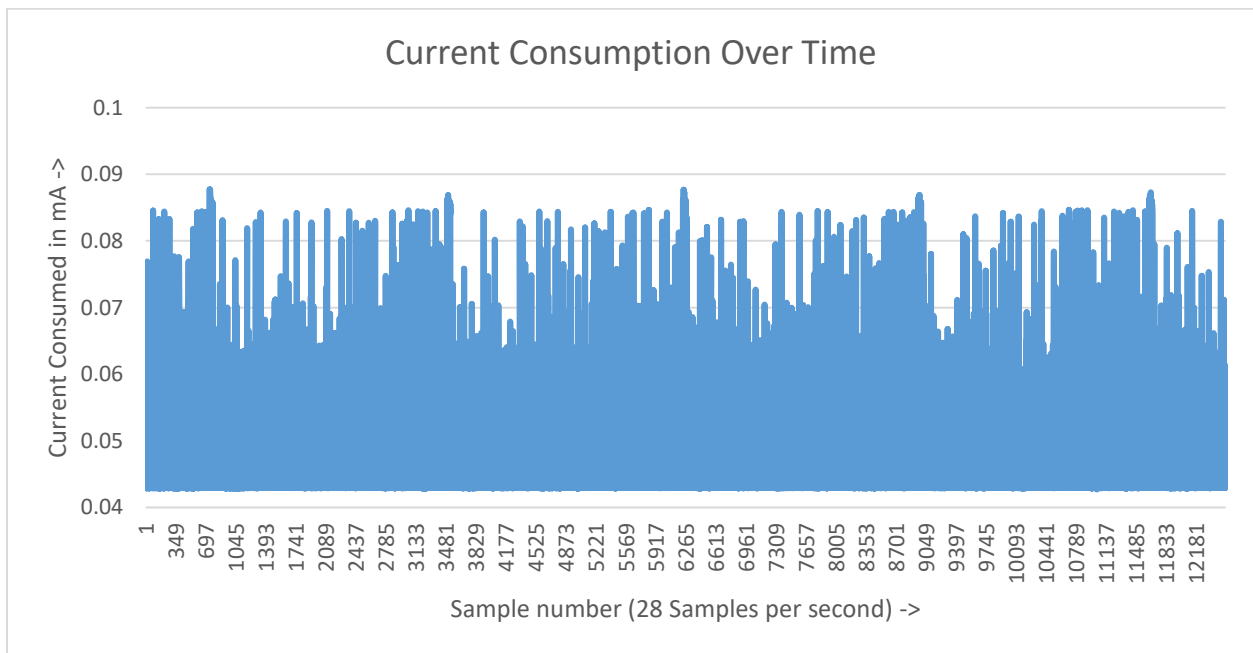


Figure 2.28 EEG system's current consumption over time.

So, according to the power calculations, the device can be run continuously for 24 hours and it will consume around $24 \times 45.869 \text{mAH} = 1100 \text{mAH}$ which is great considering the device runs continuously and there are lot of battery choices such as li-po and li-ion batteries which are available in even bigger sizes with small package size. Another analysis was run on the average number of samples captured by the recorded files over around 50 files, and it was found to be about 15350. Since this was for 30 second data, the average should come close to $30 \times 512 = 15360$. So, the number of sample loss comes out to be really low, the remaining samples which are lost could be due to wrong transmission, or wrong packet entry point at the start and end of capture.

So, the system specifications found after device testing for the designed system:

1. Data record intervals: 30 Sec set by default for each file (Can be increased or decreased as required).
2. Docking Station supply: 5V input from micro-USB power supply.
3. Minimum supply to EASE module: 5.045V.
4. TGAM input voltage range: $\pm 1 \text{mVp}$.
5. TGAM input frequency characteristics observed: High pass filter around 1Hz (Inbuilt, non-configurable), Notch filter at 60Hz (Inbuilt, externally configurable to 50Hz), low pass filter around 100Hz (Inbuilt, non-configurable).
6. TGAM sampling rate: 512 samples per second.
7. TGAM input voltage resolution: 0.488uV.
8. TGAM Serial output baud rate: 57600 BPS.
9. TGAM Operating voltage: 2.97-3.63V.
10. TGAM Sampling inputs available: EEG Electrode (Shielding available), Reference Electrode (Shielding available), Ground Electrode.

11. Typical maximum error in the recorded samples: 0.689%
12. Average current draw: 45.869mA at 5.05V supply.
13. Minimum SD card size required for continuous 24hr recording: 357 MB (8 byte for each sample, 512 samples per second, 3600 seconds per hour and one 36 byte packet per second)

CHAPTER 3

NON-INVASIVE BRAIN STIMULATION DEVICE

DEVELOPMENT

3.1. Device Components

The work presented in this part of the thesis is an extension of previous work done by the Pulvinar Neuro company which is a spin-off of the Frohlich lab on a non-invasive brain stimulation platform called XCSITE-100 [STIM1]. Fig 3.1 shows XCSITE 100 which is a research grade tDCS/tACS device that enables high-quality double-blind tDCS/tACS studies. The main goal here was to modify the present XCSITE 100 system to be able to send custom waveform based on researcher's input and to present a hardware trigger to be able to start the stimulation based on external data processing equipment without needing a manual input so that the system can be independently run with a simple pin for triggering the stimulation. The trigger can then be used with something like the TGAM chip to parse the incoming EEG bands or the EEG waveform, continually process the data and start the stimulation as required on a specific EEG behavior and to stop it as required.

The system contains the stimulation hardware and the application required to control it. The application is a simple front end for sending configuration commands to the stimulation hardware. The stimulation hardware platform consists of a circuit board with the microcontroller PIC18F handling most of the peripherals on board. The task for the microcontroller is to set up the stimulation parameters according to what the user wants, then use those parameters to give a

current output since this is current stimulation, and monitor the stimulation parameters like output voltage and current to detect the correct electrode placement and also to provide overvoltage or current protection. At the same time, this device is user friendly enough to require no external intervention other than the parameter setup to make it work properly. The XCSITE 100 also contains sham stimulation procedure which is present to take care of placebo effect and when it is set the device only delivers a ramp up to certain current and then ramps down from it and runs till the end of the stimulation time set which helps to avoid placebo effect.



Figure 3.1 XCSITE 100, by Pulvinar Neuro.

The other part of the system is the application and monitoring platform which takes care of the user interfacing. The selected method of communication between the application and the stimulation hardware is Bluetooth, so the application makes use of Bluetooth on application device to communicate with the stimulation hardware. At the same time, using some of the commands

the application logs the stimulation data returned by the stimulation hardware so it can be reviewed afterwards for correctness of the applied procedure. Here we are dealing with only the stimulation hardware area where the changes will be made.

3.2. Device Architecture

The device component blocks and their interconnections are shown in Fig 3.2. The description for the connected block in the figure and some information about them are given below:

1. The XCSITE 100 device is powered by a single 9V battery which goes primarily to the switching power supply units.
2. There is a +36V and -36V supply output for the voltage to current converter which delivers the constant current output. The reason for this is that since this is current stimulation, the voltage needs to match according to the impedance between the output terminals, so if we were stimulating at let's say 1mA and the impedance at the output terminals was 1k Ohm that means the output voltage would be 1V. So, there could be a condition of high impedance at the output which would require us to provide high voltage under monitored limits to continue current stimulation which is why the high voltage supply is required, but the current is still limited to 2mA overall. The supply here is generated using an isolated DC-DC regulated switching converter [STIM2] with efficiency up to 80%, by connecting multiple of them together since single one of them provides a $\pm 12V$ output for a wide range of input supplied. The ground signal to this supply is connected through a solid-state relay whose enable signal comes from the microcontroller, this is present to provide the microcontroller a control on the output supply so that in case of an over voltage or current condition the supply can be killed.

3. There is another $\pm 5V$ switching power supply from the 9V battery input with 83% efficiency similar to the $\pm 12V$ power supply used before. The +5V output from this supply goes to two Low Drop-Out regulators at 3.3V, one for the microcontroller and another one for the DAC chip.
4. A 3.3V switching power supply is also present on the board which supplies power to the Bluetooth module. The PIC is clocked by a 20MHz crystal oscillator whereas the DAC is clocked by a 32.768KHz low power clock oscillator which gets its clock from a 32.768KHz crystal oscillator. The DAC clock is important to be noted since it is the main clock that is supplied to the output generation circuit while reading the data from the SRAM of the DAC.
5. There is an ICSP (In-Circuit Serial Programming) [HARD3] header which enables the user/researcher to deliver their firmware to the stimulation hardware's microcontroller. The ICSP device uses SPI communication and pins to reprogram the microcontroller. The PIC has a push button switch connected to it for starting and ending the stimulation. The switch also has red and blue LEDs in it for signaling. So, the PIC outputs the GPIO signals for the LED control and the switch input goes to the PIC.
6. The PIC communicates via UART with the Bluetooth module which connects to the external world devices for getting commands for the controller and sending responses wirelessly. For communicating with the DAC chip AD9106 [STIM3] for setting the parameters of stimulation or the value to be output, the controller uses SPI protocol.
7. After the PIC sets up the DAC and sets the output, the output from the DAC is double ended. This then goes into single ended low voltage converter with an OPAMP. Then, finally it is sent to the voltage to current output generator OPAMP which is supplied by the

$\pm 36V$ switching PSU. The output from this circuit is again controlled with a solid-state relay which works on hardware output check for signal to be in set limits.

8. The output is sent to the measurement conditioning stage where the current and voltages are separated for calculation of how much current and voltage is being output currently. This then goes to the current and voltage sense formatting to be sent to the ADC of the PIC. The current is simply separated by passing it through a resistor generating a voltage to be sensed through an OPAMP so that there is no effect of sensing at the real output, which is then formatted in the 3.3V range of PIC ADC by passing through another OPAMP with the 3.3V supply. The voltage is directly measured from the output pins, but it is also passed through an OPAMP to again cause no effect to the real output, then passed through a sense OPAMP working at 3.3V range to format the voltage in PIC ADC range.

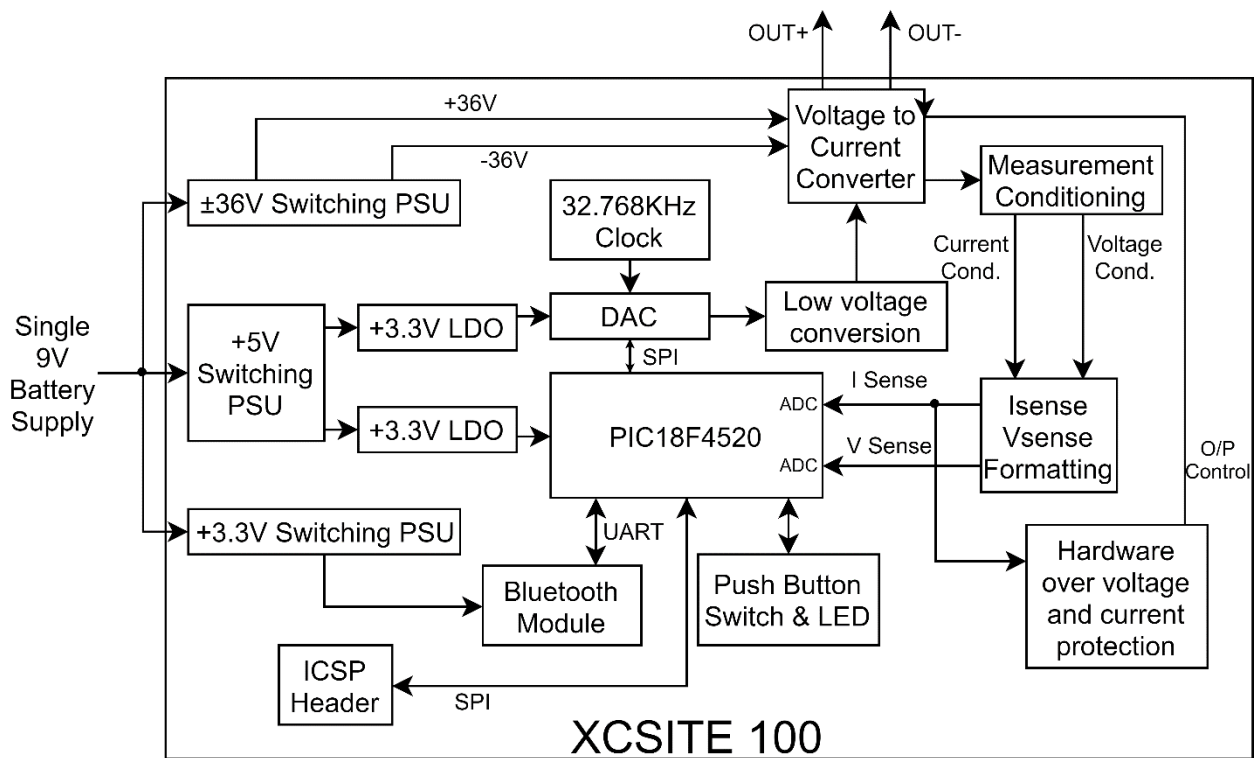


Figure 3.2 XCSITE 100 System Architecture.

9. So, after we get the current sensing and voltage sensing outputs ready, they are sent to the PIC ADC, and the current sensing output is also sent to the hardware over and under current protection circuits (for positive maximum and negative maximum). The hardware over current protection circuits take the output of sensed current, pass it through the comparators for checking if the current is more than positive maximum and if the current is more than negative maximum and then send it through an OR gate to see if there was a problem with either one of the parameters and then send it through a latch made of logic gates to generate a latched signal based on if there was a violation at the output, and the latch output is then sent to the relay controlling the output of the device. Now, the sensed values sent to the controller's ADC will be sensed till that time and the program can detect that an error has happened with either the voltage or current. So, the program will stop the execution of output and clear the latch, and show that there is an error state reached. Another use for the sensed current and voltage values is for measuring the impedance at the output terminals. This impedance is measured in the program and logged with the Bluetooth app present. If there is any problem with impedance at output (being more than max allowed impedance), the stimulation stops.

These were the hardware and software communication and interfacing blocks present in the XCSITE hardware. Now, for the software architecture and control flow the Fig 3.3 gives a higher-level view of what happens in the software. Most of the modified parts of the firmware have been explained after the description of the system flow and interfacing. The XCSITE 100 works in 2 parts, one is command decoding and another is a state machine. The command decode works on UART, for which Bluetooth has been used here to take commands wirelessly, and based

on the commands the initial triggering of the state machine occurs, and also the commands can be used to break or block the state machine. But if the state machine is simply started normally, it works till the stimulation time set and then goes back to default state. So, as it can be seen from the flow chart, when the PIC boots up it begins by setting up all the required and default parameters, GPIO and ADC setup, DAC setup and other peripheral setup. After that, in short, the main flow works like a tiny state machine where the device checks for any command waiting then moves on to change or set the parameters for the bigger state machine inside and then goes back to top. So, the controller first checks the UART buffer if there is anything received, if it is and it's a command, then the command is matched to the command length buffer which tells the controller

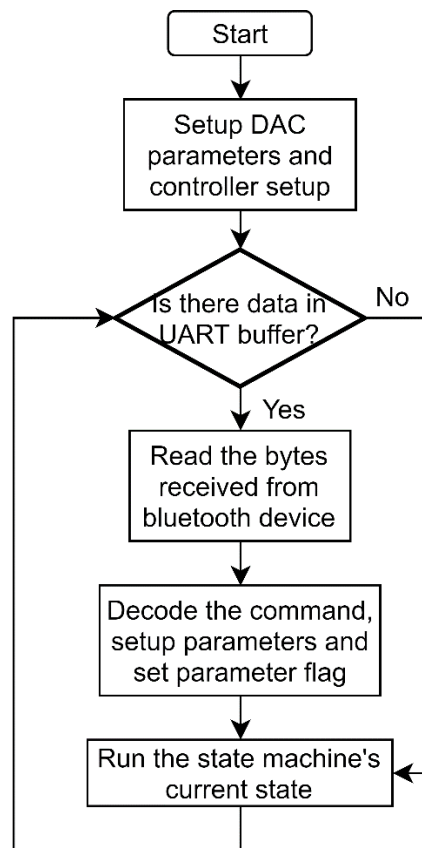


Figure 3.3 XCSITE 100 software flow.

how many total bytes are there in the command. So, after read that the controller then waits for the full command bytes to come and reads the complete command bytes. Then the command is searched and matched and the procedure related to it is executed. If the command sets a parameter, the parameter flag is also set so that before starting the stimulation i.e. triggering the state machine the controller can make sure that all required parameters have been correctly set. So, after the command procedure is finished then the state machine is looked up and the current state is run. If nothing is happening the state machine will be in STIM_OFF state at the beginning which simply keeps all the parameters to default state. So, after the state finishes executing the control goes back to the top where the controller again looks for incoming command. Now, if there is no incoming command the controller simply skips the command lookup and only the state machine is run and then the program goes back to the start to check for any available bytes in the buffer.

Before going to the main state machine, a little description about the interrupts being used by the system. There is a button interrupt for the button connected to the controller which runs at the positive edge of button press, and it is simply used to set the button press flag to 1 so that whenever the state machine checks for the button press it can simply access this variable and clear it so next button press can be captured. There is an over current interrupt which runs as soon as there is over current at the output, this comes from the current sensing and auto cut-off circuitry for handling over current protection. Whenever this interrupt comes, if AC stimulation is going on the stimulation power (+36V, -36V) is turned off, and the DAC trigger to output is turned off and a high current flag is set. There is a UART interrupt, which records data to a circular buffer. Other than these the program here is making use of a timer which ticks at a frequency of 400Hz. Each time this timer overflows the ISR runs. This ISR re-initializes the timer after the overflow, sets the ADC channel to the pins where the voltage and current sensing pins are connected and samples

them and stores the value to raw value variables, and sets a flag to indicate samples have been taken. The ISR also keeps track of the stimulation time and sends the state machine to power down mode if the time exceeds total stimulation time. There is a LED counter which is used for flashing LED. Based on the ramping state being either DC or AC or custom stimulation the ISR increments or decrements the intermediate value which has to be set for the DAC during the ramp functions to vary the output current level.

Coming to the main state machine, the Fig 3.4 shows the states of the state machine and their interconnections. This state machine takes care of setting the right DAC parameters, delivering the output as required depending on the settings set by the commands sent to the device. The tasks performed by the state machine states and their transitions are (Each of the commands and state machine states mentioned here have their value set beforehand in the source code and in the command sending device):

1. **STIM_OFF**: The state machine primarily stays in this state when there's no task to perform. The output circuitry is turned off by turning the output of the DAC off using the trigger pin. This state takes care of the LED on the push button switch based on whether or not the Bluetooth device is connected or not. When the Bluetooth device is connected, the state turns the LED to BLUE and when the device is searching for the Bluetooth device, the LED turns to RED. If the device receives a **READY_TO_START** command from the connected device, and if the password sent with it is right the next state turns to 'WAITING', so the 'A' transition in the state diagram takes place next time the state machine runs. If the command received is to do impedance check, the next state also changes to 'WAITING' state but the impedance check flag is asserted and this task doesn't need the **READY_TO_START** command or the password.

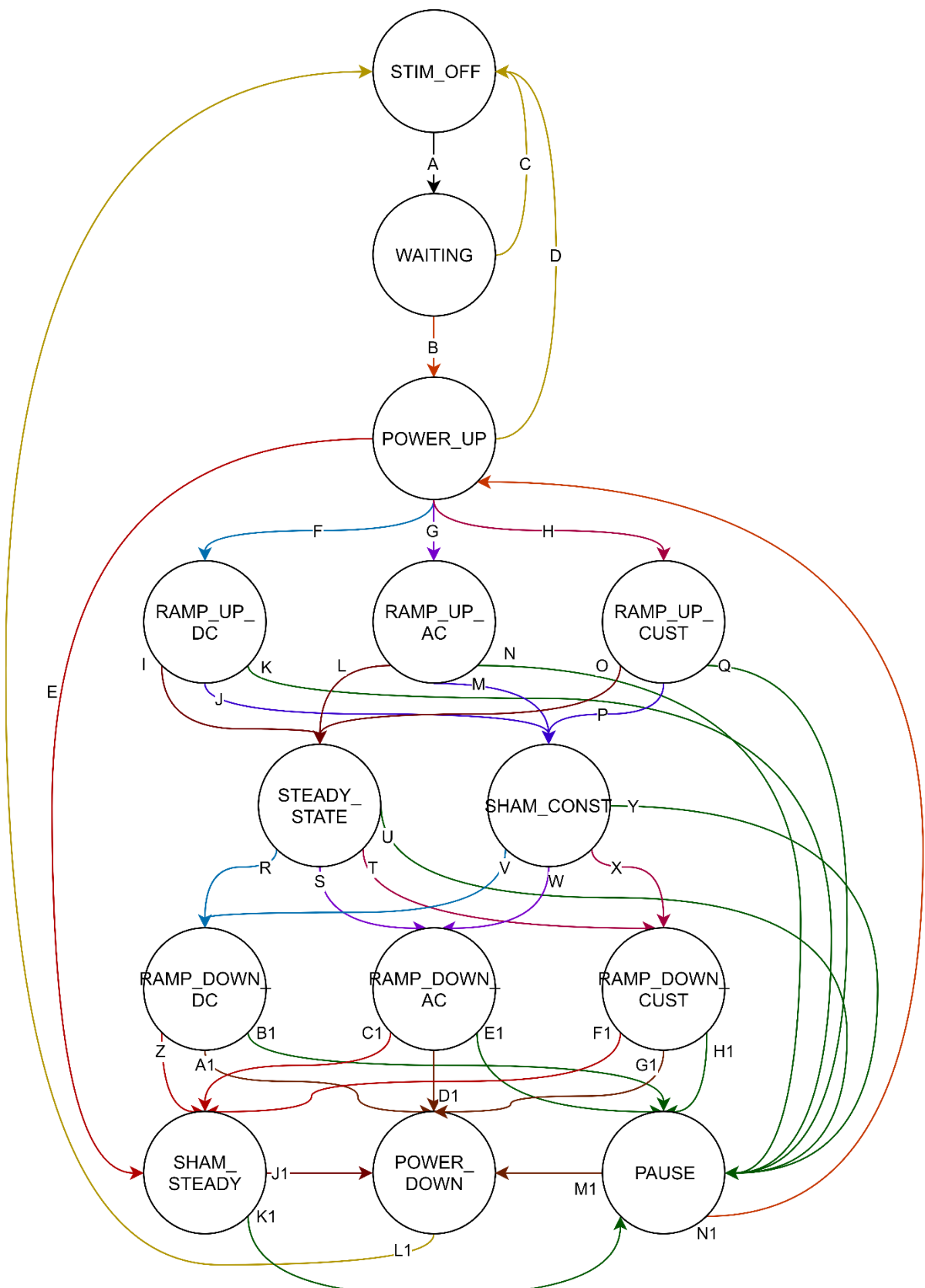


Figure 3.4 XCSITE 100 main state machine.

2. WAITING: This state mostly takes care of getting user input for going forward, or simply a trigger from another system from the button pin. So, here if the Bluetooth device is verified to be connected, if the command sent is 'CANCEL', the stimulation is cancelled and state goes back to 'STIM_OFF' (transition 'C'). If the button is pressed or a trigger is sent, the device checks if all the parameters have been set before and if they are then the next state changes to 'POWER_UP' (transition 'B'). Otherwise, if the impedance check flag is high no command is needed, transition 'B' takes place normally. If the Bluetooth device isn't connected, the state performs transition 'C'.
3. POWER_UP: This state takes care of enabling the hardware peripherals and setting some flags and values. At the start the DAC output is enabled, then the signal to the stimulator switching power supply of $\pm 36V$ is enabled, the auto cut-off circuit's latch is reset, output relay for DAC channel 1 is enabled, the LED on the push button changed to constant RED color. If AC stimulation is selected the pattern status register is used to give an output enable trigger to the DAC, but the digital gain is kept 0 beforehand. The time counter variable for the program duration tracking is reset, the Bluetooth device is sent stimulating command/indication. Instead if impedance check is turned on, next state changes to 'SHAM_STEADY' (transition 'E'), with a predefined small stimulation time since it is just an impedance check procedure, the dc stimulation flag is enabled and the output current is setup to be a small current. Otherwise, if the impedance check isn't turned on, the next state is selected based on the DC stimulation and custom stimulation flag to be either 'RAMP_UP_DC' (transition 'F') or 'RAMP_UP_AC' (transition 'G') or 'RAMP_UP_CUST' (transition 'H'). Other miscellaneous flags used are cleared to reset value before proceeding.

4. RAMP_UP_DC: This state takes care of the output signal ramp up in case DC stimulation is selected. The intermediate DC value is calculated based on the final DC output value set by the commands and the time to ramp up beforehand when the command for setting ramp up time is sent by the external device. Now, the timer which is used by the microcontroller to keep track of parameters, variables, current and voltage ADC values ticks at 400Hz. So, whenever the timer ticks the intermediate DC value is increased by a certain value so that it reaches the set final value in the given ramp up time. For example, if the final value supplied was 2mA and the time to ramp up was 2 seconds, that means we reach 0 to 2mA in $2 \times 400 = 800$ ticks, that means we have to increment the output current by $2\text{mA}/800 = 2.5\mu\text{A}$ per tick so that in 2 seconds we can reach the final intended value. The value to increment is taken care by the timer ISR, the state here simply checks if the value has reached the final value or not and pushes the intermediate calculated value to the DAC as a constant output value. So, when the ramp up is finished, i.e. when we reach the DC final value, the ramp done signals are set and the next state is changed to 'STEADY_STATE' (transition 'I') if the sham stimulation is disabled, and to 'SHAM_CONST' (transition 'J') state if sham stimulation is enabled. If the button is pressed during the execution of ramp up, the next state is set to 'STEADY_STATE' but the stim cancelled flag is asserted, so whenever the next state changes the stimulation will stop after that flag is checked. If ADC samples have been calculated in the timer ISR, their flag will be set. This state checks if the flag is set, and when the ADC samples for current sensing and voltage sensing is taken, the impedance and current check is performed. Failing the impedance check means the device isn't connected properly and failing current check means more than the set limit of current was sent to the output and this means the device will go to 'PAUSE' (transition

‘K’) state. If the impedance and current was found to be under limits the sample values for impedance and current are written to the external device via Bluetooth and the normal operation ensues.

5. RAMP_UP_AC: This state is similar in working as the DC ramp up state. The biggest difference here is that the output control employed to the DAC is using digital gain control. The output from the DAC is given by the sine wave frequency synthesizer and the amplitude set by sending the AC intermediate value to the digital gain of the DAC. So, similar to the DC ramp up, this function checks for the intermediate value if it reached the final value, if it did it sets the ramp done flag. So, when the ramp up is finished, the next state changes to ‘STEADY_STATE’ (transition ‘L’) if sham stimulation is turned off and to ‘SHAM_CONST’ (transition ‘M’) if the sham stimulation is turned on. If the button was pressed, again the state changes to steady state but the stimulation cancelled flag is set. If the voltage and current samples were present the impedance and current measurement is done, failing which the state goes to ‘PAUSE’ (transition ‘N’) and continuous normally if the impedance and current measurements were under limit and sends it to the Bluetooth external device to be logged.
6. RAMP_UP_CUST: This state is also similar to the ramp up DC and ramp up AC state. Similar to the ramp up AC state, the output voltage is controlled by the digital gain set for the DAC. The initialization of this state is a bit different, since the initialization for the AC is done when the AC parameters are set, but for the custom stimulation the initialization of the DAC is done before ramp up starts in this state (the storage procedure of custom waveform is done when the samples are supplied so that is done beforehand). So, to initialize the DAC, the wave configuration is changed to use the data stored in SRAM. The

initial gain is set to 0 so that the value can ramp up. The pattern period is set based on the range of values supplied so that all the values are covered in the output. The start address and the stop address are set based on the range supplied by the commands. The pattern type is set to repeat forever based on manual stop or with the physical pin. The start delay is set to 0, the time-base register which controls the start delay timing and output sample timing is set to the slowest value possible so that the signal can be stretched to maximum length for now. According to the delay calculations, the time-base register sets 16 clock cycles per sample output as maximum slowed output rate. With the DAC clock of 32.768KHz, that means we can output 2048 samples per seconds. Since the DAC SRAM has a space of 4096, the output signal can be spread out to maximum of 2 seconds. After the DAC setup the pattern generation is started again. Other than that, the intermediate value increment is same as AC or DC states, just that the final value is different, but it is again delivered to the DAC digital gain. After the ramp up is finished the state changes to 'SHAM_CONST' (transition 'P') if sham stimulation is enabled or 'STEADY_STATE' (transition 'O') if the real stimulation is going on. The device again checks for the impedance and current measurements to be within limits and sends them to the external device. If the values are not within limits the execution pauses when the state machine goes to 'PAUSE' (transition 'Q') state.

7. STEADY_STATE: This state simply keeps track of time to keep the output steady and keeps checking for the correct operation by the impedance and current check procedure. If the button is pressed, the stimulation is cancelled and the timer resets down to the remaining time for ramp down. So, if the time is up and the ramp down is remaining, the state changes based on the DC stim flag and custom stim flag to 'RAMP_DOWN_DC' (transition 'R')

or 'RAMP_DOWN_AC' (transition 'S') or 'RAMP_DOWN_CUST' (transition 'T'). If there is any impedance or current over the limit detected the next state changes to 'PAUSE' (transition 'U') state.

8. SHAM_CONST: This state is again nearly completely similar to the STEADY_STATE. The difference is while tracking the time to run this state, the time is compared with the sham stimulation time instead of the complete stimulation time. Also, if impedance check was initiated due to which the program might come to this state, the ramp down state is skipped, but if impedance check wasn't set then the execution flows like STEADY_STATE with sham time. So, this function again moves to the next state based on DC stim flag or custom stim flag to 'RAMP_DOWN_DC' (transition 'V') or 'RAMP_DOWN_AC' (transition 'W') or 'RAMP_DOWN_CUST' (transition 'X'). Similarly, if the impedance and current results are not in allowed limits the state goes to 'PAUSE' (transition 'Y') state.
9. RAMP_DOWN_DC: The ramp down state is the exact opposite of the ramp up state in terms of intermediate DC value. The value is still updated by the timer ISR, but the updated value is decremented from maximum value to 0. The updated value is again pushed by the constant value register. If the ramp down reached the end, the next state changes to 'POWER_DOWN' (transition 'A1') or if the button was kept pressed in case the stimulation was sham stimulation or if the button was pressed once in case it was true stimulation, but if the sham stimulation was present and the stimulation was cancelled by pressing the button once or if true stimulation was present and the stimulation was cancelled before coming to this state, the next state changes to 'SHAM_STEADY' (transition 'Z'). The impedance and current are again measured and evaluated and if found

to be out of limits the next state changes to 'PAUSE' (transition 'B1'). In case impedance check flag was on, the next state simply changes to 'SHAM_STEADY' without ramping down.

10. RAMP_DOWN_AC: This state is similar to ramp down DC, with the intermediate digital gain being controlled by the timer ISR decrementing it from maximum value to 0. When the digital gain reaches 0, the ramp down finishes and the next state changes to 'POWER_DOWN' (transition 'D1'), if the button is kept pressed and the stimulation was sham the next state also changes to power down, and if the button was pressed any way if the stimulation was true the next state changes to power down. If the stimulation was sham and button was pressed once, or stimulation was true but it got cancelled before the next state changes to 'SHAM_STEADY' (transition 'C1'). Then again, the impedance and current results are checked to be within limit, pushing them to the Bluetooth device. If they are found to be outside the set limits the next state changes to 'PAUSE' (transition 'E1') state. The wave configuration is also reset in this state so that the DAC is set to basic mode with DC output.

11. RAMP_DOWN_CUST: The procedure followed by this state is same as the other two, simply there is a configuration reset added at the end so that the DAC gets back to DC output state. So, the 'POWER_DOWN' (transition 'G1') state is selected similar to earlier configuration of button or the trigger and sham mode. And, 'SHAM_STEADY' (transition 'F1') is selected again for pressing the button once with sham stimulation or if the stimulation was cancelled beforehand. Then the finishing procedure includes resetting the setup and finish flags for custom stimulation, resetting the wave configuration and the digital gain and setting the DAC to run mode again. Then again, the impedance and current

measurement procedure checks for the values to be in limits, sending the state to ‘PAUSE’ (transition ‘H1’) in case they’re out of limits.

12. SHAM_STEADY: This state is primarily present to check for the impedance measurements while being in a steady state, something like the earlier steady output state with nothing being applied at the output so that it can run for the intended time without giving away that it is a sham stimulation procedure. In this state a constant output small current is sent once the delay set for impedance measurement reaches. After it settles for a bit, the impedance and current results are taken and sent to the external device, and if there is any problem with the results being out of limits the next state changes to ‘PAUSE’ (transition ‘K1’) state. Then the applied output current is reset to 0 after the measurement procedure finishes. Since this keeps going on at each impedance check interval, if the time reaches the stimulation time end the next state changes to ‘POWER_DOWN’ (transition ‘J1’) state. Or, if the button is pressed in between the next state changes to ‘POWER_DOWN’.

13. POWER_DOWN: As described by the name, this state powers down everything. The DAC output is disabled, the internal DAC registers are reset, the stimulation supplies are disabled along with any relays on the way. The next state simply changes to ‘STIM_OFF’ (transition ‘L1’) after everything is disabled.

14. PAUSE: The pause state is similar to power down state, since the device goes to power down after the pause state. The pause state is considered like an error state more than a waiting state to resume the tasks. So, when the device is in pause state, it performs same powering down procedure as in power down state and then if the command was ‘CANCEL’, the next state changes to ‘POWER_DOWN’ (transition ‘M1’). If the

command was 'START_STIMULATION', the next state changes to 'POWER_UP' (transition 'N1') before ramp up procedures, if the command was 'CLEAR' the next state stays the same, if the command was to do impedance check procedure 'ZCHECK', the next state changes to 'POWER_UP' with the impedance check flag set. If the button was pressed again while in paused mode, the next state just goes to 'POWER_DOWN' mode.

After analyzing the system, the changes made to be able to get a custom waveform output with the XCSITE 100 system included adding custom commands for setting the DAC parameters, DAC setup procedure for getting the custom waveform output based on the data stored in the DAC SRAM, extra functions for easily setting up and writing to the DAC. Most of the added commands and changed procedures are explained further. Although commands were added, the procedure used to test the system was by loading an array from the flash memory of PIC into the DAC SRAM then testing the working of the system, since the changes to application to input the custom waveform were decided later. Functions added to the firmware and their descriptions:

1. 'changeptimebase(value)': This function can be used to set the time-base of the output samples from the DAC, for the SRAM output. The time-base is selected by sending the value to the PAT_TIMEBASE register of the DAC. The waveform start delay base is programmed in the START_DELAY_BASE field of the PAT_TIMEBASE register. The PAT_PERIOD_BASE field in the PAT_TIMEBASE register sets the number of CLKP/CLKN clocks per PATTERN_PERIOD LSB. The pattern period can be set by the PAT_PERIOD register, and should be set according to the time-base register parameters to be able to get the complete samples at the output which might be different from the total length of the samples if continuous wave

repetition is required. The minimum possible sample output rate is $F_{clk}/16$, where F_{clk} is the clock applied to the DAC.

2. 'setsramvalue(address,value)': This function can be used to push individual values to the DAC SRAM and verify the correct write. The address to send the data value to and the value is sent as arguments to this function. The address allowed is from 0 to 4095, since total 4096 values can be stored in the SRAM and they are from 0x6000 to 0x6fff [STIM3]. The values allowed are from -2048 to 2047, which is 12 bits of signed values.

Custom/changed commands added to the program and their descriptions:

1. 'DC_CONST1': This command can be used to calculate the maximum value allowed at the output. This is calculated using the currently set output values and the maximum output that the DAC can supply. Using this the scaling factor is calculated which can help shift the gain value according to the set maximum current value. For example, let's say we need to reach 1mA maximum amplitude at the output, the current maximum amplitude possible with full values is 2.908mA. So, the scaling factor will become $1/2.908 = 0.344$. Now, maximum digital gain value is 0x3FF, so the maximum digital gain we should go to becomes $0x3FF * 0.344 = 351$ or 0x15F. We use this value further in ramp up and ramp down parameters to ramp the signal over time.
2. 'RAMPUP_PARAMETERS': Here, a procedure to calculate the value to increment the digital gain by per tick of the timer is given. This value is calculated using the final value that has to be reached which is calculated in '1', the time sent to this function and the ticks/sec for the timer i.e. 400.

3. 'RAMPDN_PARAMETERS': The calculation done here is same as the ramp-up parameter calculation, just that this is a value to decrement from the final value instead of incrementing from 0.
4. 'SET_CUSTOMSTIM': This command sets the flag to let the remaining code know that custom stimulation procedures will be followed.
5. 'SET_SRAM': This command can be used by the Bluetooth external device to set the values stored in each SRAM location of the DAC. This command also returns the value that was actually written to the specified SRAM location which can be used to verify a correct write, in case there was any problem while getting the value so the final waveform is confirmed to be right.
6. 'SET_SRAMRANGE': This command can be used to set the SRAM range from where the stimulation has to be performed. This command can be used to select parts of stored data in SRAM which need to be stimulated so that a big portion can be written to the SRAM then reused without needing writes again and again.

These commands and functions have been incorporated with the added state machine states along with few changes in other states to get all the functionality without breaking the current stimulation platform. The calculations prepared in DC_CONST1 command to scale the output waveform also help overcome overcurrent stimulation so that only around 2mA maximum output can be sent as was intended with the original device.

3.3. Result

A python script was written to generate a known signal of 2048 samples so that it can be written to the DAC SRAM from the program memory of the PIC. The waveform generated is shown in Fig 3.5.

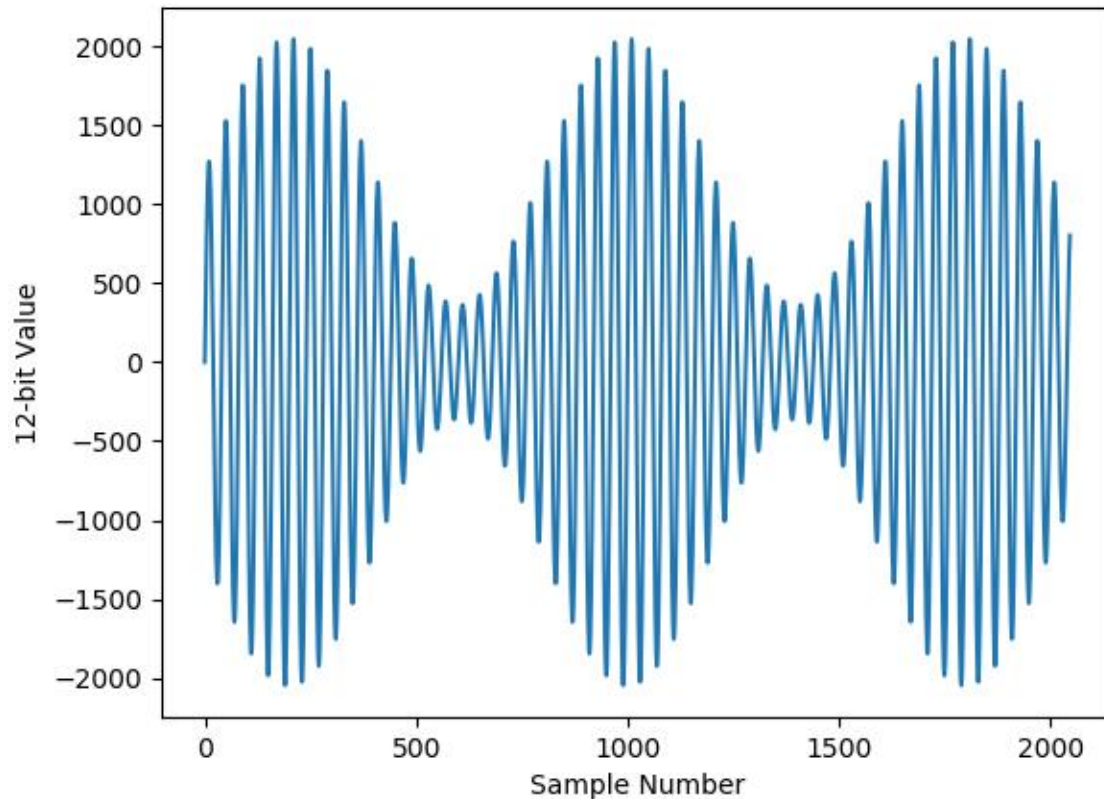


Figure 3.5 Applied custom waveform.

The relevant configuration used to analyze the output is given in table 3.1. The device used for recording the output is analog discovery 2 [ASIS6], with its CH1 oscilloscope leads simply connected across the resistor connected to the stimulation terminals. The result obtained by using this waveform for stimulation with the modified system is shown in Fig 3.6.

Table 3.1 XCSITE 100 change analysis relevant configuration.

Parameter	Configuration
Duration	14 Sec
Ch.1 Amplitude	2 mA
Ramp Up Time	2 Sec
Ramp Down Time	2 Sec
Output Resistance Connected	1 KOhm

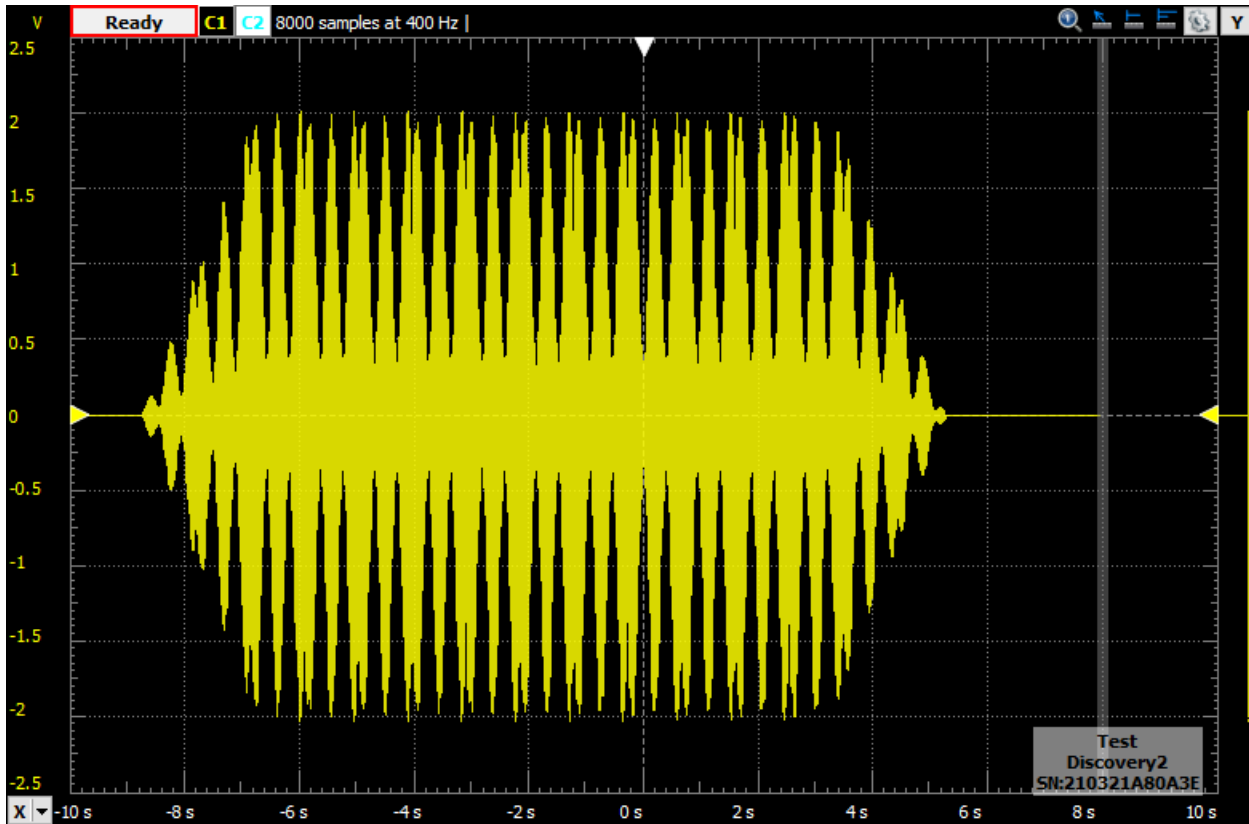


Figure 3.6 Output custom waveform result.

From the result we can see the waveform is nicely ramping up for 2 sec, staying steady for 10 sec and then ramping down for 2 sec while staying within the output current levels, the output shown is in voltage across the 1 KOhm resistor which should come out to be $2 \text{ mA} * 1 \text{ KOhm} = 2\text{V}$ same as what the result is coming out to be. Another example of the same waveform applied for viewing the output signal properly, with a longer steady state time and a different time-base explained in earlier section is shown in Fig 3.7. From this result we can confirm that the output waveform is the same as the one intended, reconfigured to a different time-base (different sample hold time).

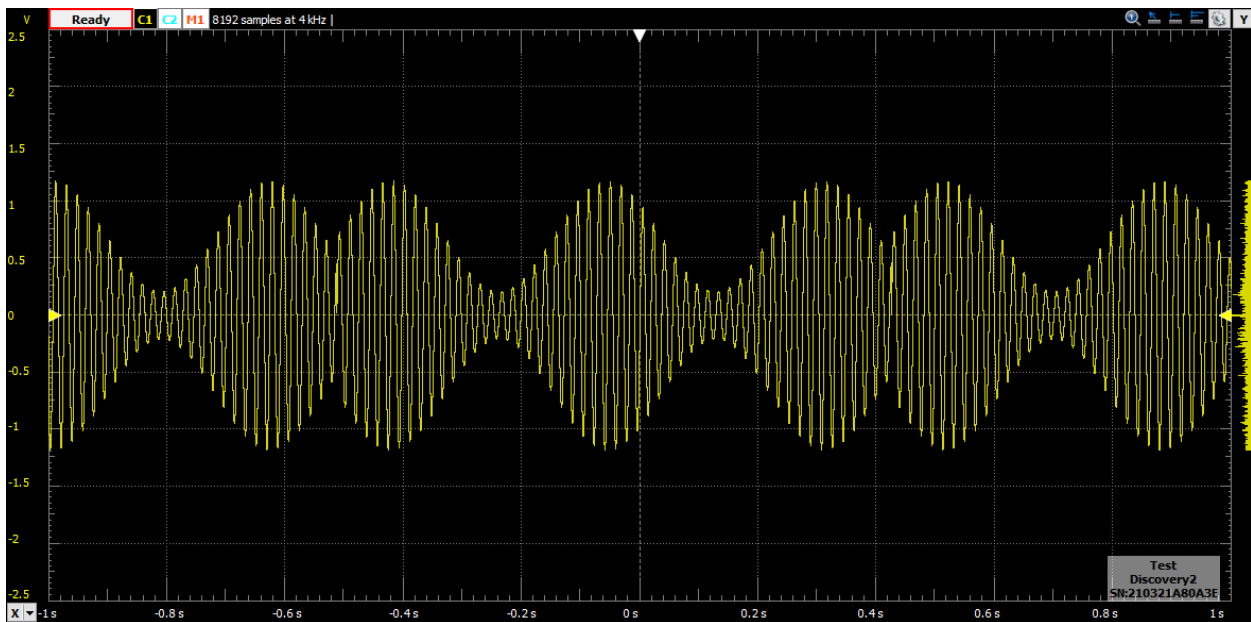


Figure 3.7 Output custom waveform with a different time-base.

REFERENCES

- [INTR1] Electroencephalography, Wikipedia.
<https://en.wikipedia.org/wiki/Electroencephalography>
- [INTR2] Electroencephalography: Basic Principles, Clinical Applications, and Related Fields, By Niedermeyer E. ISBN: 978-0-7817-5126-1.
- [INTR3] Handbook of EEG interpretation, By Tatum, William O. ISBN: 9781617051807.
- [INTR4] Continuous EEG Monitoring in the Intensive Care Unit: Early Findings and Clinical Efficacy, By Vespa, Paul M.; Nenov, Val; Nuwer, Marc R., Journal of Clinical Neurophysiology, 1999. DOI:10.1097/00004691-199901000-00001
- [INTR5] Global, regional, and national incidence, prevalence, and years lived with disability for 310 diseases and injuries, 1990-2015: a systematic analysis for the Global Burden of Disease Study 2015, Lancet. DOI:10.1016/S0140-6736(16)31678-6.
- [INTR6] Global, regional, and national life expectancy, all-cause mortality, and cause-specific mortality for 249 causes of death, 1980-2015: a systematic analysis for the Global Burden of Disease Study 2015, Lancet. DOI:10.1016/s0140-6736(16)31012-1.
- [INTR7] Magnetoencephalography-theory, instrumentation, and applications to noninvasive studies of the working human brain, By Hämäläinen, Matti; Hari, Riitta; Ilmoniemi, Risto J.; Knuutila, Jukka; Lounasmaa, Olli V., Reviews of Modern Physics. DOI:10.1103/RevModPhys.65.413.
- [INTR8] Epilepsy Fact sheet, WHO. February 2016.
- [INTR9] Detecting epileptic seizures with electroencephalogram via a context-learning model, By Guangxu Xun, Xiaowei Jia, Aidong Zhang, NCBI. DOI: 10.1186/s12911-016-0310-7

- [INT10] Epilepsy seizure detection using EEG signals, By Zakareya Lasefr ; Sai Shiva V N R Ayyalasomayajula ; Khaled Elleithy, IEEE Xplore. DOI: 10.1109/UEMCON.2017.8249018
- [INT11] Real-Time Epileptic Seizure Detection Using EEG, By Lasitha S. Vidyaratne ; Khan M. Iftekharuddin, IEEE Xplore. DOI: 10.1109/TNSRE.2017.2697920
- [INT12] Patient-Specific Early Seizure Detection from Scalp EEG, By Georgiy R. Minasyan, John B. Chatten, Martha Jane Chatten, and Richard N. Harner, NCBI. DOI: 10.1097/WNP.0b013e3181e0a9b6
- [INT13] Automatic Epileptic Seizure Detection Using Scalp EEG and Advanced Artificial Intelligence Techniques, By Paul Fergus, David Hignett, Abir Hussain, Dhiya Al-Jumeily, and Khaled Abdel-Aziz, BioMed Research International. DOI: 10.1155/2015/986736
- [INT14] Mindwave, NeuroSky. <https://store.neurosky.com/pages/mindwave>
- [INT15] Mindset research tools- NeuroView and NeuroSkyLab, NeuroSky. <https://store.neurosky.com/products/mindset-research-tools>
- [INT16] EMOTIV Insight 5, EMOTIV. <https://www.emotiv.com/product/emotiv-insight-5-channel-mobile-ee/>
- [INT17] Transcranial direct-current stimulation, Wikipedia. https://en.wikipedia.org/wiki/Transcranial_direct-current_stimulation
- [INT18] Cranial electrotherapy stimulation, tACS, Wikipedia. https://en.wikipedia.org/wiki/Cranial_electrotherapy_stimulation
- [INT19] Adverse events of tDCS and tACS: A review, By Hideyuki Matsumotoa, Yoshikazu Ugawabc, Elsevier. DOI: 10.1016/j.cnp.2016.12.003

- [INT20] Alpha-Stim and its waveform technology, Alpha-Stim. <https://www.alpha-stim.com/healthcare-professionals/history-of-the-waveform/>
- [TGAM1] EEG: TGAM, NeuroSky. <https://store.neurosky.com/products/eeg-tgam>
- [TGAM2] TGAM1 Spec Sheet, NeuroSky, Mar 2010.
http://wearcam.org/ece516/neurosky_eeg_brainwave_chip_and_board_tgam1.pdf
- [TGAM3] Guideline 8: Guidelines for Recording Clinical EEG on Digital Media, By American Clinical Neurophysiology Society. <https://www.acns.org/pdf/guidelines/Guideline-8.pdf>
- [TGAM4] Guideline One: Minimum Technical Requirements for Performing Clinical Electroencephalography, By American Clinical Neurophysiology Society.
<https://www.acns.org/pdf/guidelines/Guideline-1.pdf>
- [TGAM5] ThinkGear MindSet Communication Protocol, NeuroSky, June 2010.
http://developer.neurosky.com/docs/lib/exe/fetch.php?media=mindset_communications_protocol.pdf
- [ASIS1] Matrix Laboratory (MATLAB) R2016a, By MathWorks, Wikipedia.
<https://en.wikipedia.org/wiki/MATLAB>
- [ASIS2] Über das Elektrenkephalogramm des Menschen, By H. Berger. Archiv f. Psychiatrie (1929) 87: 527.
- [ASIS3] The Berger rhythm: potential changes from the occipital lobes in man, By E.D. Adrian and B.H.C. Matthews. Brain 1934: 57; 355–385.
- [ASIS6] Analog Discovery 2: 100MS/s USB Oscilloscope, Logic Analyzer, Function Generator and Variable Power Supply, By Digilent.

<https://store.digilentinc.com/analog-discovery-2-100msps-usb-oscilloscope-logic-analyzer-and-variable-power-supply/>

[EASE1] ATmega328PB - 8 bit AVR Microcontroller Complete Specifications, By Atmel/Microchip.

<http://ww1.microchip.com/downloads/en/DeviceDoc/40001906C.pdf>

[EASE2] I-squared-C half-duplex synchronous communication protocol, Wikipedia.

<https://en.wikipedia.org/wiki/I%C2%B2C>

[EASE3] Serial Peripheral Interface full-duplex synchronous communication protocol, Wikipedia. https://en.wikipedia.org/wiki/Serial_Peripheral_Interface

[EASE4] Arduino Integrated Development Environment, Arduino.

<https://www.arduino.cc/en/guide/environment>

[EASE5] Secure Digital (SD) Simplified Specifications, SD Association.

<https://www.sdcard.org/downloads/pls/>

[EASE6] PCF8523, Real-Time Clock (RTC) and Calendar, NXP Semiconductors.

<https://www.nxp.com/docs/en/data-sheet/PCF8523.pdf>

[DOCK1] MATLAB MAT-File Format, MATLAB.

https://www.mathworks.com/help/pdf_doc/matlab/matfile_format.pdf

[DOCK2] Greenwich Mean Time Zone, Wikipedia.

https://en.wikipedia.org/wiki/Greenwich_Mean_Time

[DOCK3] SciPy, Python library for scientific computing and technical computing, By Community library project. <https://www.scipy.org/>

[DOCK4] Raspbian Stretch, By Raspberry Pi Foundation.

<https://www.raspberrypi.org/blog/raspbian-stretch/>

- [DOCK5] FatFs - Generic FAT Filesystem Module, By ELM-ChaN. http://elm-chan.org/fsw/ff/00index_e.html
- [HARD1] LP38691 500mA Low Dropout and Quiescent current CMOS Linear Regulator, By Texas Instruments. DigiKey part number: LP38691DT-5.0/NOPB-ND. <http://www.ti.com/lit/ds/snvs321o/snvs321o.pdf>
- [HARD2] ADP121 150mA Low Dropout and Quiescent current CMOS Linear Regulator, By Analog Devices. DigiKey part number: ADP121-AUJZ33R7CT-ND. <https://www.analog.com/media/en/technical-documentation/data-sheets/adp121.pdf>
- [HARD3] 6-pin In-System Programming (ISP) or In-Circuit Serial Programming using SPI, AVR910: In-System Programming Application Note, By Atmel/Microchip. ICSP for PIC devices, By Microchip. http://ww1.microchip.com/downloads/en/appnotes/atmel-0943-in-system-programming_applicationnote_avr910.pdf
<http://ww1.microchip.com/downloads/en/devicedoc/31028a.pdf>
- [HARD4] IEC 60601 Medical Device Design Standards, specifically part 60601-2-26, By International Electrotechnical Commission. <https://webstore.iec.ch/publication/2637>
- [HARD5] CRYSTAL 32.7680KHz 12.5PF, By Abracon. <https://www.digikey.com/product-detail/en/abracon-llc/ABS25-32.768KHZ-T/535-9166-1-ND/675683>
- [STIM1] XCSITE-100, By Pulvinar Neuro. <http://www.pulvinarneuro.com/xcsite-100.html>
- [STIM2] RSO_1212DZ, Regulated DC-DC Switching Converter, By Recom Power. Mouser Part Number: 919-RSO-1212DZ. <https://www.mouser.in/ProductDetail/RECOM-Power/RSO-1212DZ?qs=sGAEpiMZZMvGsmoEFRKS8P9bZCysYvvSwgaGtbJEtIg=>

[STIM3] AD9106, Quad, Low Power, 12-Bit, 180 MSPS, Digital-to-Analog Converter and
Waveform Generator, By Analog Devices.

<https://www.analog.com/media/en/technical-documentation/data-sheets/ad9106.pdf>

APPENDICES

Appendix A

PROGRAMS USED

A.1. Program for ATMEGA328PB on the 'EASE' Module

A.1.1. 'main.ino'

```
#include <SPI.h> //Library for SPI communication, used for SD card
#include <SD.h> //Library wrapper for SDFat SD card library for AVR
#include <Wire.h> //Library for I2C communication
#include "gpio.h" //GPIO defs

// PCF8523 RTC response conversion defs, based on incoming BCD values
#define secmin(val) ((val & 0xf) + ((val >> 4) & 0x7) * 10)
#define hrsday(val) ((val & 0xf) + ((val >> 4) & 0x3) * 10)
#define months(val) ((val & 0xf) + ((val >> 4) & 0x1) * 10)
#define years(val) ((val & 0xf) + (val >> 4) * 10)

// Initiate DateTime read from PCF8523 RTC beginning with seconds, with
address auto-incrementing in the RTC after each read
#define initread(void)
    Wire.beginTransaction(0x68);
    Wire.write(byte(0x03));
    Wire.endTransmission();
    Wire.requestFrom(0x68, 7);

#define RECORDTIME 30L // Data record time in seconds
#define WRITEBUFFER 256L // SD card data buffer size during record and write

// SPI speed setup for SD card communication.
// Available options are SPI_FULL_SPEED giving F_CPU/2 speed, SPI_HALF_SPEED
giving F_CPU/4 speed and SPI_QUARTER_SPEED giving F_CPU/8 speed.
// SPI_HALF_SPEED selected by default. Enable speed selection by uncommenting
SD.begin with speed
#define SD_SPEED SPI_FULL_SPEED
#define SD_CS 10 // SD card chip select pin used, select based on arduino pin
mapping

// Change SERIAL_RX_BUFFER_SIZE to 128 and SERIAL_TX_BUFFER_SIZE to 8 in the
included HardwareSerial.h file, to avoid any potential serial/UART RX data
loss during capture

// File element and filename buffer
File myFile;

// Open a new file based on current RTC time
void newfile()
```

```

{
    char buff[20]; // Misc buffer to store strings
    uint8_t DT[7]; // Buffer to store time from RTC
    uint8_t temp1, i;
    initread(); // Init reading from PCF8523 RTC
    while (Wire.available() != 7)
        ; // Wait for I2C device to be available
    for (i = 0; i < 7; i++) // Read and store time from RTC
    {
        temp1 = Wire.read(); // Single byte read from RTC
        if (i < 2)
            DT[i] = secmin(temp1);
        else if (i < 4)
            DT[i] = hrsday(temp1);
        else if (i == 5)
            DT[i] = months(temp1);
        else if (i == 6)
            DT[i] = years(temp1);
    }
    sprintf(buff, "%02d%02d%02d%02d.txt", DT[3], DT[2], DT[1],
        DT[0]); // Generate filename based on current Datetime read
    myFile = SD.open(buff,
        O_CREAT | O_APPEND | O_WRITE); // Create or open the file for writing
and appending data
    // Generate full timestamp ASCII string to store in opened file
    // Writes the timestamp according to year:month:day:hrs:mins:sec
    myFile.write((const char*)F("Timestamp: "));
    sprintf(buff, "%02d:%02d:%02d:%02d:%02d:%02d\n", DT[6], DT[5], DT[3],
DT[2], DT[1], DT[0]);
    myFile.write(buff, sizeof(buff)); // Write the datetime to the file
    myFile.write(
        (const char*)F("Format: year:month:day:hrs:mins:sec\n")); // Write
the format used
}

void setup()
{
    Serial.begin(57600); // Initialize UART at 57.6k baud rate for TGAM
    setinb(switchpin); // Set switch pin as input
    setinempty(); // Set the incompatible pins as input, adjust for
ATMEGA328PB, not required for ATMGEGA328P since those pins are not present
there
    setoutb(pipin); // Set the pi pin as output
    pinclrb(pipin); // Set output at pi pin as LOW
    // Set the LED pins as output
    setoutc(RED);
    setoutc(GREEN);
    setoutc(YELLOW);
    setoutc(BLUE);
    // Turn ON the RED LED
    pinsetc(RED);
    // Turn off GREEN, YELLOW and BLUE LEDs
    pinclrc(GREEN);
    pinclrc(YELLOW);
    pinclrc(BLUE);
    // Initialize I2C
    Wire.begin();
}

```

```

}

// Data capture iterator limit based on time to record
const PROGMEM uint32_t NUMCAPTURES = (RECORDTIME * (8 * 512 + 36) /
WRITEBUFFER);
// Remaining bytes to record after NUMCAPTURES captures of WRITEBUFFER size
are finished
const PROGMEM uint8_t REMSAMPLES
    = ((RECORDTIME * 8 * 512 + RECORDTIME * 36) - (WRITEBUFFER *
NUMCAPTURES));

uint8_t buf[WRITEBUFFER]; // Write buffer to take data from UART and write to
SD card
uint8_t temp = 0; // Pin status value for switch
uint32_t samples = 0; // Sampling counter for keeping track of data stored
bool runthrough = 0; // Variable to keep run sequence track for Switch ON and
exit sequence at switch OFF so they're only run once
bool breakthrough = 1; // Variable to keep track of SD card initialization

void loop()
{
    temp = pinstatb(switchpin); // Read switch value
    if (temp & (!runthrough)) // When switch is turned on for the first time,
initialize SD card and create first file
    { // Turn off RED LED and turn on GREEN LED
        pinclrc(RED);
        pinsetc(GREEN);
        runthrough = 1;
        pinsetb(pipin); // Set pi pin as 1 so Raspberry Pi doesn't try to
connect to SD card if EASE module was still connected while turning switch ON
        samples = 0;
        // while (!SD.begin(SD_SPEED, SD_CS) // SD.begin with speed selection
        while (!SD.begin(SD_CS) & breakthrough) // Keeps running sequence
till SD card initialized
        {
            temp = pinstatb(switchpin); // Read switch to see if it was
turned off after SD card initalization was tried
            if (!temp)
                breakthrough = 0; // Set to 0 to break out of the loop if
switch turned OFF and also cancels newfile creation
        }
        if (breakthrough) // Create new file if switch was when the earlier
loop was finished
            newfile();
        else
            breakthrough = 1; // Initialization cancelled by switch, reset
the variable to 1 so next initialization can be done
            // Turn off GREEN LED and turn on YELLOW LED
            pinclrc(GREEN);
            pinsetc(YELLOW);
        }
        if (temp) // If switch is ON/initialization successful, records data
        {
            Serial.readBytes(buf, WRITEBUFFER); // Record 'WRITEBUFFER' amount of
bytes in buffer
            myFile.write(buf,

```

```

        WRITEBUFFER); // Write those 'WRITEBUFFER' amount of bytes to the
currently open file
        samples += 1; // Increase the samples recorded counter to calculate
approximate time passed
        if (samples == NUMCAPTURES) // Close current file and open new file
when required time of data is captured
        {
            if (REMSAMPLES > 0) // Only runs if there are any samples left
            {
                Serial.readBytes(buf, REMSAMPLES); // Record remaining sample
bytes
                myFile.write(buf, REMSAMPLES); // Write the recorded bytes to
file
            }
            myFile.close(); // Close currently open file
            samples = 0; // Reset recorded sample counter
            newfile(); // Open new file using current time
        }
    }
else if (runthrough) // If switch is closed, run this once
{
    if (myFile)
        myFile.close(); // Close the currently open file
        samples = 0; // Reset recorded sample counter
        runthrough = 0;
        // Turn YELLOW LED off and turn RED LED on
        pinclr(YELLOW);
        pinsetc(RED);
        pinclrb(pipin); // Reset the pi pin to 0 to let it know it can
connect to the SD card
    }
else
    delay(20); // 20ms delay before rechecking pin selection if device is
in OFF mode
}

```

A.1.2. 'gpio.h'

```

// PORT accesses and functions for the GPIO pins on port B and port C

// Read input from port B pin z
#define pinstatb(z) ((PINB & (1 << z)) >> z)
// Set port B pin z as input
#define setinb(z) (DDRB &= ~(1 << z))
// Sets 4 pins on port E as input to avoid controller problems, required if
ATMEGA328PB is used which has extra pins
#define setinempty() (DDRE &= ~(0xf))
// Set port B pin z as output
#define setoutb(z) (DDRB |= (1 << z))
// Set output at port B pin z as 'high'
#define pinsetb(z) (PORTB |= (1 << z))
// Set output at port B pin z as 'low'
#define pinclrb(z) (PORTB &= ~(1 << z))
// Set port C pin z as output

```

```

#define setoutc(z) (DDRC |= (1 << z))
// Set output at port C pin z as 'high'
#define pinsetc(z) (PORTC |= (1 << z))
// Set output at port C pin z as 'low'
#define pinclrc(z) (PORTC &= ~(1 << z))

// Pin connected to raspberry pi showing data is ready for download, located
on port B
#define pipin 1
// Pin connected to the switch, located on port B
#define switchpin 0

// LED locations for each color on port C
#define RED 0
#define GREEN 1
#define YELLOW 2
#define BLUE 3

```

A.2. Programs for Raspberry Pi on the Docking Station

A.2.1. 'main.py'

```

#!/usr/bin/python
# -*- coding: utf-8 -*-
import numpy
import time
import scipy.io as s
import os
from subprocess import Popen as p
from shutil import rmtree
import RPi.GPIO as gp

# All pins are in BCM mode

# Pin where EASE Module signals ready to connect

easepin = 4

# SPI connection channel pin numbers

SDchan = [10, 9, 11, 7]

# LED locations on GPIO
# RED LED turned ON when device is ready to be connected to EASE module, or
EASE module is ready to be removed if already connected

RED = 1

# GREEN LED turned ON when the device is trying to connect to the SD card

GREEN = 12

# YELLOW LED turned ON when the device is downloading data from the SD card

```

```

YELLOW = 16

# BLUE LED turned ON when the device is parsing the downloaded data, and EASE
module can be removed from this stage

BLUE = 21

LEDS = [RED, GREEN, YELLOW, BLUE]

# Function to set the default pin mode and status

def setpins():
    gp.setmode(gp.BCM) # Set pin usage mode as BCM
    gp.setup(easepin, gp.IN, pull_up_down=gp.PUD_UP) # Set the EASE module pin
as input with an internal pullup
    gp.setup(LEDS, gp.OUT) # Set the LEDs as output
    gp.setup(SDchan, gp.IN) # Set the SD card SPI pins as input by default to
put them in high impedance mode
    gp.output(LEDS, 0) # Set the output on LEDs as LOW to turn them all OFF

setpins()

while 1: # Infinite loop
    gp.output(RED, 1) # Turn the RED LED ON
    while 1: # Waiting till the EASE module is connected
        gp.wait_for_edge(easepin, gp.FALLING) # Wait for EASE module to be
connected in connection ready state
        time.sleep(5) # Recheck after 5mS
        if gp.input(easepin): # If it was just a misconnection by any chance,
ignore the falling edge
            pass
        else:
            break # If EASE module is really connected stop waiting and move on to
next step
    gp.output(RED, 0) # Turn the RED LED OFF
    gp.output(GREEN, 1) # Turn the GREEN LED ON
    print 'EASE module connected.' # Debug text
    proc = p('python ./rtc/settime.py', shell=1) # Run the program to get the
GMT and set the RTC to that time
    proc.wait() # Wait for the program to finish setting the time
    print 'RTC time set.' # Debug text
    dirs=[] # Empty list for directory check
    for z in os.listdir('./'): # Check for currently present directories
        if os.path.isdir(os.path.join(os.getcwd(),z)): # If the path is a directory
add it to the list
            d += [z]
    reqdirs=['parse','data','record'] # List of the required directories
    for y in reqdirs: # Check if the required directories are present
        isdir=0 # Flag to keep track if the directories are present
        for z in d:
            if z == y: # If the directory is present, set the flag to 1 and break out
of loop
                isdir = 1
                break

```

```

    if (isdir == 0): # If the flag is set, the directory is present, if not
make the directory
    os.mkdir(y)
    print 'Copying Data...' # Debug text
    proc = p('./sdsrmod/getsd', shell=0) # Run the program to download the
data from the SD card
    proc.wait() # Wait till the data has been downloaded and files cleared from
SD card
    gp.cleanup() # Cleanup the GPIO to set all the GPIO used as input mode to
disconnect the EASE module without any problem
    setpins() # Set the default state of the pins again
    gp.output(BLUE, 1) # Turn the BLUE LED ON, GREEN and YELLOW LEDs are
controlled by the SD Card download program
    print 'Device ready to be disconnected.\n' # Debug text
    print 'Beginning data parsing and conversion...' # Debug text
    for x in os.listdir('./record/'): # Begin data parsing for all the files
downloaded
        filename = x[:-4]
        f = open('./record/' + filename + '.TXT', 'rb') # Open the downloaded file
to read
        f1 = open('./parse/' + filename + '_2.TXT', 'wb') # Open the debug parsing
file to write
        while 1:
            try: # Runs if the file hasn't reached the end during the packet read
                if f.read(2) == '\xaa\xaa': # Packet sync bytes
                    if ord(f.read(1)) == 4: # Check for right packet length
                        buff = f.read(4) # Read 4 bytes as packet length
                        chksm = 0
                        for z in range(4): # Calculate the checksum
                            chksm += ord(buff[z])
                        realchksm = ord(f.read(1)) # Read 1 more byte for the real checksum
output by device
                        if chksm & 0xff ^ 0xff == realchksm: # If the checksum matches, write
the data to parsed file
                            f1.write(buff[2])
                            f1.write(buff[3])
                        else:
                            f.seek(-5, os.SEEK_CUR) # If something doesn't match go back to the
sync bytes to avoid other packet's data getting skipped
                        else:
                            f.seek(-2, os.SEEK_CUR) # If something doesn't match go back to the
sync bytes to avoid other packet's data getting skipped
                        else:
                            f.seek(-1, os.SEEK_CUR) # If something doesn't match go back to the
sync bytes to avoid other packet's data getting skipped
                    except:
                        break # Break out of the loop when the file has reached the end
                    if len(f.read(2)) == 1: # Break if it's the end of the file, checking for
EOF character
                        break
                    else:
                        f.seek(-2, os.SEEK_CUR) # If not the end of file, go back to the last
position
                f1.close() # Close the parsed data file
                f.close() # Close the recorded data file
                f = open('./parse/' + filename + '_2.TXT', 'rb') # Open the parsed data
file to convert the data and store

```

```

datalist = [] # Empty data list for the current file
z = 0
while 1:
    try: # Try getting a byte from the file
        k = ord(f.read(1))
    except:
        break # Break if it's the end of file
    if z % 2 == 0: # Add the higher byte to the data list
        datalist += [k << 8]
    else:
        datalist[z / 2] = datalist[z / 2] | k # Add the lower byte to the
        currently stored higher byte in the data list
        z += 1
f.close() # Close the parsed data file
neg = 0
for z in range(len(datalist)): # Convert the data from 16 bit signed to
    default 32 bit signed
    if datalist[z] & 0x8000: # 2's compliment method to convert the data
        datalist[z] = -((datalist[z] ^ 0xffff) + 1)
    if len(datalist) % 2 != 0: # If the length of the data list isn't even,
    remove the last sample to keep it symmetric for analysis
    del datalist[-1]
s.savemat('./data/' + filename + '.mat',
    mdict={'data': datalist}) # Store the data in MAT format (32 bit with
padding) in variable named 'data', using scipy lib
print filename + ' file samples: ' + str(len(datalist)) # Debug text to
check amount of samples found in the current file
del datalist # Unlink the data list to avoid constant memory usage
os.remove('./record/' + filename + '.TXT', 'rb') # Remove the parsed file
data
os.remove('./parse/' + filename + '_2.TXT', 'rb') # Remove the parsed file
data
print 'Finished data parsing and conversion.' # Debug text
gp.output(BLUE, 0) # Turn the BLUE LED OFF at the end of parse and
conversion

```

A.2.2. './rtc/settime.py'

```

#!/usr/bin/python
# -*- coding: utf-8 -*-
import smbus # Library for I2C communication
import urllib2 # Library for dealing with webpages
from datetime import datetime # Inbuilt datetime module

# Get the access to port I2C port 1

bus = smbus.SMBus(1)

# Device address for the PCF8523 RTC

dev = 0x68

# Value for performing software reset in the RTC

```

```

rescom = 0x58

# Perform RTC software reset by writing above value in Control register 1
bus.write_byte_data(dev, 0, rescom)

# Register mappings for the RTC

sec = 3
min = sec + 1
hrs = min + 1
day = hrs + 1
weekday = day + 1
month = weekday + 1
year = month + 1

# Function to convert the input integer into a BCD format

def writeparse(val):
    return (((val / 10) << 4) | (val % 10))

# Reading current GMT and parsing it

time = \
    datetime.strptime(urllib2.urlopen('http://just-the-time.appspot.com/'
    ,timeout=5).read().strip(), '%Y-%m-%d %H:%M:%S')

# Set the RTC time according to the GMT time obtained

bus.write_byte_data(dev, sec, writeparse(time.second)) # Setting the seconds
value by seconds register
bus.write_byte_data(dev, min, writeparse(time.minute)) # Setting the minutes
value by minutes register
bus.write_byte_data(dev, hrs, writeparse(time.hour)) # Setting the hours
value by hours register
bus.write_byte_data(dev, day, writeparse(time.day)) # Setting the day value
by the day register
bus.write_byte_data(dev, month, writeparse(time.month)) # Setting the month
value by the month register
bus.write_byte_data(dev, year, writeparse(time.year % 100)) # Setting the
year value by the year register, and wrapping it around 100 years time to
meet with RTC allowed time

# Wrapping the calculated weekday by the OS to the RTC format, since for RTC
Sunday=0 and for linux Monday=0
# Then writing that to the RTC

if time.weekday() == 6:
    bus.write_byte_data(dev, weekday, 0)
else:
    bus.write_byte_data(dev, weekday, time.weekday() + 1)

```

A.2.3. './sdsrmod/main.c'

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "ff.h" //FatFs library by ElmChan
#include <bcm2835.h> //Library for GPIO operations

#define buffsize 4096 // File read buffer size

FATFS Fatfs; // File system object
FIL Fil; // FatFs File object
BYTE Buff[buffsize]; // File read buffer
int datadownload = 0; // Flag to check current state

// Variable setting the SPI speed for SD card access, originally defined in
pin mapping file
extern int setdelay;
// LED pin mappings from the pin mapping file
extern int RED, GREEN, YELLOW, BLUE;

void emp( // Process error message
    FRESULT rc // FatFs return value
)
{
    printf("\nFailed with rc=%u.\n", rc); // Print the error message for
debug
    if (datadownload) // If the data download was ongoing when the error
appeared
    {
        printf("\nRetrying with slower speed.\n"); // Debug msg
        setdelay++; // Increase the delay between toggles, i.e. reduce the
SPI speed
        bcm2835_gpio_set(
            GREEN); // Turn ON the GREEN LED to show connection problem
        bcm2835_delay(10); // Wait for 10mS to let the user notice
        bcm2835_gpio_clr(
            GREEN); // Turn OFF the GREEN LED
    }
    else
    {
        if (rc != 3)
            exit(0); // If data download isn't going on and there is no
initialization problem, exit with an error code 0
    }
}

int main(void)
{
    FRESULT rc; // Result object, storing result from FatFs library
    DIR dir; // Directory object
    FILINFO fno; // File information object
    unsigned int br; // Number of bytes read from file tracker
    FILE* fp; // File pointer for file I/O
    char namebuf[5] = { 0 }; // Temporary buffer for file extension check
```

```

char filenamebuf[30]; // File name buffer
unsigned int nums = 0; // Number of files found counter

f_mount(0, &Fatfs); // Register volume work area
printf("\nConnecting to the SD card and testing connection speed.\n"); //
Debug text

setdelay = 5; // Set the SPI speed very high at first
bcm2835_gpio_set(
    GREEN); // Turn ON the GREEN LED to show Raspberry Pi is trying to
connect to SD card
do
{
    rc = f_opendir(&dir, ""); // Try opening the base directory of the SD
card

    if (rc) // Error message handling if there was any error during SD
card access
    {
        emp(rc);
        setdelay++; // Reduce the SPI speed if there was any problem
connecting
    }
    bcm2835_delay(100); // Wait for 100mS before trying to connect again
} while (rc == 3); // Keep running only if there was a problem while
connecting to SD card, not if SD card returned something else
// If the setdelay variable was greater than 15, try to connect to the SD
card again with a slower starting speed
if (setdelay > 15)
{
    setdelay = 10;
do
{
    rc = f_opendir(&dir, "");
    if (rc)
    {
        emp(rc);
        setdelay++;
    }
    bcm2835_delay(100);
} while (rc == 3);
}
// Turn the ON LED from GREEN to YELLOW to show data downloading will be
started and SD card connection was successful
bcm2835_gpio_clr(GREEN);
bcm2835_gpio_set(YELLOW);
// SPI Speed calculation for debugging according to the value of setdelay
variable, calculation based on approximate real world calculations
if (setdelay > 10)
    printf("\nCopying files with SPI speed approximately %3.2fMHz\n",
        (4 / (2 + (setdelay - 11) / 10.0)));
else
    printf("\nCopying files with SPI speed approximately %3.2fMHz\n",
        (4 / (1.1 + (setdelay - 5) / 5.0)));
datadownload = 1; // Set the data download flag to 1 before entering data
download mode
for (;;)

```

```

    {
        rc = f_readdir(&dir, &fno); // Read a directory item
        if (rc || !fno.fname[0])
            break; // Finish the program if any error was there or if the SD
card is empty
        strncpy(namebuf, &fno.fname[8],
            4); // Copy the file extension in a buffer, according to SD
filename standards
        if (strcmp(namebuf, ".TXT")
            == 0) // Check the extension to be TXT according to the files
stored while recording
            {
                rc = f_open(
                    &Fil, fno.fname, FA_READ); // If the extension was TXT, open
the file for read
                if (rc) // SD card Error handling
                    emp(rc);
                sprintf(filenamebuf, "./record/%s", fno.fname); // Generate the
filename to download data in same as the filename from which we are reading
                printf("Writing to file %s\n", filenamebuf); // Debug text
                fp = fopen(filenamebuf, "wb"); // Open the local file to write
downloaded data in
                for (;;)
                {
                    rc = f_read(&Fil, Buff, bufsize, &br); // Read a chunk of
file from the SD card based on read buffer size selected earlier
                    if (rc || !br)
                        break; // Error while reading or end of file if br is 0
                    fwrite(Buff, sizeof(char), br, fp); // Write the downloaded
data from the SD card
                }
                if (rc) // SD card Error handling
                    emp(rc);
                fclose(fp); // Close the file opened for writing the downloaded
data to
                rc = f_close(&Fil); // Close the opened file on SD card
                if (rc) // SD card Error handling
                    emp(rc);
                rc = f_unlink(fno.fname); // Delete the downloaded file from SD
card
                if (rc) // SD card Error handling
                    emp(rc);
                nums++; // Increment the number of files read
            }
        if (rc) // SD card Error handling
            emp(rc);
    }

    printf("\n%d data records found.\n", nums); // Debug text to know how
many files were downloaded
    printf("\nFinished copying.\n"); // Debug text
    bcm2835_gpio_clr(YELLOW); // Turn the YELLOW LED OFF to show data
download finished
    return 1; // Successfully finished execution
}

```

A.2.4. './sdsrmod/mmcbb.c'

```
#include "diskio.h" //Common include file for FatFs and disk I/O layer
#include <bcm2835.h> // BCM library for GPIO pin control

#define INIT_PORT() init_port() // Initialize SD/MMC control port (CS=H,
CLK=L, DI=H, DO=in)
#define DLY_US(n) bcm2835_delayMicroseconds(n) // Delay n microseconds

// Set delay for generating delay between pin toggles, for manual delay
control
int setdelay = 20;
// LED GPIO pin mappings according to BCM
int RED = 1, GREEN = 12, YELLOW = 16, BLUE = 21;

void MYDLY(void)
{
    for (int i = setdelay; i; i--)
        ;
} // Custom delay function for generating a delay between pin toggles

#define DO bcm2835_gpio_lev(RPI_GPIO_P1_21) // I/P read for SD/MMC DO; MISO
void CS_H(void)
{
    bcm2835_gpio_set(RPI_GPIO_P1_26);
    MYDLY();
} // Set SD/MMC CS "high"
void CS_L(void)
{
    bcm2835_gpio_clr(RPI_GPIO_P1_26);
    MYDLY();
} // Set SD/MMC CS "low"
void CK_H(void)
{
    bcm2835_gpio_set(RPI_GPIO_P1_23);
    MYDLY();
} // Set SD/MMC SCLK "high"
void CK_L(void)
{
    bcm2835_gpio_clr(RPI_GPIO_P1_23);
    MYDLY();
} // Set SD/MMC SCLK "low"
void DI_H(void)
{
    bcm2835_gpio_set(RPI_GPIO_P1_19);
    MYDLY();
} // Set SD/MMC DI "high"; MOSI
void DI_L(void)
{
    bcm2835_gpio_clr(RPI_GPIO_P1_19);
    MYDLY();
} // Set SD/MMC DI "low"; MOSI

void init_port(void)
```

```

{
    bcm2835_init(); // Init BCM library
    bcm2835_gpio_fsel(RPI_GPIO_P1_23, BCM2835_GPIO_FSEL_OUTP); // Set SCK as
O/P
    bcm2835_delayMicroseconds(1);
    bcm2835_gpio_clr(RPI_GPIO_P1_23); // Clear SCK pin
    bcm2835_delayMicroseconds(1);
    bcm2835_gpio_fsel(RPI_GPIO_P1_19, BCM2835_GPIO_FSEL_OUTP); // Set MOSI as
O/P
    bcm2835_delayMicroseconds(1);
    bcm2835_gpio_set(RPI_GPIO_P1_19); // Set MOSI pin
    bcm2835_delayMicroseconds(1);
    bcm2835_gpio_fsel(RPI_GPIO_P1_26, BCM2835_GPIO_FSEL_OUTP); // Set CS1
channel for SPI as O/P
    bcm2835_delayMicroseconds(1);
    bcm2835_gpio_set(RPI_GPIO_P1_26); // Set CS pin
    bcm2835_gpio_fsel(RED, BCM2835_GPIO_FSEL_OUTP); // Set RED LED as O/P
    bcm2835_delayMicroseconds(1);
    bcm2835_gpio_clr(RED); // Turn OFF RED LED
    bcm2835_gpio_fsel(GREEN, BCM2835_GPIO_FSEL_OUTP); // Set GREEN LED as O/P
    bcm2835_delayMicroseconds(1);
    bcm2835_gpio_set(GREEN); // Turn ON GREEN LED, since we begin in SD
connection mode
    bcm2835_gpio_fsel(YELLOW, BCM2835_GPIO_FSEL_OUTP); // Set YELLOW LED as
O/P
    bcm2835_delayMicroseconds(1);
    bcm2835_gpio_clr(YELLOW); // Turn OFF YELLOW LED
    bcm2835_gpio_fsel(BLUE, BCM2835_GPIO_FSEL_OUTP); // Set BLUE LED as O/P
    bcm2835_delayMicroseconds(1);
    bcm2835_gpio_clr(BLUE); // Turn OFF BLUE LED
    bcm2835_gpio_fsel(RPI_GPIO_P1_21, BCM2835_GPIO_FSEL_INPT); // Set MISO as
I/P
    bcm2835_gpio_set_pud(RPI_GPIO_P1_21, BCM2835_GPIO_PUD_UP); // Enable
Pull-Up resistor on MISO
}

```

A.3. Programs Used for Analysis of the Recorded Data

A.3.1. “multipleanalysis.m”

```

for volts = [10 50 100] % Voltages tested, in uV
    maxerrors = []; % Empty Maximum Errors array for each voltages and
frequencies
    for freq = [2 4 6 8 10 12 14 16 18 20] % Frequencies tested, in Hz
        freqdata = []; % Empty frequency data array for frequencies obtained on
each iteration
        for z = 0:9 % Loop through 10 iterations of each frequency at each
voltage level
            load(['./data/data_' num2str(freq) 'Hz_' num2str(volts) 'uV_'
num2str(z) '.mat']); % Load the data file
            data = cast(data, 'double'); % Case the data from u16 to double
            Fs = 512; % Sampling frequency of TGAM

```

```

T = 1 / Fs; % Time per sample
L = length(data); % Length of the data signal loaded
t = (0:L - 1) * T; % Array of sample in time

% Uncomment the following code to apply a low pass butterworth filter
given by cutoff frequency
% N = 10; % Order of the filter
% cutoff_Hz = 30; % -3dB cutoff frequency
% [b,a]=butter(N,cutoff_Hz/(Fs/2),'low'); % Low-pass butterworth
filter setup
% filterdata = filter(b,a,data); % Filter the data defined by b, a as
transfer function coefficients
% save(['./data/filterdata_' num2str(freq) 'Hz_' num2str(volts) 'uV_'
num2str(z) '.mat'],'filterdata'); % Store the filtered data

Y = fft(data); % Apply FFT on the data selected
P2 = abs(Y / L); % Absolute function on normalized data, to wrap both
sides
P1 = P2(2:L / 2 + 1); % Wrap the mirrored FFT data in half
f = Fs * (1:(L / 2)) / L; % Generate the frequency plot axis data
plot(f, P1); % Plot the frequency and normalized/relative amplitude
xlim([0 30]); % Limit frequency axis to the frequencies we care about
[Peak, PeakIdx] = max(P1(2:600)); % Capture the peak from the data
text(f(1 + PeakIdx), Peak, sprintf(' Peak F = %.2f Hz', f(1 +
PeakIdx))); % Print the peak on the graph
freqdata = [freqdata, f(1 + PeakIdx)]; % Store the peak in freqdata
array
xlabel('Frequency (in Hz) ->'); % X label
ylabel('Relative Amplitude ->'); % Y label
title(['Frequency Plot ' num2str(z) ' for applied frequency '
num2str(freq) 'Hz at ' num2str(volts) 'uV']); % Title
saveas(gcf, ['./graphs/nofilter' num2str(freq) 'Hz_' num2str(volts)
'uV_' num2str(z) '.png']); % Store the plotted graph
xlim auto; % Rearrange x axis limits
plot(t, data); % Plot the time domain graph for the unfiltered signal,
just for debugging
xlabel('Time (in Sec) ->'); % X label
ylabel('Amplitude (0.49uV/unit) ->'); % Y label
title(['Wave Plot ' num2str(z) ' for applied frequency ' num2str(freq)
'Hz at ' num2str(volts) 'uV']); % Title
saveas(gcf, ['./wave/nofilter' num2str(freq) 'Hz_' num2str(volts) 'uV_'
num2str(z) '.png']); % Store the plotted graph

% Uncomment the following lines to plot the frequency response for the
filtered signal
% xlim auto; % Rearrange x axis limits
% Y = fft(filterdata); % Apply the FFT on the filtered data
% P2 = abs(Y/L); %Absolute function on normalized data, to wrap both
sides
% P1 = P2(2:L/2+1); %Wrap the mirrored FFT data in half
% f = Fs*(1:(L/2))/L; %Generate the frequency plot axis data
% plot(f,P1); %Plot the frequency and normalized/relative amplitude for
the filtered signal
% xlim([0 30]); %Limit frequency axis to the frequencies we care about
% xlabel('Frequency (in Hz) ->'); %X label
% ylabel('Relative Amplitude ->'); %Y label

```

```

    % title(['Filtered Frequency Plot ' num2str(z) ' for applied frequency
' num2str(freq) 'Hz at ' num2str(volts) 'uV']); %Title
    % saveas(gcf,['./graphs/filter' num2str(freq) 'Hz_' num2str(volts)
'uV_' num2str(z) '.png']); %Store the plotted graph
    % xlim auto; % Rearrange x axis limits
    % plot(t,filterdata); % Plot the time domain graph for the filtered
signal, just for debugging
    % xlabel('Time (in Sec) ->'); %X label
    % ylabel('Amplitude (0.49uV/unit) ->'); %Y label
    % title(['Filtered Wave Plot ' num2str(z) ' for applied frequency '
num2str(freq) 'Hz at ' num2str(volts) 'uV']); %Title
    % saveas(gcf,['./wave/filter' num2str(freq) 'Hz_' num2str(volts) 'uV_'
num2str(z) '.png']); %Store the plotted graph
    % pause(1);

end
fid = fopen(['./resultstxt/' num2str(freq) 'Hz_' num2str(volts)
'uV.txt'], 'w'); % File to store the error results
for ii = 1:size(freqdata, 2) % Write the error results to the text file,
based on expected frequency and result frequency
    fprintf(fid, '%.2f\t', freqdata(ii));
    fprintf(fid, '\n');
end
fclose(fid); % Close the opened file
maxerrors = [maxerrors, freq - min(freqdata)]; % Add the maximum error to
the array keeping track of maximum errors
end
figure; % Generate a new figure control for the bar graph to plot errors
x = [2 4 6 8 10 12 14 16 18 20]; % Array of applied frequencies
bar(x, maxerrors); % Plot bargraph of applied frequencies vs maximum error
out of 10 iterations
for abc = 1:10 % Place text to write the maximum frequency value on the bar
graph
    if (volts < 50); sub = 0.0034; else sub = 0.0034 / 2; end
    if (maxerrors(abc) > 0);
        text(x(abc) - 1, maxerrors(abc) + sub, sprintf('    Error \n= %.3f
Hz', maxerrors(abc)), 'fontsize', 6);
    else
        text(x(abc) - 1, maxerrors(abc) - sub, sprintf('    Error \n= %.3f
Hz', maxerrors(abc)), 'fontsize', 6);
    end
end
end
xlabel('Frequency expected in Hz'); % Bar graph x label
ylabel('Maximum error found in Hz'); % Bar graph y label
title(['Maximum error graph for signal applied at ' num2str(volts)
'uV']); % Title for the current voltage level
xlim([0 22]); % X limit for our frequencies tested
saveas(gcf, ['Maxerror_' num2str(volts) 'uV.png']); % Store the plotted
graph
end

```

A.3.2. 'analyze.m'

```
files = dir('data'); % Load all the names of data files from the data
directory
for z = 3:length(files) % Iterate through all the data files present, first
two contain links to previous and current directory, so ignored
    load(['./data/' files(z).name]); % Load the data file from the data
directory
    data = cast(data, 'double'); % Case the data from u16 to double
    Fs = 512; % Sampling frequency of TGAM
    T = 1 / Fs; % Time per sample
    L = length(data); % Length of the data signal loaded
    t = (0:L - 1) * T; % Array of sample in time

    % Uncomment the following code to apply a low pass butterworth filter given
by cutoff frequency
    % N = 10; % Order of the filter
    % cutoff_Hz = 30; % -3dB cutoff frequency
    % [b,a]=butter(N,cutoff_Hz/(Fs/2),'low'); % Low-pass butterworth filter
setup
    % filterdata = filter(b,a,data); % Filter the data defined by b, a as
transfer function coefficients
    % save(['./data/filterdata_' files(z).name(1:end-4) '.mat'],'filterdata');
% Store the filtered data

    Y = fft(data); % Apply FFT on the data selected
    P2 = abs(Y / L); % Absolute function on normalized data, to wrap both sides
    P1 = P2(2:L / 2 + 1); % Wrap the mirrored FFT data in half
    f = Fs * (1:(L / 2)) / L; % Generate the frequency plot axis data
    plot(f, P1); % Plot the frequency and normalized/relative amplitude
    xlim([0 30]); % Limit frequency axis to the frequencies we care about
    [Peak, PeakIdx] = max(P1(2:600)); % Capture the peak from the data
    text(f(1 + PeakIdx), Peak, sprintf(' Peak F = %.2f Hz', f(1 + PeakIdx)));
% Print the peak on the graph
    freqdata = [freqdata, f(1 + PeakIdx)]; % Store the peak in freqdata array
    xlabel('Frequency (in Hz) ->'); % X label
    ylabel('Relative Amplitude ->'); % Y label
    title('Frequency Plot'); % Title
    saveas(gcf, ['./graphs/nofilter_' files(z).name(1:end - 4) '.png']); %
Store the plotted graph
    xlim auto; % Rearrange x axis limits
    plot(t, data); % Plot the time domain graph for the unfiltered signal, just
for debugging
    xlabel('Time (in Sec) ->'); % X label
    ylabel('Amplitude (0.49uV/unit) ->'); % Y label
    title('Wave Plot'); % Title
    saveas(gcf, ['./wave/wave' files(z).name(1:end - 4) '.png']); % Store the
plotted graph

    % Uncomment the following lines to plot the frequency response for the
filtered signal
    % xlim auto; % Rearrange x axis limits
    % Y = fft(filterdata); % Apply the FFT on the filtered data
    % P2 = abs(Y/L); %Absolute function on normalized data, to wrap both sides
    % P1 = P2(2:L/2+1); %Wrap the mirrored FFT data in half
```

```

    % f = Fs*(1:(L/2))/L; %Generate the frequency plot axis data
    % plot(f,P1); %Plot the frequency and normalized/relative amplitude for the
filtered signal
    % xlim([0 30]); %Limit frequency axis to the frequencies we care about
    % xlabel('Frequency (in Hz) ->'); %X label
    % ylabel('Relative Amplitude ->'); %Y label
    % title('Filtered Frequency Plot'); %Title
    % saveas(gcf,['./graphs/filter_' files(z).name(1:end-4) '.png']); %Store
the plotted graph
    % xlim auto; % Rearrange x axis limits
    % plot(t,filterdata); % Plot the time domain graph for the filtered
signal, just for debugging
    % xlabel('Time (in Sec) ->'); %X label
    % ylabel('Amplitude (0.49uV/unit) ->'); %Y label
    % title('Filtered Wave Plot'); %Title
    % saveas(gcf,['./wave/filtered_wave' files(z).name(1:end-4) '.png']);
%Store the plotted graph
    % pause(1);

```

end

Appendix B

PCB LAYOUTS

B.1. 'EASE' Module Layout

Front Layout:

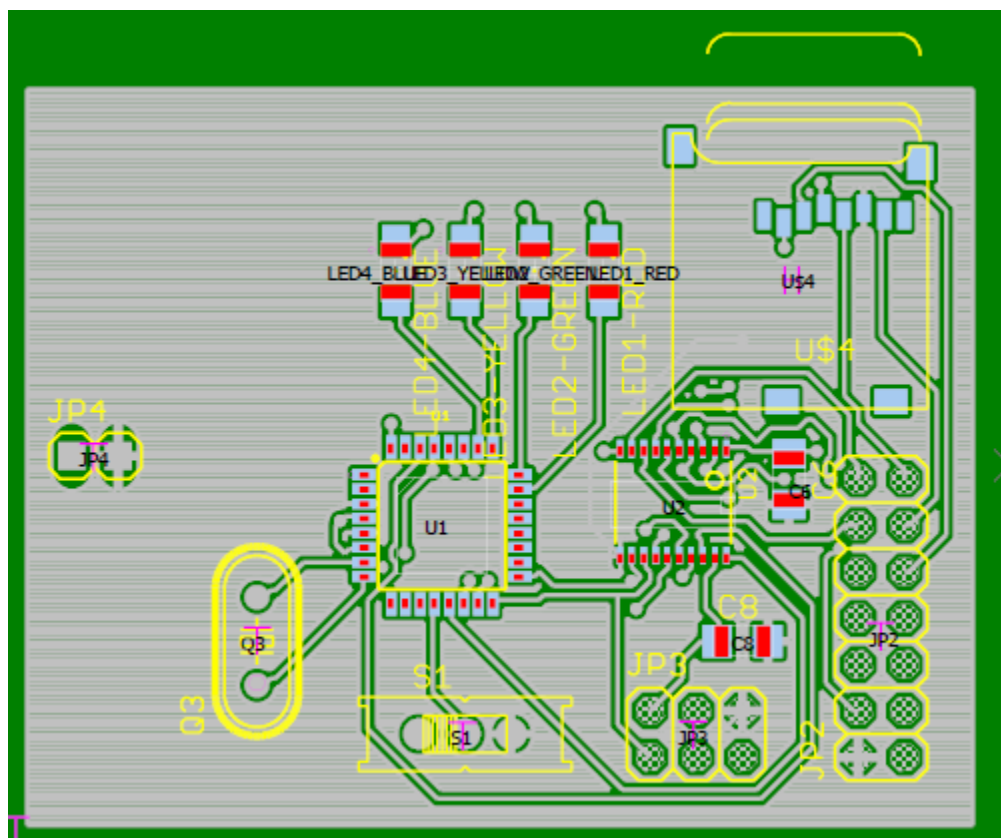


Figure B.1 EASE Module Front Layout.

Part names not shown in the layout (Parts starting with 'C' are SMD ceramic capacitors):

1. U1: Microcontroller, ATmega328PB.
2. U\$4: SD Card Housing.
3. Q3: 16 MHz crystal oscillator.

4. JP4: 2 pin header for input supply.
5. S1: Slide switch.
6. JP3: 6 pin ISP header.
7. JP2: 14 pin EASE module connector header.
8. U2: Level converter IC, TXS0108EPWR.

Back Layout:

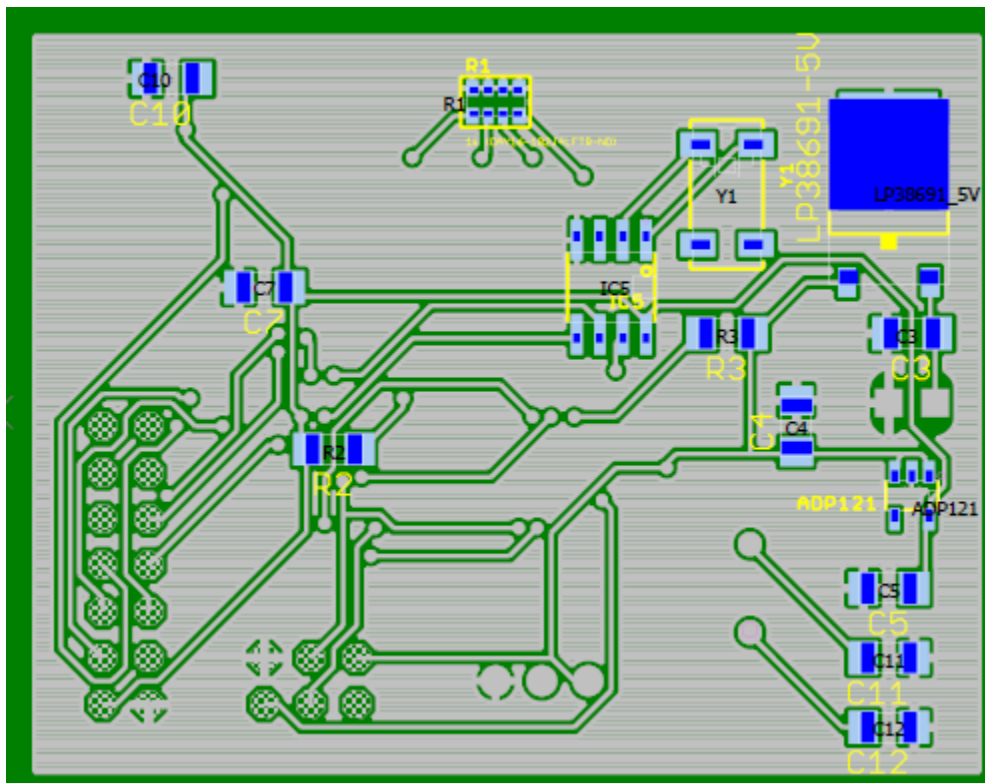


Figure B.2 EASE Module Back Layout.

Part names not shown in the layout (Parts starting with ‘C’ are SMD ceramic capacitors, parts starting with ‘R’ are SMD resistors or resistor pack):

1. IC5: Real Time Clock, PCF8523.
2. Y1: 32.768KHz crystal with load capacitances.

B.2. Docking Station Header Layout

Front Layout:

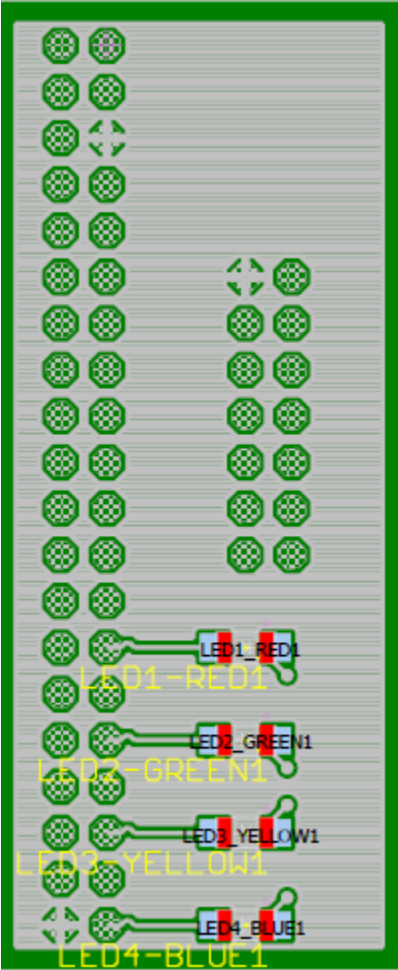


Figure B.3 Docking Station Header Front Layout.

Back Layout:

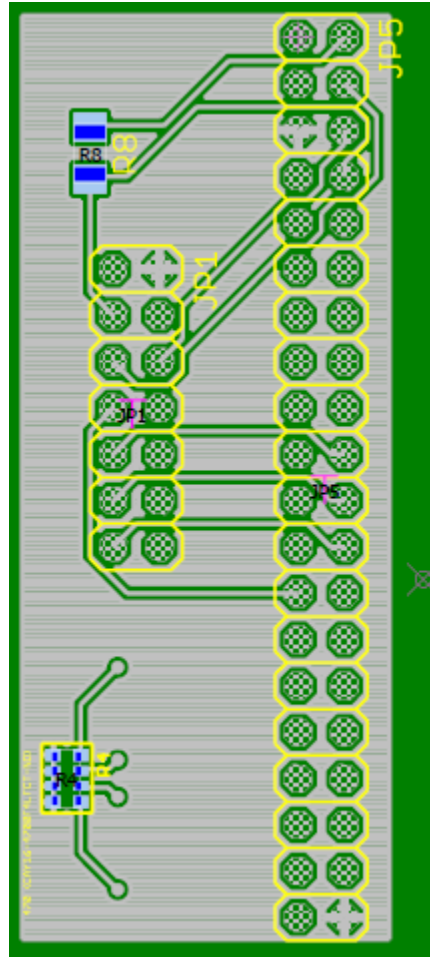


Figure B.4 Docking Station Header Back Layout.

Part names not shown in the layout (Parts starting with ‘R’ are SMD resistors or resistor pack):

1. JP1: 14 Pin right angled connector to EASE module.
2. JP5: 40 Pin female connector for Raspberry Pi pins.

(All the PCB layout designs have been made using EAGLE CAD)