FLINDERS UNIVERSITY

HONOURS THESIS

Performance Augmentation using Biosignals

Author:
Cooper Wolfden

Supervisor:
Associate Professor
Kenneth Pope

A thesis submitted in fulfillment of the requirements
for the degree of Bachelor of Engineering (Electronics) (Honours)

January 4, 2024

DECLARATION

I certify that this thesis:

1. does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university
2. and the research within will not be submitted for any other future degree or diploma without the permission of Flinders University; and
3. to the best of my knowledge and belief, does not contain any material previously published or written by another person except where due reference is made in the text.

Signature of student.....

Print name of student.....Cooper Wolfden.....

Date.....16/10/2023.....

I certify that I have read this thesis. In my opinion it is/is not (please circle) fully adequate, in scope and in quality, as a thesis for the degree of Bachelor of Engineering (Electrical and Electronic) (Honours). Furthermore, I confirm that I have provided feedback on this thesis and the student has implemented it minimally/partially/fully (please circle).

Signature of Principal Supervisor.....Kenneth Pope.....

Print name of Principal Supervisor.....A/Prof. Kenneth Pope.....

Date.....16/10/2023.....

FLINDERS UNIVERSITY

Abstract

College of Science and Engineering

Bachelor of Engineering (Electronics) (Honours)

Performance Augmentation using Biosignals

by Cooper Wolfden

This thesis aims to establish a platform for performers to utilize biological signals in order to create engaging live performances. This project is a continuation of the 'Music from Biosignals' project, with the new addition of lighting control.

The initial phase of the project began with the implementation of the lighting controller, enabling the system to interface with standard performance lighting fixtures. There are two parts of the lighting controller, an off-the-shelf controller, ensuring high reliability, and a prototyping fixture, allowing fast and cost-effective testing. These additions allow for the creation of exciting and engaging lighting patterns from collected biosensors.

Further project developments involved the previously designed hardware, consisting of a PIC32 microcontroller, an ESP32 wireless transmitter, and an ADS1294R analog-to-digital converter. For this hardware, programming inconsistencies were resolved, over-the-air updates were implemented for the ESP32, and hardware debugging was enabled for the PIC32.

Following this, consistent device-to-device communication was established. With successful communication between the PIC32 and ESP32 over SPI, challenges overcome with communication between the ADS1294R and PIC32, and the ESP32 acting as a wireless bridge with the off-body PC via TCP/IP.

Despite these achievements, power issues during the final stage of development caused the device to fail in providing wireless sensor readings for the off-body PC to process. Thorough testing showed that the issue was related to the high impedance of the power plane. In conclusion, while the performance augmentation platform is not fully functional, significant strides have been made in resolving communication issues and establishing a foundation for future iterations. Recommendations for a second version include addressing power distribution challenges and optimizing overall current draw.

Acknowledgements

I would like to thank Craig Dawson for their professional advice and skills that helped immensely in the development of this project. I would like to thank Associate Professor Kenneth Pope for their help in guiding me throughout this project and providing the much needed pressure when necessary. Lastly, I would like to thank my friends and family for providing much needed relief from the various stresses of this year.

Contents

List of Figures

1 Introduction

This project proposes the use of biosignals, such as heart rate and muscle movement, to augment performances by generating correlated music and lighting. For instance, a person could perform a movement routine and have the music of that routine be generated in response to their movement, as opposed to learning a routine based on a preexisting piece of musical composition. Similarly, lighting could be used to enhance a performance by allowing audiences to have a visual representation of the inner working of a performer's body, and how that changes based on the state of the performance and the response of the audience. This opens up room for future exploration into how performance can change when specific movements have additional audio and visual elements, and how different movements may be endowed with novel meaning from these additions.

1.1 Background

A biosignal is a form of communication between biological systems [63], they are used in the body to detect various biological events such as muscle contractions and heartbeats [22]. These signals can be detected using various types of sensors, including electric, mechanical, acoustic, and infrared sensors [34].

Music has been a form of human expression for over 40,000 years [36]. In recent times in western musical expression, the creation of music has relied on the skill and dexterity of artists who have dedicated years to practicing in order to become proficient. This has presented accessibility challenges for individuals who may be unable to physically perform such actions or to those who do not have the time required to learn. This project offers a solution to this challenge by providing a platform for creating music that can be accessible to everyone. Additionally, this project allows for multifaceted performances due to the lack of physical restrictions on performers during the dynamic creation of music.

This project is also beneficial to current artists as the addition of lighting control allows for more engaging performances. Traditionally, lighting control is operated manually by a skilled lighting technician or automatically triggered by sound. This project allows the lighting to be controlled directly by the performer, which could allow for much more compelling lighting setups.

1.2 Aims

The aims of this project are to create a software and hardware platform that can:

- Control music and lighting in real-time from biosignals.
- Operate effectively in live performance spaces.
- Be wearable by a performer for an extended period of time without causing physical distress.

1.3 Literature Review

1.3.1 Introduction

The use of biosignals to generate music and lighting has been an area of exploration and innovation in the field of live performance technology. In this literature review, we will examine various approaches and technologies that have been developed in the pursuit of creating music and lighting from biosignals. We will begin by discussing early attempts at generating music from brainwaves, and the challenges associated with using electroencephalogram (EEG) signals for live performances. Then, we will explore the use of other biosignals such as electrocardiogram (ECG), electromyography (EMG), galvanic skin response (GSR), and respiratory rate, which offer more predictability and discuss why they are better suited to this project. Finally, we will investigate wireless solutions that enable the integration of biosensors into live performance devices, and look into existing biosignal based lighting systems, before delving into the Music from Biosignals project; a project that aims to incorporate biosignals into a wireless platform for live performance. By reviewing these advancements, we hope to gain insights into the current state of the field and identify areas for further improvement and development.

1.3.2 Music from Brainwaves

The earliest attempt at creating music from brain activity is Alvin Lucier's 'Music For Solo Performer' [16][64]. While this system suffers from various technical issues such as high noise, the fundamental issue with trying to use electroencephalogram (EEG) to generate any kind of performance signal is that the output of an EEG is not at all rhythmic and contains a lot of randomness. Additionally, EEG signals have a high potential for artifacting [44] and require a large number of electrodes [58]. These factors make EEG signals unsuitable for performance in a live setting and thus they will not be incorporated into the project.

1.3.3 Rhythmic Signal Approach

Other biosignals that could be used are electrocardiograms (ECG) [3][54], which is a common and painless measurement that is used to monitor the

heart [45]. Electromyography (EMG) [68][76], which measures electrical activity due to the response of muscles [72]. Galvanic skin response (GSR) [38], which can show the intensity of emotional changes due to the change in conductance of the skin [23]. And respiratory rate [10], which measures the number of breaths per minute [73]. These signals have a degree of predictability [67] which makes them better suited for use in this project. There are a number of examples of these kinds of signals being incorporated into live performance settings such as the 'Conductor's Jacket' by Nakra and Picard [48], and 'Stethophone' by Nerness and Fuloria [50]. However, these applications are still limited in their flexibility and use due to the restrictive wired nature of these devices.

1.3.4 Wireless Solutions

Wireless biosensor based performance devices do exist, but there is limited information on them. Examples of such systems are Yamaha AI's 'Transforms a Dancer into a Pianist' [24], and 'Emovere' by Jaimovich [29]. These systems allow performers to elevate their performances by adding an experimental aspect. However, further exploration could still be done in this space with the addition of lighting control.

1.3.5 Biosignal-based Lighting Control

There is limited activity in controlling lighting using biosignals. The most relevant research available is Wang's EMG-based Interactive Control Scheme for Stage Lighting [74]. This project uses EMG signals to control lighting in a live performance environment. However, this project focuses on providing specific control of stage lighting using gestures. For these gestures to work, the feature extraction algorithm of the device needs to be aware of specific gestures, which are predetermined by the developer. This limits possible areas of creative exploration, as lighting compositions are predetermined rather than being 'found' by the performer. Thus, there is still room for further developments in this area.

1.3.6 The Music from Biosignals Project

The music from biosignals project has been an ongoing project that attempts to develop and integrate these previously mentioned gaps in literature. The project has developed an on-body device that acquires biosensors and allows

them to be wirelessly transmitted to a PC for processing [17][71]. Additionally, software developed in MATLAB has been developed that processes the incoming signals and generates music in real-time [13][51]. Prior to the work set out in this thesis, the hardware and software elements of this project had been separate and not yet integrated. This leaves potential future work open in connecting both sides of the project to create one cohesive whole. Additionally, there has only ever been a musical aspect to the project, allowing further exploration into how lighting could improve the system.

1.3.7 Conclusion

In conclusion, the exploration of generating music from biosignals has seen significant progress in recent years. While early attempts using brainwaves faced challenges due to their non-rhythmic and unpredictable nature, other biosignals such as ECG, EMG, GSR, and respiratory rate have shown promise in generating more predictable and rhythmic signals for use in live performance environments. While a number of projects have incorporated these signals, there is still room in the literature for exploration in applying these signals to lighting systems, in a way that does not limit the creativity of the performer.

1.4 Project Plan

1.4.1 Project Objectives

For this project to be successful, it is necessary to fulfill various requirements. The requirements for this project are:

1. The system must implement lighting control
   (a) The protocol for lighting control should be available on at least 80% of performance based lighting fixtures
   (b) The lighting control implementation must be compliant to the protocol
2. The system output must operate remotely to the acquired sensor data at a range of at least 30 m
   (a) The acquisition of sensor data must be detached from the system output to allow for more complicated off-body setups
   (b) The remote system must be compliant with ISO/IEC 15149-1:2014 or equivalent
   (c) The wireless system must have a Packet Error Rate (PER) of no more than 1%
   (d) The wireless system must still be functional in a noisy environment
3. The system must generate rhythmic music from the acquired biosignals
   (a) The musical signals need to be correlated to a consistent beat

1.4.2 Assumptions and Constraints

Assumptions:

- The system will be used in an indoor environment with stable temperature and humidity.
- The performers will be able to wear biosignal sensors comfortably during their performance.
- The sweat from the performers will be mitigated to avoid short-circuiting the electrodes.

- The system will exist in a stable performative environment.
  - The various audio visual elements of the performative environment are functional.
  - The system is being operated by skilled and knowledgeable professionals.

Constraints:

- The total cost of the system cannot exceed $600.
- The size and weight of the whole system must be compact enough to transport in a standard travel bag.
  - The system must weigh less than 20kg.
  - The system must take up less than 50L.
- The size and weight of the on-body subsystem must be wearable for several hours without fatigue.
  - The on-body element of the system must weigh less than 5kg.
- The system must be compatible with a standard lighting protocol.

1.4.3 Previous Work

This project is a continuation of the Music from Biosignals project. As such, a number of these objectives have already been fulfilled, and some hardware and software has been developed. The extent of these developments are as follows:

- Hardware for an on-body device has been developed.
  - One of the microcontrollers has been programmed intermittently.
  - That microcontroller has reported a successful connecting to WiFi.
- Software for the off-body PC has been written.
  - a MATLAB script that generates music using a prerecorded ECG biosignal has been developed.

1.4.4 Scope

Given this project outline and the previous project work. The scope of the project can be defined in Table 1.1, Table 1.2, and Table 1.3. This scope contains stretch goals because some of the previous work had not been completed at the start of the project, as those students were finishing mid-year. Therefore, the stretch goals exist to allow the necessary project flexibility to account for various outcomes of those student's projects.

TABLE 1.1: Project in-scope list

On-body Device	Communication between sub-systems
	Basic sensor reading
Lighting Control	Lighting fixture control
	Test setup
System Integration	Wireless communication
	Integration with music generation
	Integration with lighting generation

TABLE 1.2: Project stretch-goal list

Sensors	Wearable sensor design
MIDI	Compliant MIDI implementation
	MIDI timecode quantisation
	MIDI output timecode
System Integration	Performance application testing
	Bi-directional communication
	Real-time system tunability
	PCB housing design for main board

TABLE 1.3: Project out-of-scope list

Sensors	Sensors as independent systems
MIDI	Wireless MIDI
	MIDI threshold points
Lighting Control	Lighting controller input support
System Integration	Cableless system
	User interface for configuration

2 Lighting Controller

The first element of the system to be developed was the lighting controller. This part of the project had no prior development, so all contributions were done from scratch.

The lighting controller has a simple function in this system; it maps processed biosignals to lighting position and intensity. Therefore, it should be able to communicate with the off-body PC and pass various commands through to the lights connected to it.

During the planning stage of the project, DMX was determined to be the most appropriate protocol to use for this purpose. Additionally, both a off-the-shelf controller, and prototyping fixture are to be developed to give the project robustness, while allowing for cost-effective prototyping.

2.1 Methods

Off-the-shelf controllers were researched and an appropriate controller was chosen. The only device that was within the project's budget while providing an open source driver, for aid in development, was the ENTEC OpenDMX controller.

While the controller was being ordered, a prototyping fixture was developed. The choice of parts used for this fixture came from what was available as to not add additional cost to the project. The fixture was made using an Arduino and eight NeoPixel LEDs.

The Arduino was programmed such that the eight NeoPixels could be controlled via DMX frames. The DMX frame is sent via serial over USB and the Arduino maps intensity, red, green, and blue channels for each individual LED, making a total of four channels per LED.

On the off-body PC, the driver for the ENTEC OpenDMX controller is written in C#. A C# program was written to allow control of the device

from other languages. Additionally, Arduino serial communication was integrated to allow the prototyping fixture to be controlled from the same interface.

2.2 Results

The off-the-shelf controller and custom software allows lighting fixtures to be controlled from other high-level languages in real-time, shown in Figure 2.1. The implementation of this also allows it to be running on a separate PC connected to the network, if lighting control needed to be done away from the main processing, or to free up resources on the main processing PC.

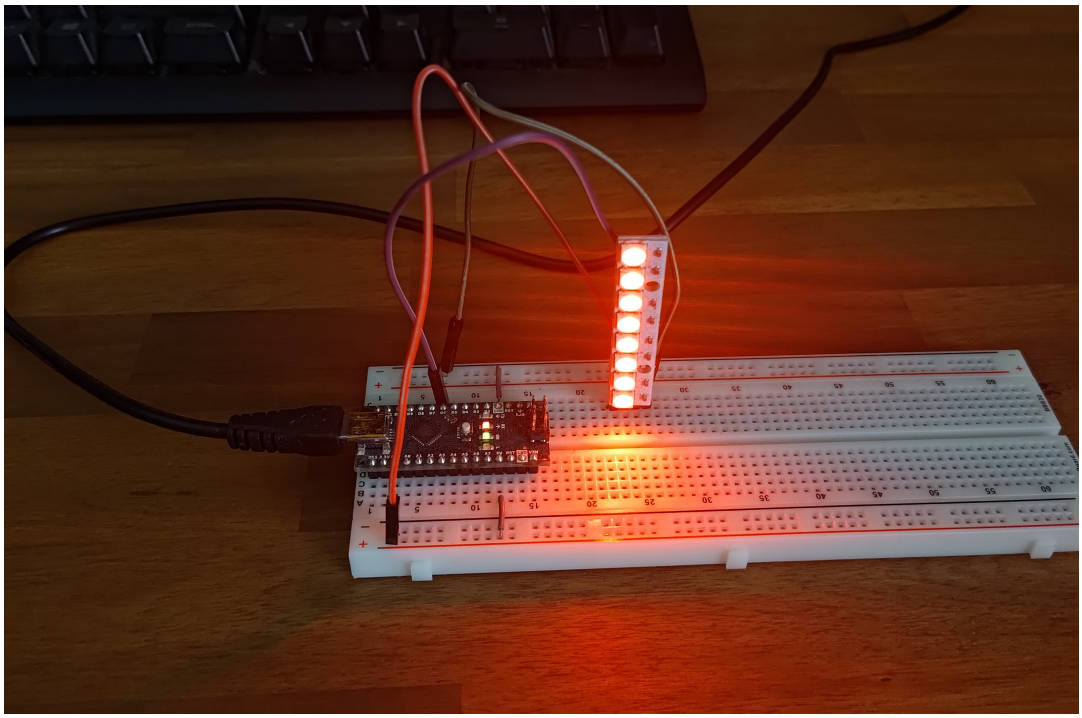
The off-the-shelf controller has a delay of 25 ms. It is capable of driving 512 DMX addresses of any kind.

FIGURE 2.1: Professional lighting fixture control using off-the-shelf controller and custom driver interface

The prototyping fixture uses the exact same interface, allowing it to be interchanged with the off-the-shelf controller, seen in Figure 2.2. Each LED can be controlled independently with red, green, blue, and intensity channels, making it suitable for simulating larger strings of fixtures.

The prototyping fixture communication has an average delay of 5.56 ms. It has a total of 32 address that can be individually controlled. The device has been run for over two hours with no address misalignment.

FIGURE 2.2: Prototyping fixture using the same driver interface



2.3 Discussion

The implementation of lighting control in this manner allows for rapid development of processing algorithms for multiple lights, without needing expensive hardware. This allows performers to fine tune their lighting displays prior the show when they may not have access to the lighting fixtures yet.

Because of the implementation, algorithms that work on the prototyping fixture will also work on the off-the-shelf controller. The only change necessarily is to update the DMX addresses to match the corresponding fixture address.

The biggest challenge with the implementation of the lighting controller was the prototyping fixture. DMX is a protocol that sends 'frames' of 512 bytes continuously. Each byte corresponds to a DMX address, the address number is the byte's offset in the 512 byte array. So, the 12^{th} byte will correspond to the 12^{th} address and the value of that byte will be what is applied to the fixture at that address. Frames are sent out multiple times per second, regardless of if any values in the 512 byte array have changed, so that the fixtures can determine when the controller is unplugged. For the prototyping fixture receiving DMX over serial, there is no simple way to know where the start of the frame is. If it is assumed to be when the device first receives serial

data, it may be misaligned due to differences in when the PC starts sending serial DMX frames and when the device first powers on. Additionally, if data is lost over the transmission, the data will be misaligned with no way of recovering without a reset.

One method to solve this would be to wait for a pause greater than a specific length between characters and use that to mark the beginning of a new transmission. However, if different timings were to be used the code would have to be updated and slight delays in transmission could lead to misaligned.

Another method is to mark the end of transmission with a special character. However, there is no way to distinguish between data and special characters without using an alternative character set. This is because a byte worth of data constitutes all binary values between '00000000' and '11111111'. So there would be no way to determine if binary '01000011' was supposed to be interpreted as the character 'C' or the number 67. There are a number of different character sets, for this portion of the project base64 [65] was chosen. This is because it only contains printable characters, making each string easier to debug, and it is well documented. As this is for a prototyping fixture, it is more important to have it be functional than to be as well optimised as something going into production.

The base64 encoded string represents the entire DMX frame. To reduce unnecessary processing, the frame is reduced to only 32 addresses, which is the number of addresses used by the fixture. Before the frame is base64 encoded, each address concatenated into a single string that represents the entire DMX frame. To make the addresses easier to separate, as well as to ensure both the base64 string and the decoded string are a fixed length, the addresses are sent as a fixed three-digit decimal number. These three-digit numbers are concatenated into a string and base64 encoded before being sent to the Arduino. This makes debugging easier, as the numbers can be directly read as their decimal representation before they are encoded and once they are decoded. Extracting the numbers is simpler because each number has a known position and can be simply converted from a string into a number, and storing the strings is easier as they are constant lengths, so fixed length buffers can be used, which is favourable on a limited memory device like an Arduino.

3 On-Body Device

The on-body device was developed by William Tran in 2022 [71] and consists of a PIC32 as the primary microcontroller and an ESP32 acting as a wireless bridge, as well as ADS1294R 24-bit analog-to-digital converter for taking biosignal measurements.

The ESP32 had been programmed to connect to WiFi, but was only programming intermittently, and the WiFi connection was reported but no data had been written or received.

The only other contributions were from Craig Dawson who supplied information and boilerplate code for programming the PIC32, as well as attaching wires to specific pins of the PIC32, allowing it to be probed using an oscilloscope.

3.1 Methods

The previously programmed ESP32 was verified to determine if data would be able to be sent via WiFi. During this process, inconsistencies in programming were discovered. Over-the-air updates [61] were implemented to make programming more consistent. Then, the device was connected to the off-body PC wirelessly.

The PIC32 was programmed with a basic test program. Then, PLL [43] clock configuration was performed and validated.

SPI communication between the PIC32 and the ESP32 was established. This was determined by connecting both devices to their hardware debuggers and printing characters to be sent (from the PIC32), and that were received (from the ESP32) and verifying that they matched. Further communication between the PIC32 and the off-body PC through the ESP32 was verified using a similar method. Then, the entire communication chain was tested by setting a counter on the PIC32 to zero, transmitting the counter to the off-body PC using the ESP32, and incrementing the counter each time a transmission occurred. A similar counter was set on the off-body PC, which

then received the transmissions, verified both counters matched, and incremented its counter. The time between receiving was measured on the off-body PC, and the average time, multiplied by eight, was calculated as the average bit time. If the transmitted counter and the local counter did not match, the local counter was set to the next valid number that was received, the number of incorrectly received bytes were kept track of, which determined the byte error.

The ADS1294R communication line was probed, and SPI communication was briefly established with the PIC32. Issues surrounding the ADS1294R prompted an investigation into the power distribution of the on-body PCB. Additionally, further issues had arose surrounding power into the ESP32. The PCB was tested in the following states in order to measure changes in how the ESP32 programmed.

- The board was connected to a lab bench power supply with a non-restrictive current limit.
- The boot select switch was held for the extent of the programming cycle.
- The boot select switch was held until programming began.
- The boot select switch was held from when the device was powered on until programming ended.
- The board was powered from a lab bench power supply as well as via USB through an ICD 3.
- All the same boot select switch options were repeated with the additional power being supplied.

3.2 Results

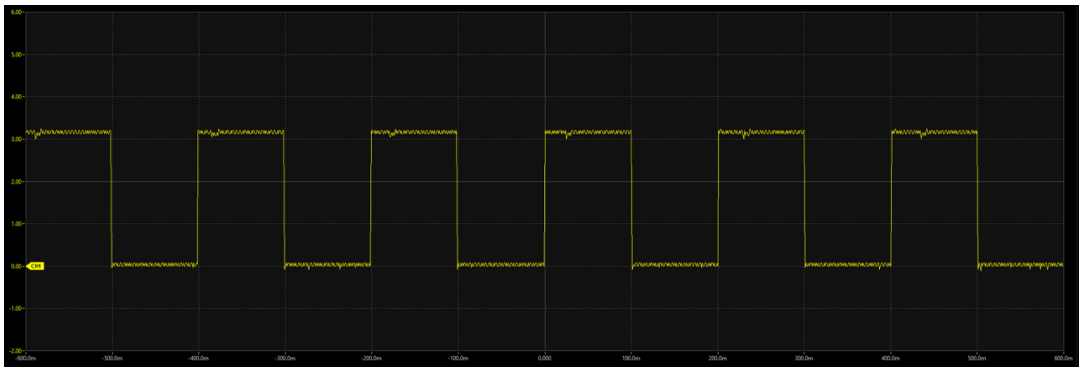
The ESP32 had a programming consistency of less than 50%. After over-the-air updates were implemented, the board successfully received updates more than 90% of the time. The remaining programming inconsistencies come from power supply issues as will be addressed later in this section.

The ESP32 is able to connected to the network and can wirelessly transmit arbitrary bytes of data to the off-body PC consistently at 1 MB/s.

The PIC32 was configured with an 80 MHz system clock and is able to be put into hardware debugging mode. The clock has been verified using an

oscilloscope to measure the time between pin changes using blocking delays, this can be seen in Figure 3.1.

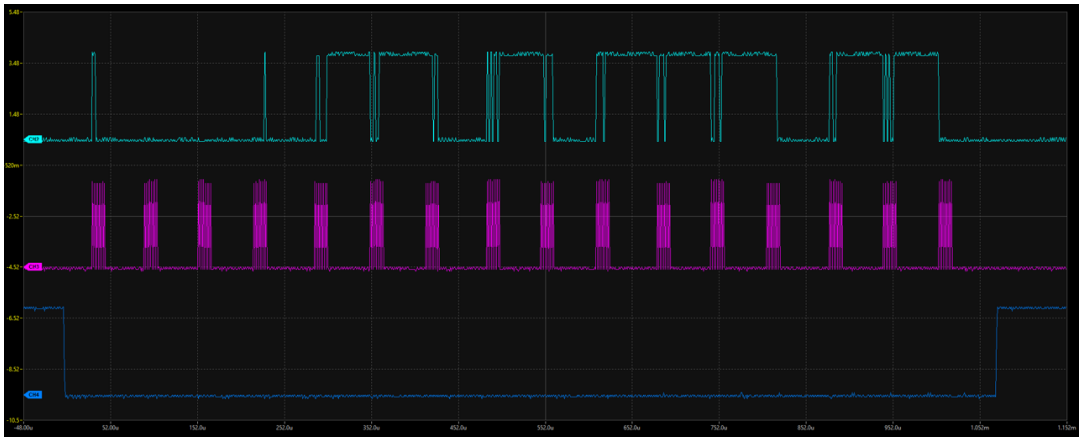
FIGURE 3.1: Oscilloscope measurement of 100ms delay between pulses



The PIC32 and ESP32 are able to communicate with the off-body PC at a speed of 1.45 kbits/s with 0 byte errors over a half hour run-time.

Communication between the ADS1294R and the PIC32 worked, seen in Figure 3.2, and test signals from the ADS1294R were measured. This communication did not last long enough to substantially test, as the PCB began to have power issues.

FIGURE 3.2: Oscilloscope measurement of ADS1294R and PIC32 SPI communication, with the ADS1294R in continuously transmission mode, the top trace showing the ADS1294R response, the middle trace showing the PIC32 clock, and the bottom trace showing the chip select pin assertion



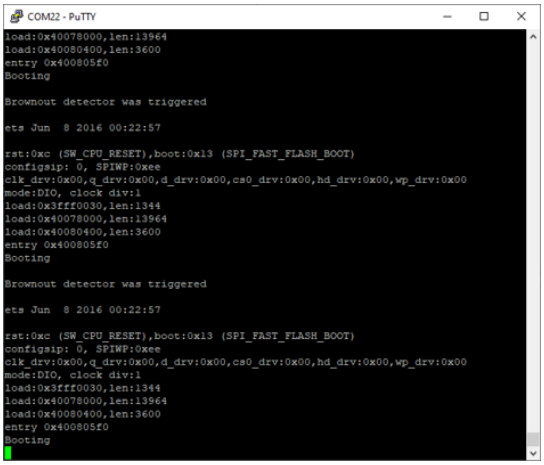
The voltage at the input of the ESP32 was 0.7 V less than what was measured at the input power supply (3.3 V). The input power supply was not reaching it's current limit. Changing the boot select pin did not make a difference in the way that the ESP32 was programmed, until the additional power supply was added. Adding the power supply made the voltage at the ESP32

higher, around 3.1 V. This allows it to be programmed, but it is not completely consistent as sometimes the voltage drops low enough that the device resets.

3.3 Discussion

The inconsistencies regarding the ESP32 programming all revolve around unintentional resetting of the device due to low voltage. When the device is being programmed using the hardware programmer, the device needs to be put into the programming boot mode. To do this, the boot switch should be held down as the device is powered on. Then, once the device is on, the boot switch can be released. In normal operation, the device would stay in programming mode until it is either programmed or reset. However, in this circuit, the ESP32 is often reset due to brownout detection, such a reset can be seen in Figure 3.3.

FIGURE 3.3: ESP32 brownout detection example



This reset would either cause the device to reset again, due to the sudden draw of current of the device turning on being enough to drop the voltage to the point where it would turn off again, or if it did manage to turn on, no longer be in programming mode. Additionally, to get the ESP32 into programming mode required the PCB to be completely unpowered so that the switch could be held as it is powered again. This would also cause issues as all the devices on the PCB would power on at the same time and often the ESP32 would reset at this point, requiring the entire process to be repeated.

The implementation of over-the-air programming fixed the inconsistencies because it does not require the ESP32 to be put into programming mode. This means that the PCB can be powered consistently and all devices can be

powered on completely, with only the ESP32 resetting due to the programming. Which is a lot less likely to cause a substantial voltage drop and trip the brownout detection of the ESP32.

The PIC32 programming was challenging because the device was not correctly labelled on the schematics. Due to chip shortages during manufacturing, a different model of PIC32 had to be chosen, however this was not obviously reflected in the schematics. This meant that the initial implementation of the PIC32 revolved around attempting to connect to a device with a different device ID.

Communication between the PIC32 and the ESP32 was done with SPI. SPI is typically used between a controller and a peripheral. In this case, the PIC32 would be the controller, and the ESP32 would be the wireless transmitter 'peripheral'. However, since both devices are microcontrollers, both devices are configured to be the controller. In order to make the ESP32 the controller, it would need to be continuously sending dummy data in order to generate the clock pulses for the PIC32 to send data, which is non-optimal for this use case as the PIC32 is only ever sending data and the ESP32 is only ever receiving it. So, it makes more sense to make the PIC32 the controller.

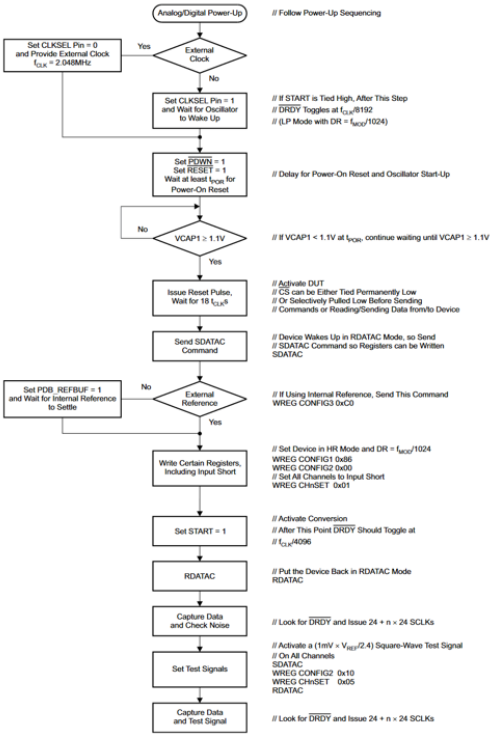
The library that was used to make the ESP32 act as an SPI peripheral is the ESP32DMASPI library [27]. This library is based on the official driver from the manufacture [66]. SPI was implemented using task based DMA receiving. The PIC32 is configured as a 32-bit master operating in SPI mode 3 [6]. The baud-rate generator value has been calculated using the equation $BRG = (F_{PB}/2 \times F_{SCK}) - 1$, and a SPI clock speed of 31.3 kHz has been generated (using a BRG value of 50). This value has been intentionally set low so that the transmission speeds are well below what both devices are capable of for testing.

An additional delay is required between SPI writes in order to stop data from becoming corrupt. This delay was embedded into the low-level driver to make eventual performance optimization centralized, since this delay is a clear cost to performance and by far the cause of the most communication slowdown. This delay is most likely necessary due to the operation of the SPI chip select line. Specifically, the way the chip select line does not reset between SPI transmissions without adequate delays between writes. This could be solved by manually asserting the chip select line instead of allowing the peripheral to control it. However, due to time restrictions it was decided that features should be implemented in a basic functional form, and performance could be optimized once the system was fully developed, and thus

the driver was implemented using significantly slower delay.

The ADS1294R also communicates using SPI. A recommended startup sequence for the device is shown in Figure 3.4. When first communicating with the device, this startup sequence was followed as closely as possible.

FIGURE 3.4: ADS1294R startup sequence

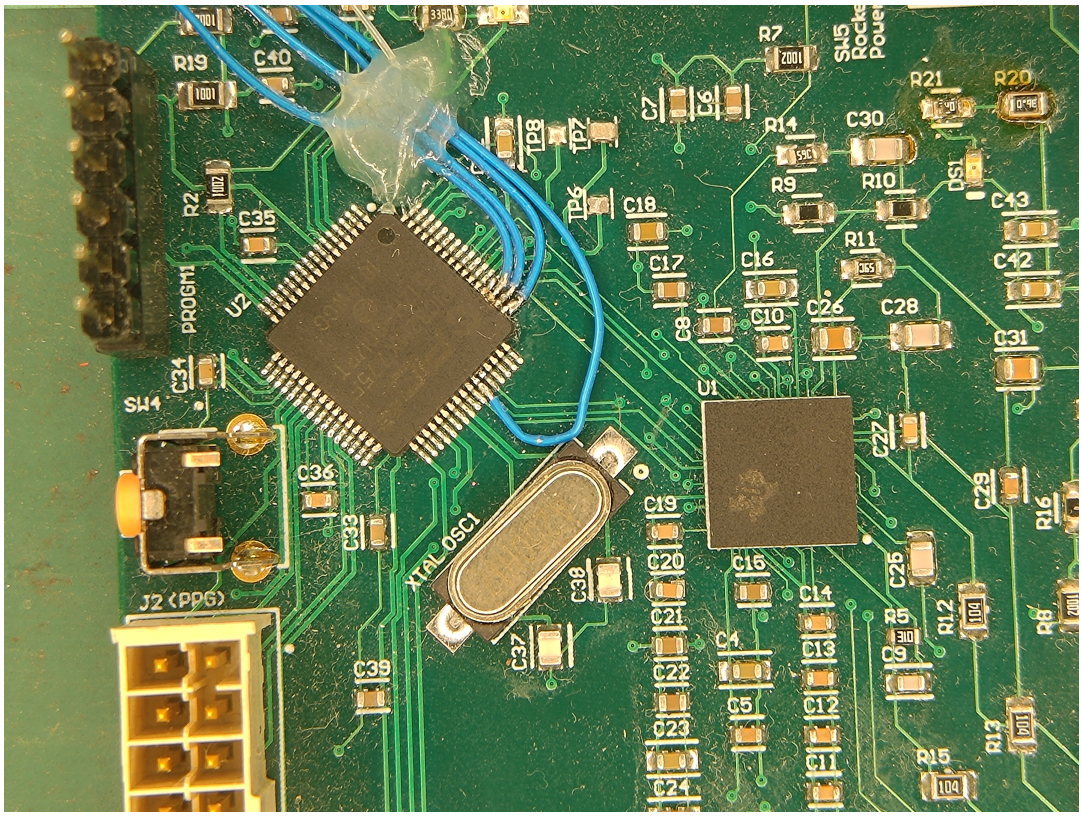


However, the device did not operate as expected. No matter what command the device was given, it would continually output the same byte. The byte did not correlate to the ID of the device, and it did not appear to change, regardless of what commands were sent to the device.

The only commands that the device did respond to were single byte commands. But because the device was responding to the commands, it proved that the communication must work in at least one direction. Still, to aid in troubleshooting, debugging wires were attached to the board, shown in Figure 3.5.

A four channel oscilloscope could then be used to analyze the SPI communication. What was eventually discovered was that the chip select pin was not being asserted for long enough. The SPI communication between the two devices is shown in Figure 3.6. When compared to the working communication, shown in Figure 3.2, the difference is that the chip select pin is not being asserted for long enough. Since the chip select pin was being controlled by the PIC SPI module, it was being automatically driven low during

FIGURE 3.5: PIC32 board attached debugging wires



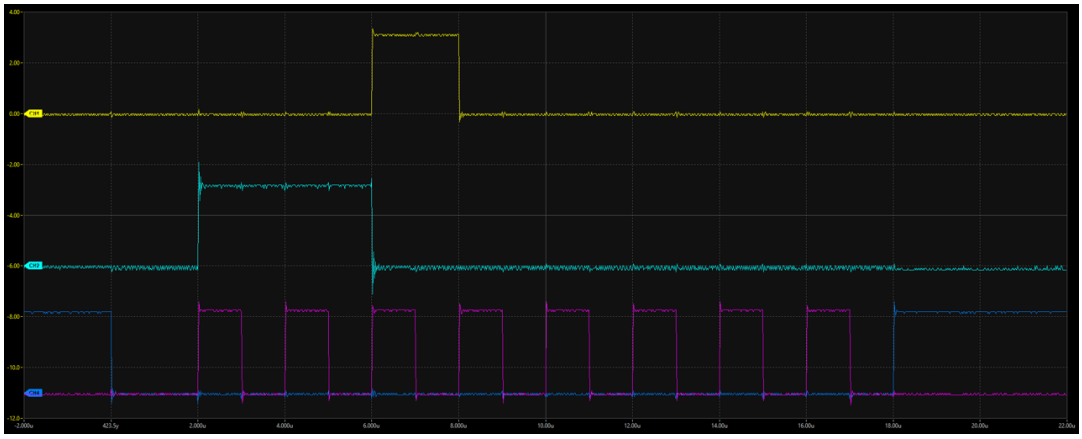
transmission and then high again after transmission completed. For multi-byte commands the chip select line needed to stay asserted until all of the bytes had been transferred and received. That is why it would only work for single-byte commands, because once the byte had been send the communication could be reset.

What is seen in Figure 3.6 is the ADS1294R ignoring the command given by the PIC32 and instead trying to provide the readings for its various ADCs, as continuous data transmission mode is the default mode the device enters on startup. Once the chip select pin is unasserted, the continuous data transmission is reset, and so each time a new command is given from the PIC32, the response is always the same byte, as it is the beginning of the continuous transmission.

The board power distribution issues arose once the SPI communication between the PIC32 and the ADS1294R had been established. There are a number of possibilities for why this would happen.

The first possibility that was invested was the addition of the attached debugging wires, as they may have created a short circuit. However, none of the adjacent pins of any of the debugging wires were power related, nor were they electrically connected as measured using the continuity test function of

FIGURE 3.6: Non-functional SPI communication between PIC32 and ADS1294R, the top trace is the PIC32 command, the middle trace is the ADS1294R response, the bottom two traces are the chip select and clock



a multi-meter.

Another possibility is that the voltage regulator and/or power source are not rated to high enough current. To verify this was not the cause, the voltage regulator was bypassed and the board was directly driven by the source power supply, the feasibility of this was determined by referencing the board schematics. The current limit of the source supply was also increased so that it remained in constant voltage mode, and was monitored to determine that the current limit was never reached.

The final possibility that was investigated was the increase in current draw from all of the integrated circuits now being functional. As the resistance of the power plane of the board is constant, the equation $V = IR$ tells us that an increase in current will cause a proportional increase in voltage drop across the power plane. If the resistance of the power plane is high enough, and the current increase great enough, it is possible that the voltage across the power plane could trigger the brownout detection, or other undefined behaviour, for some of the devices. If this caused a number of devices to reset at the same time, it may cause a peak in current consumption at the point were they all startup again. This would cause a greater voltage drop across the power plane, thus repeating the cycle. One attempt to circumvent this was to insert power into a number of different points on the board closer to where the supply pins of the integrated circuit are. This had some brief success, but ultimately did not provide a consistent method for keeping the device powered correctly, as the devices would intermittently reset

due to power issues. It appeared that the only way to keep the board powered consistently was to keep the debugging tools for each device connected permanently, and to power each device through its corresponding debugger.

This implies that the issues are more related to the source supply rather than the power plane. However, each individual debugger is not able to power the board without it having the same issue. Additionally, several debuggers in parallel still do not solve the issue. It is only the addition of the debugging tools and the source supply that solve the issue. With the debugging tools disconnected, the source supply current reading increases. So, the additions do not provide power that the source supply is unable to supply itself. Unfortunately, due to time constraints, no further investigation could be conducted.

4 Integration

4.1 Methods

The integration of the system was performed by individually implementing each sub-section and ensuring that the sub-section's outputs are compatible with adjacent sub-section inputs.

With this methodology in mind, the system was integrated in the order of outputs to inputs. This is so that with the edition of each new sub-section, the system response to the new sub-section can be verified.

First, the lighting controller was implemented, as shown in the Lighting Controller chapter. Then, software for the off-body PC was written that could communicate with the lighting controller, and control the prototyping fixture based on a prerecorded dataset. The dataset was that of an electrocardiogram sampled at 360 Hz. The lights were controlled by pulsing their intensity at the peaks of the electrocardiogram. Then, the software was modified so that the data being processed could be received as a continuous stream, instead of a fixed dataset.

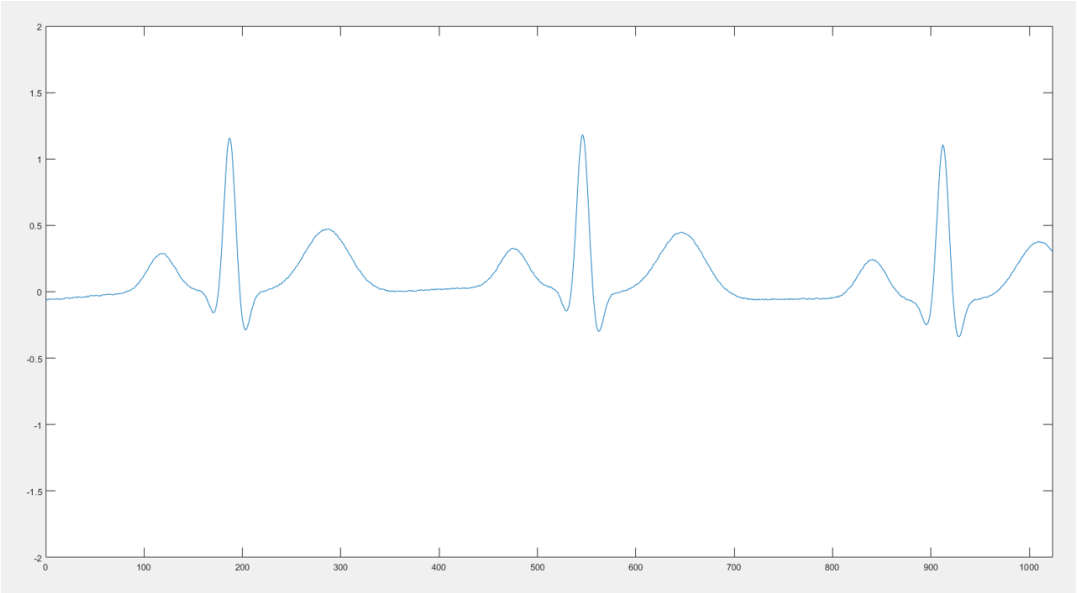
Communication between the on-body device and the off-body PC has already been documented in the On-Body Device chapter. This communication was expanded upon by receiving the data into the same continuous stream that had already been established with the dataset, thus allowing the on-body device to control the prototyping fixture through peaks in the transmitted data.

The on-body device needs to collect different types of data with varying sample and transmission rates. Much like implementing the prototyping fixture for the lighting controller where the end of each DMX frame needed to be marked, the varying data sizes were implemented using special characters to mark the end of data transmission using base64. Additionally, packet headers were created to specify the different kinds of data that are being sent (implementation was prior to the board power distribution issues).

4.2 Results

The off-body PC has a fixed length first-in-first-out buffer that can be filled with either testing data or streamed data from the on-body device. The received data can control the prototyping fixture, with various methods of control through data processing. Beat detection was implemented on the testing dataset, allowing the prototyping fixture to increase intensity at each peak of the input data.

FIGURE 4.1: Fixed length FIFO data buffer with testing dataset



The implementation of varying data sizes and the inclusion of packet headers allows data to be effectively transmitted and received. An unintended benefit of this is that a debug packet header can be specified, and since the data can be variable length, a fully functional 'printf' function on the PIC32 can be used to send arbitrary debugging messages to the off-body PC.

As was presented in the On-Body Device chapter, the PIC32 and ESP32 are able to communicate with the off-body PC at a speed of 1.45 kbits/s. At a sampling rate of 50 Hz [4], the bandwidth is high enough to send 29 bits. For instance, you could send 8-bits of header information and 16-bits of electrocardiogram data, while maintaining sampling and transmission at 50 Hz.

4.3 Discussion

To reduce the amount of hardware that is required to be worn by the performer, the bulk of the processing is to be done on an off-body PC. This PC communicates wirelessly with the on-body device, receiving sensor data from the various biosignals that the device is measuring. The PC then must process the data and coordinate the music and lighting generation.

MATLAB was used as the off-body PC language for processing because of ease of use, versatility, and since it had been previously used on this project.

Since MATLAB functions operate on arrays and matrices, the desired behaviour of our program is to store a length of data and process it all together, rather than try to process each sample individually as it arrives. To keep the system responsive, the buffer is kept at a fixed length, so that over time the processing does not incrementally take longer due to the increase in data to process.

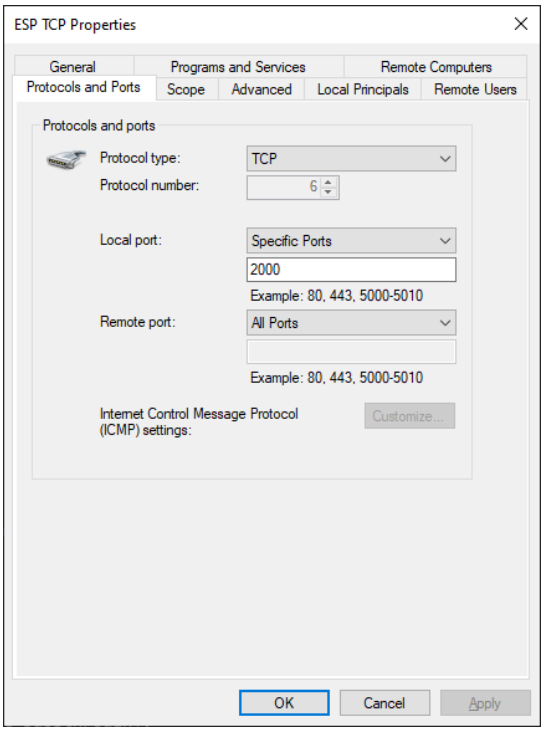
For testing, an electrocardiogram (ECG) data-set from a previous iteration of the project was used. The data-set was imported into the workspace in a way that mimicked wireless reception. Specifically, the data was packetized and shifted into the predefined fixed length buffer. Then, the program loops, continually sending data as new packets, and repeates once it reaches the end of the data-set. This is so that processing methods can be developed experimentally with the assurance that the behaviour will remain consistent once the system transistions away from the testing data-set.

The first processing method that was developed was beats per minute (BPM) prediction using the ECG QRS signal. This method calculates the BPM of the signal by measuring the gaps between the peaks of the QRS signals.

In order to get these devices to communicate, inbound and outbound rules need to be defined in the off-body PC firewall. This is to allow connections on the specified port, since most PC ports are typically closed for security reasons. The port settings can be seen in Figure 4.2.

The implementation of wireless printf debugging has significant benefits as it not only speeds up debugging but also allows debugging in the interrupt service routine, which cannot be debugged with breakpoints. The speed increase of using printf is also seen each time the device is programmed, as the device takes longer to boot when being put into debugging mode. Finally, the inclusion of printf debugging allows messages to be sent beyond debugging, such as status, warning, or error messages. These messages will

FIGURE 4.2: TCP/IP firewall port settings



be received along with the sensor data, potentially giving reason for unexpected behaviour or delays in incoming data.

The speed at which transmission occurs is only fast enough to send a single stream of 24-bit ECG data. While this may be fine for testing, the transmission speed will need to increase significantly if this device is ever intended to be used in an actual performance. As it is likely that a performer would wish to have more than just their heart rate be used to augment the performance.

The final part of the project is the testing and validation. Unfortunately due to power supply issues, a proper testing and validation suite could not be conduceted on the device. The power issues are related to what was seen earlier with the ESP32. As more devices came online, the current draw across on-body device's power plane increased, which subsequently increased the voltage drop. Eventually, this voltage drop became too large and the devices started to lose power. Thus, the testing and validation of the system remains unfinished, as the device was no longer responsive at the point when testing began.

5 Future Work

There are a number of recommendations for future work on the project.

Firstly, the on-body device should be revised. The device that has previously been developed is a good starting point. However, the next revision should include a more prototyping friendly desing (more LEDs, test points, etc.), improved power distribution, and potentially simplified wireless communication, would allow the project to be developed further.

Another aspect of future work that could be undertaken is the development of wearable sensors. There is room for a lot of research and experimentation when it comes to ergonomic sensor design.

Further developments can be made in the software aspect of the project. Generally, the communication speeds between all the devices could be increased. At present, speeds are lower than the maximum capacity of the devices to ensure communication is consistent. There are also a lot of unnecessary overheads on a number of the communication lines. This was to get the devices connected, but further work could be done to optimize this communication.

Lastly, the system could be tested in a real live performance space. There is obviously work that would need to be completed before that would be possible. However, with that work complete, a live performance test with feedback from the performers would be of immense value.

6 Conclusion

This project was a continuation of the Music from Biosignals project. It focused on developing the hardware for biosignal acquisition, and a controller for controlling lighting fixtures. The project was successful in creating the lighting controller, as well as programming the hardware. However, due to previous board design issues, the devices were not able to be powered together. Therefore, while the device-to-device connections on the board were tested and working, the full system integration was not able to be completed.

This project succeeded in demonstrating the design flaws of the on-body device, while also providing a number of recommendations for a future redesign.

Bibliography

[1] Adafruit. *Adafruit NeoPixel Library*. 2023. URL: https://github.com/adafruit/Adafruit_NeoPixel (visited on 10/17/2023).

[2] *ADS129x Low-Power, 8-Channel, 24-Bit Analog Front-End for Biopotential Measurements datasheet (Rev. K)*. 2023.

[3] V.X. Afonso et al. "ECG beat detection using filter banks". In: *IEEE Transactions on Biomedical Engineering* (1999). DOI: 10.1109/10.740882.

[4] Era Ajdaraga and Marjan Gusev. "Analysis of sampling frequency and resolution in ECG signals". In: *2017 25th Telecommunication Forum (TELFOR)*. 2017, pp. 1–4. DOI: 10.1109/TELFOR.2017.8249438.

[5] Mohammadreza Asghari Oskoei and Huosheng Hu. "Myoelectric control systems—A survey". In: *Biomedical Signal Processing and Control* 2.4 (2007), pp. 275–294. ISSN: 1746-8094. DOI: 10.1016/j.bspc.2007.07.009. URL: https://www.sciencedirect.com/science/article/pii/S1746809407000547.

[6] Peter Barry and Patrick Crowley. "Chapter 4 - Embedded Platform Architecture". In: *Modern Embedded Computing*. Ed. by Peter Barry and Patrick Crowley. Boston: Morgan Kaufmann, 2012, pp. 41–97. ISBN: 978-0-12-391490-3. DOI: https://doi.org/10.1016/B978-0-12-391490-3.00004-7. URL: https://www.sciencedirect.com/science/article/pii/B9780123914903000047.

[7] D. Benitez et al. "The use of the Hilbert transform in ECG signal analysis". In: *Computers in Biology and Medicine* 31.5 (2001), pp. 399–406. ISSN: 0010-4825. DOI: 10.1016/S0010-4825(01)00009-9. URL: https://www.sciencedirect.com/science/article/pii/S0010482501000099.

[8] Richard Berry et al. "Use of Chest Wall Electromyography to Detect Respiratory Effort during Polysomnography". In: *Journal of clinical sleep medicine : JCSM : official publication of the American Academy of Sleep Medicine* 12 (2016). DOI: 10.5664/jcsm.6122.

[9] N. Bulusu, J. Heidemann, and D. Estrin. "GPS-less low-cost outdoor localization for very small devices". In: *IEEE Personal Communications* 7.5 (2000), pp. 28–34. DOI: 10.1109/98.878533.

[10] Rovira Carlos et al. "Web-based sensor streaming wearable for respiratory monitoring applications". In: *SENSORS, 2011 IEEE*. 2011, pp. 901–903. DOI: 10.1109/ICSENS.2011.6127168.

[11] CCS Inc. *Light Control through an EtherNet/IP Network*. 2018. URL: https://www.ccs-grp.com/ecsuites/media/download/pamphlet/FL_CN_e.pdf (visited on 03/24/2023).

[12] F.H.Y. Chan et al. "Fuzzy EMG classification for prosthesis control". In: *IEEE Transactions on Rehabilitation Engineering* 8.3 (2000), pp. 305–311. DOI: 10.1109/86.867872.

[13] Chen Chen. "Music from Bio-Signals". Bachelor's Thesis. Flinders University, 2016.

[14] Amer Chlaihawi et al. "Development of printed and flexible dry ECG electrodes". In: *Sensing and Bio-Sensing Research* 20 (2018). DOI: 10.1016/j.sbsr.2018.05.001.

[15] Vinod Chouhan and Sarabjeet Mehta. "Total Removal of Baseline Drift from ECG Signal". In: 2007, pp. 512–515. DOI: 10.1109/ICCTA.2007.126.

[16] Carlos Conceição. *Alvin Lucier - "Music For Solo Performer" (1965)*. https://www.youtube.com/watch?v=bIPU2ynqy2Y. 2010.

[17] Adam De Pierro. "Music from Biosignals". Bachelor's Thesis. Flinders University, 2019.

[18] Digital Illumination Interface Alliance. *Introducing DALI*. 2021. URL: https://www.dali-alliance.org/dali/ (visited on 03/24/2023).

[19] Gilles Dubost and Atau Tanaka. "A Wireless, Network-based Biosensor Interface for Music". In: (2002).

[20] EnOcean. *Smart Lighting*. 2023. URL: https://www.enocean.com/en/applications/building-automation/lighting/ (visited on 03/24/2023).

[21] ENTTEC. *Open DMX USB*. 2023. URL: https://www.enttec.com/product/dmx-usb-interfaces/open-dmx-usb/ (visited on 10/16/2023).

[22] Monty Escabí. "Chapter 11 - Biosignal Processing". In: *Introduction to Biomedical Engineering (Third Edition)* (2012), pp. 667–746.

[23] Bryn Farnsworth. "What is GSR (galvanic skin response) and how does it work?" In: *Research Fundamentals* (2018).

[24] Yamaha Global. *Yamaha Artificial Intelligence (AI) Transforms a Dancer into a Pianist*. 2018. URL: https://www.yamaha.com/en/news_release/2018/18013101/ (visited on 05/18/2023).

[25] Xuhong Guo et al. "A Self-Wetting Paper Electrode for Ubiquitous Bio-Potential Monitoring". In: *IEEE Sensors Journal* 17.9 (2017), pp. 2654–2661. DOI: 10.1109/JSEN.2017.2684825.

[26] Matthias Hertel et al. *DMXSerial*. 2022. URL: https://github.com/mathertel/DMXSerial (visited on 10/17/2023).

[27] hideakitai. *ESP32DMASPI*. 2023. URL: https://github.com/hideakitai/ESP32DMASPI (visited on 10/16/2023).

[28] M. K. Islam et al. "Study and Analysis of ECG Signal Using MATLAB & LABVIEW as Effective Tools". In: *International Journal of Computer and Electrical Engineering* 4.3 (2012).

[29] Javier Jaimovich. *Emovere: Designing Sound Interactions for Biosignals and Dancers*. Tech. rep. Departamento de Música y Sonología Universidad de Chile, 2016.

[30] Javier Jaimovich and R. Benjamin Knapp. "Creating Biosignal Algorithms for Musical Applications from an Extensive Physiological Database". In: *Proceedings of the International Conference on New Interfaces for Musical Expression* (2015), pp. 1–4. URL: https://hdl.handle.net/10919/80529.

[31] Reema Jain and Vijay Kumar Garg. "EMG Classification Using Nature-Inspired Computing and Neural Architecture". In: *2021 9th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*. 2021, pp. 1–5. DOI: 10.1109/ICRITO51393.2021.9596077.

[32] Sarang L. Joshi, Rambabu A Vatti, and Rupali V. Tornekar. "A Survey on ECG Signal Denoising Techniques". In: *2013 International Conference on Communication Systems and Network Technologies*. 2013, pp. 60–64. DOI: 10.1109/CSNT.2013.22.

[33] J.M. Kahn and J.R. Barry. "Wireless infrared communications". In: *Proceedings of the IEEE* 85.2 (1997), pp. 265–298. DOI: 10.1109/5.554222.

[34] Eugenijus Kaniusas. *Biomedical Signals and Sensors I*. Springer Berlin, 2012.

[35] Kirti Khunti. "Accurate interpretation of the 12-lead ECG electrode placement: A systematic review". In: *Health Education Journal* 73.5 (2014), pp. 610–623. DOI: 10.1177/0017896912472328.

[36] Anton Killin. "The origins of music: Evidence, theory, and prospects". In: *Music & Science* 1 (2018), p. 1. DOI: 10.1177/2059204317751971. URL: https://doi.org/10.1177/2059204317751971.

[37] Neeraj Kumar, Imteyaz Ahmad, and Pankaj Rai. "Signal Processing of ECG Using Matlab". In: *International Journal of Scientific and Research Publications* 2 (2012).

[38] Hindra Kurniawan, Alexandr V. Maslov, and Mykola Pechenizkiy. "Stress detection from speech and Galvanic Skin Response signals". In: *Proceedings of the 26th IEEE International Symposium on Computer-Based Medical Systems*. 2013, pp. 209–214. DOI: 10.1109/CBMS.2013.6627790.

[39] T.V. Lakshman and U. Madhow. "The performance of TCP/IP for networks with high bandwidth-delay products and random loss". In: *IEEE/ACM Transactions on Networking* 5.3 (1997), pp. 336–350. DOI: 10.1109/90.611099.

[40] Lightology. *What is 0-10V Dimming?* 2023. URL: https://www.lightology.com/index.php?module=tools_faq_0_10v_control (visited on 03/24/2023).

[41] Yuan-Pin Lin et al. "EEG-Based Emotion Recognition in Music Listening". In: *IEEE Transactions on Biomedical Engineering* 57.7 (2010), pp. 1798–1806. DOI: 10.1109/TBME.2010.2048568.

[42] Oscar Luna, Daniel Torres, and RTAC Americas. *DMX512 Protocol Implementation Using MC9S08GT60 8-Bit MCU*. 2006. URL: https://www.nxp.com/docs/en/application-note/AN3315.pdf (visited on 03/24/2023).

[43] Shilpi Maji, Supantha Mandal, and Suraj Kumar Saw. "Phase Locked Loop - A Review". In: *INTERNATIONAL JOURNAL OF ENGINEERING RESEARCH & TECHNOLOGY (IJERT)* 4 (2 2016).

[44] Malik Muhammad Naeem Mannan, Muhammad Ahmad Kamran, and Myung Yung Jeong. "Identification and Removal of Physiological Artifacts From Electroencephalogram Signals: A Review". In: *IEEE Access* 6 (2018), pp. 30630–30652. DOI: 10.1109/ACCESS.2018.2842082.

[45] Mayo Foundation for Medical Education and Research. *Electrocardiogram (ECG or EKG)*. 2023. URL: https://www.mayoclinic.org/tests-procedures/ekg/about/pac-20384983 (visited on 10/17/2023).

[46] Modbus Organization, Inc. *Modbus*. 2023. URL: https://www.modbus.org/ (visited on 03/24/2023).

[47] Mohsen Naji, Mohammad Firoozabadi, and Parviz Azadfallah. "Emotion classification based on forehead biosignals using support vector machines in music listening". In: *2012 IEEE 12th International Conference on Bioinformatics & Bioengineering (BIBE)*. 2012, pp. 396–400. DOI: 10.1109/BIBE.2012.6399657.

[48] Teresa Nakra and Rosalind Picard. "The "Conductor's Jacket": A Device For Recording Expressive Musical Gestures". In: (1998).

[49] Amine Naït-Ali, ed. *Advanced Biosignal Processing*. 1st ed. Berlin, Heidelberg: Springer, 2009.

[50] Barbara Nerness and Anika Fuloria. *Stethophone by Barbara Nerness and Anika Fuloria*. 2019. URL: https://ccrma.stanford.edu/~afuloria/stethophone.html (visited on 04/02/2023).

[51] Isaac Nicholls. "Music with Bio-signals". Bachelor's Thesis. Flinders University, 2019.

[52] Miguel Ortiz et al. "Biosignal-driven Art: Beyond biofeedback". In: (2011).

[53] Joonas Paalasmaa, David J. Murphy, and Ove Holmqvist. "Analysis of Noisy Biosignals for Musical Performance". In: *Advances in Intelligent Data Analysis XI*. Ed. by Jaakko Hollmén, Frank Klawonn, and Allan Tucker. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 241–252. ISBN: 978-3-642-34156-4.

[54] Jiapu Pan and Willis J. Tompkins. "A real-time QRS detection algorithm". In: *IEEE Transactions on Biomedical Engineering* BME-32.3 (1985), pp. 230–236. DOI: 10.1109/TBME.1985.325532.

[55] Alexandros Pantelopoulos and Nikolaos G. Bourbakis. "A Survey on Wearable Sensor-Based Systems for Health Monitoring and Prognosis". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 40.1 (2010), pp. 1–12. DOI: 10.1109/TSMCC.2009.2032660.

[56] Hailong Peng et al. "Preparation of photonic-magnetic responsive molecularly imprinted microspheres and their application to fast and selective extraction of 17B-estradiol". In: *Journal of Chromatography A* 1442 (2016), pp. 1–11. ISSN: 0021-9673. DOI: 10.1016/j.chroma.2016.03.

003. URL: https://www.sciencedirect.com/science/article/pii/S0021967316302308.

[57] *PIC32MX5XX/6XX/7XX Family Data Sheet*. 2019.

[58] Marek Piorecky et al. "Dream Activity Analysis in Low-number Electrode EEG: A Methodology Proposal". In: *2019 E-Health and Bioengineering Conference (EHB)*. 2019, pp. 1–4. DOI: 10.1109/EHB47216.2019.8969949.

[59] RDM Task Group. *RDMnet*. 2023. URL: https://www.rdmprotocol.org/rdm/rdmnet/ (visited on 03/24/2023).

[60] E.M. Royer and Chai-Keong Toh. "A review of current routing protocols for ad hoc mobile wireless networks". In: *IEEE Personal Communications* 6.2 (1999), pp. 46–55. DOI: 10.1109/98.760423.

[61] A. S.A.Quadri and B. Othman Sidek. "An Introduction to Over-the-Air Programming in Wireless Sensor Networks". In: *International journal of Computer Science & Network Solutions* 2.2 (2014). ISSN: 2345-3397.

[62] Science Buddies. *Engineering Design Process*. 2023. URL: https://www.sciencebuddies.org/science-fair-projects/engineering-design-process/engineering-design-process-steps (visited on 04/02/2023).

[63] John Semmlow. "Chapter 1 - The Big Picture: Bioengineering Signals and Systems". In: *Circuits, Signals and Systems for Bioengineers (Third Edition)* (2018), pp. 3–50.

[64] Volker Straebel and Wilm Thoben. "Alvin Lucier's Music for Solo Performer: Experimental music beyond sonification". In: *Organised Sound* 19 (2014), pp. 17–29. DOI: 10.1017/S135577181300037X.

[65] Isnar Sumartono, Andysah Putera Utama Siahaan, and Arpan Arpan. "Base64 Character Encoding and Decoding Modeling". In: *International Journal of Recent Trends in Engineering & Research* 2 (Dec. 2016), pp. 63–68.

[66] Espressif Systems. *SPI Slave Driver*. 2023. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/spi_slave.html#spi-slave-driver (visited on 10/16/2023).

[67] Koray Tahiroğlu, Hannah Drayson, and Cumhur Erkut. "An Interactive Bio-Music Improvisation System". In: 2008.

[68] Atau Tanaka and R. Knapp. "Multimodal Interaction in Music Using the Electromyogram and Relative Position Sensing". In: 2002.

[69] The IES Controls Protocol Committee. *Lighting Control Protocols*. Illuminating Engineering Society of North America, 2011.

[70] Matthew John Thies and Bruce Pennycook. *Controlling game music in real time with biosignals*. Tech. rep. The University of Texas at Austin, 2012.

[71] William Tran. "Music from Biosignals". Bachelor's Thesis. Flinders University, 2022.

[72] The Johns Hopkins University, The Johns Hopkins Hospital, and Johns Hopkins Health System. *Electromyography (EMG)*. 2023. URL: https://www.hopkinsmedicine.org/health/treatment-tests-and-therapies/electromyography-emg (visited on 10/17/2023).

[73] The Johns Hopkins University, The Johns Hopkins Hospital, and Johns Hopkins Health System. *Vital Signs (Body Temperature, Pulse Rate, Respiration Rate, Blood Pressure)*. 2023. URL: https://www.hopkinsmedicine.org/health/conditions-and-diseases/vital-signs-body-temperature-pulse-rate-respiration-rate-blood-pressure (visited on 10/17/2023).

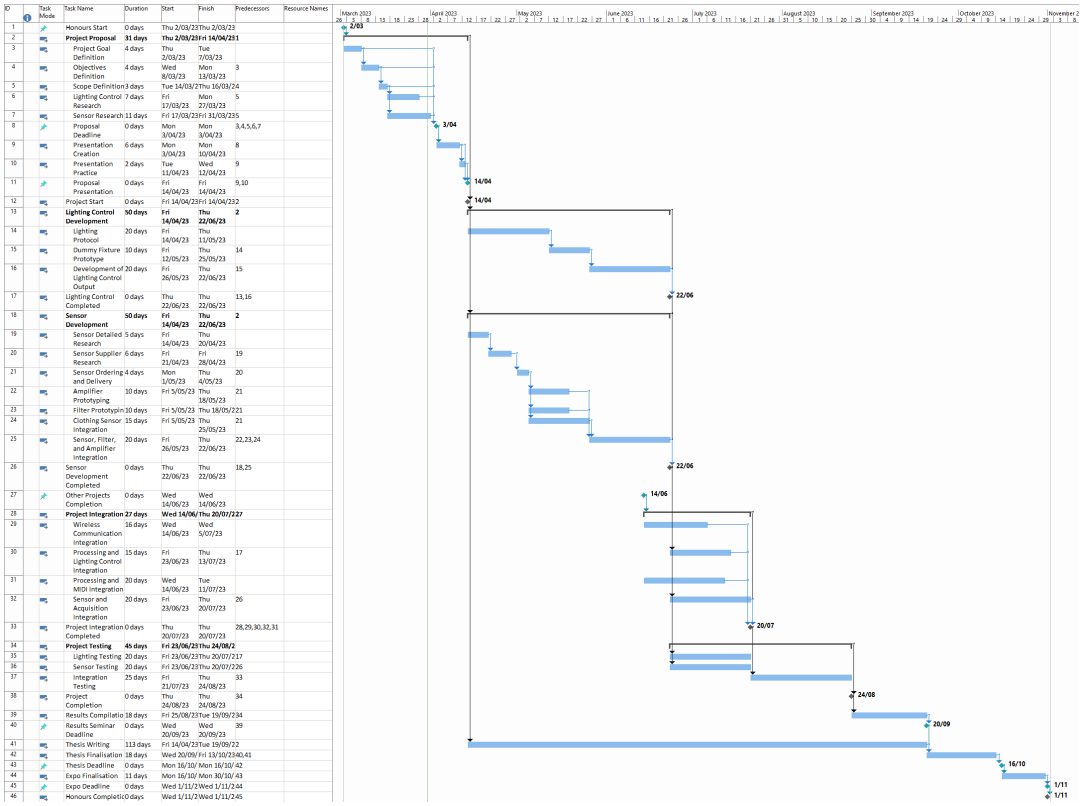
[74] Huiqin Wang and Jiajun Li. "EMG-based Interactive Control Scheme for Stage Lighting". In: *2022 International Conference on Culture-Oriented Science and Technology (CoST)*. 2022, pp. 288–291. DOI: 10.1109/CoST57098.2022.00066.

[75] Li Da Xu, Wu He, and Shancang Li. "Internet of Things in Industries: A Survey". In: *IEEE Transactions on Industrial Informatics* 10.4 (2014), pp. 2233–2243. DOI: 10.1109/TII.2014.2300753.

[76] Aaron J. Young et al. "Classification of Simultaneous Movements Using Surface EMG Pattern Recognition". In: *IEEE Transactions on Biomedical Engineering* 60.5 (2013), pp. 1250–1258. DOI: 10.1109/TBME.2012.2232293.

[77] zencontrol. *BACnet*. 2022. URL: https://zencontrol.com/bacnet/ (visited on 03/24/2023).

A Gantt Chart



ID	Task Mode	Task Name	Duration	Start	Finish	Predecessors	Resource Names
1		Honours Start	0 days	Thu 2/03/23	Thu 2/03/23		
2		**Project Proposal**	**31 days**	**Thu 2/03/23**	**Fri 14/04/23**	**1**	
3		Project Goal Definition	4 days	Thu 2/03/23	Tue 7/03/23		
4		Objectives Definition	4 days	Wed 8/03/23	Mon 13/03/23	3	
5		Scope Definition	3 days	Tue 14/03/23	Thu 16/03/23	4	
6		Lighting Control Research	7 days	Fri 17/03/23	Mon 27/03/23	5	
7		Sensor Research	11 days	Fri 17/03/23	Fri 31/03/23	5	
8		Proposal Deadline	0 days	Mon 3/04/23	Mon 3/04/23	3,4,5,6,7	
9		Presentation Creation	6 days	Mon 3/04/23	Mon 10/04/23	8	
10		Presentation Practice	2 days	Tue 11/04/23	Wed 12/04/23	9	
11		Proposal Presentation	0 days	Fri 14/04/23	Fri 14/04/23	9,10	
12		Project Start	0 days	Fri 14/04/23	Fri 14/04/23	2	
13		**Lighting Control Development**	**50 days**	**Fri 14/04/23**	**Thu 22/06/23**	**2**	
14		Lighting Protocol	20 days	Fri 14/04/23	Thu 11/05/23		
15		Dummy Fixture Prototype	10 days	Fri 12/05/23	Thu 25/05/23	14	
16		Development of Lighting Control Output	20 days	Fri 26/05/23	Thu 22/06/23	15	
17		Lighting Control Completed	0 days	Thu 22/06/23	Thu 22/06/23	13,16	
18		**Sensor Development**	**50 days**	**Fri 14/04/23**	**Thu 22/06/23**	**2**	
19		Sensor Detailed Research	5 days	Fri 14/04/23	Thu 20/04/23		
20		Sensor Supplier Research	6 days	Fri 21/04/23	Fri 28/04/23	19	
21		Sensor Ordering and Delivery	4 days	Mon 1/05/23	Thu 4/05/23	20	
22		Amplifier Prototyping	10 days	Fri 5/05/23	Thu 18/05/23	21	
23		Filter Prototyping	10 days	Fri 5/05/23	Thu 18/05/23	21	
24		Clothing Sensor Integration	15 days	Fri 5/05/23	Thu 25/05/23	21	
25		Sensor, Filter, and Amplifier Integration	20 days	Fri 26/05/23	Thu 22/06/23	22,23,24	
26		Sensor Development Completed	0 days	Thu 22/06/23	Thu 22/06/23	18,25	
27		Other Projects Completion	0 days	Wed 14/06/23	Wed 14/06/23		
28		**Project Integration**	**27 days**	**Wed 14/06/23**	**Thu 20/07/23**	**27**	
29		Wireless Communication Integration	16 days	Wed 14/06/23	Wed 5/07/23		
30		Processing and Lighting Control Integration	15 days	Fri 23/06/23	Thu 13/07/23	17	
31		Processing and MIDI Integration	20 days	Wed 14/06/23	Tue 11/07/23		
32		Sensor and Acquisition Integration	20 days	Fri 23/06/23	Thu 20/07/23	26	
33		Project Integration Completed	0 days	Thu 20/07/23	Thu 20/07/23	28,29,30,32,31	
34		**Project Testing**	**45 days**	**Fri 23/06/23**	**Thu 24/08/23**		
35		Lighting Testing	20 days	Fri 23/06/23	Thu 20/07/23	17	
36		Sensor Testing	20 days	Fri 23/06/23	Thu 20/07/23	26	
37		Integration Testing	25 days	Fri 21/07/23	Thu 24/08/23	33	
38		Project Completion	0 days	Thu 24/08/23	Thu 24/08/23	34	
39		Results Compilation	18 days	Fri 25/08/23	Tue 19/09/23	34	
40		Results Seminar Deadline	0 days	Wed 20/09/23	Wed 20/09/23	39	
41		Thesis Writing	113 days	Fri 14/04/23	Tue 19/09/23	2	
42		Thesis Finalisation	18 days	Wed 20/09/23	Fri 13/10/23	40,41	
43		Thesis Deadline	0 days	Mon 16/10/23	Mon 16/10/23	42	
44		Expo Finalisation	11 days	Mon 16/10/23	Mon 30/10/23	43	
45		Expo Deadline	0 days	Wed 1/11/23	Wed 1/11/23	44	
46		Honours Complation	0 days	Wed 1/11/23	Wed 1/11/23	45	

B Prototyping Fixture Code

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <Adafruit_NeoPixel.h>
#include <Base64.h>

#define PIXELS_PIN 6
#define NUM_PIXELS 8
#define ADDR_PER_PIXEL 4
#define DMX_LENGTH (NUM_PIXELS * ADDR_PER_PIXEL)
#define UNIVERSE_SIZE 32

#define INPUT_BUFFER_LENGTH 1024
#define BASE64_LENGTH 128
#define RAW_LENGTH 95

Adafruit_NeoPixel pixels(NUM_PIXELS, PIXELS_PIN, NEO_GRB +
   NEO_KHZ800);
uint8_t dmx_values[UNIVERSE_SIZE];

char input_buffer[INPUT_BUFFER_LENGTH];
char decode_buffer[BASE64_LENGTH];
uint16_t input_index = 0;
uint8_t new_data = 0;

void setup() {
  Serial.begin(115200);
  pixels.begin();
  pixels.clear();
  pixels.show();
}

void serialEvent() {
  while (Serial.available()) {
    if (input_index > INPUT_BUFFER_LENGTH) { input_index = 0; }
       // Should never hit this
```

```
    char c = Serial.read();

    if (c == '\n') {
      memset(decode_buffer, '\0', BASE64_LENGTH);
      memcpy(decode_buffer, input_buffer, input_index);
      new_data = 1;
      input_index = 0;
      continue;
    }

    input_buffer[input_index++] = c;
  }
}

void loop() {
  if (new_data) {
    new_data = 0;

    int decoded_length = Base64.decodedLength(decode_buffer,
       BASE64_LENGTH);
    char decoded_string[decoded_length];
    Base64.decode(decoded_string, decode_buffer, BASE64_LENGTH);

    int array_index = 0;
    for (int i = 0; i < decoded_length; i += 3) {
      char val[3];
      val[0] = (decoded_string[i+0]);
      val[1] = (decoded_string[i+1]);
      val[2] = (decoded_string[i+2]);

      dmx_values[array_index++] = atoi(val);
    }

    for (int addr = 0; addr < DMX_LENGTH; addr += ADDR_PER_PIXEL
       ) {
      uint8_t i = dmx_values[addr + 0];
      uint8_t r = dmx_values[addr + 1];
      uint8_t g = dmx_values[addr + 2];
      uint8_t b = dmx_values[addr + 3];

      uint32_t color = pixels.Color(
        (r * i) >> 8,
        (g * i) >> 8,
        (b * i) >> 8
      );
```

```
      pixels.setPixelColor(addr / ADDR_PER_PIXEL, color);
    }

    pixels.show();

  }
}
```

C DMX Server Code

```
using System;
using System.Runtime.InteropServices;
using System.IO;
using System.Threading;

namespace DMXServer
{

    public class OpenDMX

    {

        public static byte[] buffer = new byte[513];
        public static uint handle;
        public static bool done = false;
        public static int bytesWritten = 0;
        public static FT_STATUS status;
        public static Thread thread;

        public const byte BITS_8 = 8;
        public const byte STOP_BITS_2 = 2;
        public const byte PARITY_NONE = 0;
        public const UInt16 FLOW_NONE = 0;
        public const byte PURGE_RX = 1;
        public const byte PURGE_TX = 2;



        [DllImport("FTD2XX.dll")]
        public static extern FT_STATUS FT_Open(UInt32 uiPort,
           ref uint ftHandle);
        [DllImport("FTD2XX.dll")]
        public static extern FT_STATUS FT_Close(uint ftHandle);
        [DllImport("FTD2XX.dll")]
```

```
public static extern FT_STATUS FT_Read(uint ftHandle,
   IntPtr lpBuffer, UInt32 dwBytesToRead, ref UInt32
   lpdwBytesReturned);
[DllImport("FTD2XX.dll")]
public static extern FT_STATUS FT_Write(uint ftHandle,
   IntPtr lpBuffer, UInt32 dwBytesToRead, ref UInt32
   lpdwBytesWritten);
[DllImport("FTD2XX.dll")]
public static extern FT_STATUS FT_SetDataCharacteristics
   (uint ftHandle, byte uWordLength, byte uStopBits,
   byte uParity);
[DllImport("FTD2XX.dll")]
public static extern FT_STATUS FT_SetFlowControl(uint
   ftHandle, char usFlowControl, byte uXon, byte uXoff)
   ;
[DllImport("FTD2XX.dll")]
public static extern FT_STATUS FT_GetModemStatus(uint
   ftHandle, ref UInt32 lpdwModemStatus);
[DllImport("FTD2XX.dll")]
public static extern FT_STATUS FT_Purge(uint ftHandle,
   UInt32 dwMask);
[DllImport("FTD2XX.dll")]
public static extern FT_STATUS FT_ClrRts(uint ftHandle);
[DllImport("FTD2XX.dll")]
public static extern FT_STATUS FT_SetBreakOn(uint
   ftHandle);
[DllImport("FTD2XX.dll")]
public static extern FT_STATUS FT_SetBreakOff(uint
   ftHandle);
[DllImport("FTD2XX.dll")]
public static extern FT_STATUS FT_GetStatus(uint
   ftHandle, ref UInt32 lpdwAmountInRxQueue, ref UInt32
    lpdwAmountInTxQueue, ref UInt32 lpdwEventStatus);
[DllImport("FTD2XX.dll")]
public static extern FT_STATUS FT_ResetDevice(uint
   ftHandle);
[DllImport("FTD2XX.dll")]
public static extern FT_STATUS FT_SetDivisor(uint
   ftHandle, char usDivisor);


public static void start()
{
    handle = 0;
    status = FT_Open(0, ref handle);
    thread = new Thread(new ThreadStart(writeData));
```

```
        thread.Start();
        setDmxValue(0, 0);  //Set DMX Start Code
    }

    public static void stop()
    {
        if (thread != null && thread.ThreadState !=
            ThreadState.Stopped)
        {
            thread.Abort();
        }

        status = FT_Close(handle);
    }

    public static void setDmxValue(int channel, byte value)
    {
        if (buffer != null)
        {
            buffer[channel + 1] = value;
        }
    }

    public static void writeData()
    {
        while (!done)
        {
            initOpenDMX();
            FT_SetBreakOn(handle);
            FT_SetBreakOff(handle);
            bytesWritten = write(handle, buffer, buffer.
                Length);
            Thread.Sleep(20);
        }

    }

    public static int write(uint handle, byte[] data, int
        length)
    {
        IntPtr ptr = Marshal.AllocHGlobal((int)length);
        Marshal.Copy(data, 0, ptr, (int)length);
        uint bytesWritten = 0;
        status = FT_Write(handle, ptr, (uint)length, ref
            bytesWritten);
        return (int)bytesWritten;
```

```
        }

        public static void initOpenDMX()
        {
            status = FT_ResetDevice(handle);
            status = FT_SetDivisor(handle, (char)12);  // set
                baud rate
            status = FT_SetDataCharacteristics(handle, BITS_8,
                STOP_BITS_2, PARITY_NONE);
            status = FT_SetFlowControl(handle, (char)FLOW_NONE,
                0, 0);
            status = FT_ClrRts(handle);
            status = FT_Purge(handle, PURGE_TX);
            status = FT_Purge(handle, PURGE_RX);
        }

    }

    /// <summary>
    /// Enumaration containing the varios return status for the
        DLL functions.
    /// </summary>
    public enum FT_STATUS
    {
        FT_OK = 0,
        FT_INVALID_HANDLE,
        FT_DEVICE_NOT_FOUND,
        FT_DEVICE_NOT_OPENED,
        FT_IO_ERROR,
        FT_INSUFFICIENT_RESOURCES,
        FT_INVALID_PARAMETER,
        FT_INVALID_BAUD_RATE,
        FT_DEVICE_NOT_OPENED_FOR_ERASE,
        FT_DEVICE_NOT_OPENED_FOR_WRITE,
        FT_FAILED_TO_WRITE_DEVICE,
        FT_EEPROM_READ_FAILED,
        FT_EEPROM_WRITE_FAILED,
        FT_EEPROM_ERASE_FAILED,
        FT_EEPROM_NOT_PRESENT,
        FT_EEPROM_NOT_PROGRAMMED,
        FT_INVALID_ARGS,
        FT_OTHER_ERROR
    };

}
```

```
using System;
using System.Threading;
using System.IO.Ports;
using System.Linq;

namespace DMXServer
{
    public class ArduinoDMX
    {
        static SerialPort port;
        static Thread thread;

        static byte[] buffer = new byte[8*4];

        public void start()
        {
            if (port != null && port.IsOpen)
            {
                port.Close();
            }

            port = new SerialPort("COM25", 115200);
            port.Open();

            thread = new Thread(new ThreadStart(writeData));
            thread.Start();
        }

        public void stop()
        {
            if (thread != null && thread.ThreadState !=
               ThreadState.Stopped)
            {
                thread.Abort();
            }

            if (port != null && port.IsOpen)
            {
                port.Close();
            }
        }

        public void setDmxValue(int channel, byte value)
        {
            if (buffer != null)
            {
```

```
                if (channel < buffer.Length)
                {
                    buffer[channel] = value;
                }
            }
        }

        public void writeData()
        {
            while (true)
            {
                char[] end_char = { '\n' };

                string raw = string.Concat(buffer.Select(x => x.
                    ToString("000")));
                raw = raw.Remove(raw.Length - 1, 1);
                string b64 = Base64Encode(raw);

                port.Write(b64);
                port.Write(end_char, 0, 1);

                Thread.Sleep(20);
            }
        }

        public static string Base64Encode(string plainText)
        {
            var plainTextBytes = System.Text.Encoding.UTF8.
                GetBytes(plainText);
            return System.Convert.ToBase64String(plainTextBytes)
                ;
        }

        public void WriteArray(byte[] buffer)
        {
            for (int i = 0; i < buffer.Length; i++)
            {
                Console.Write(buffer[i]);
                Console.Write("␣");
            }
        }
    }
}

using System;
using System.IO;
```

```
using System.Net;
using System.Net.Sockets;
using System.Security.AccessControl;
using System.Security.Principal;
using System.Text;
using System.Windows.Input;

namespace DMXServer
{
    internal class Program
    {

        static void Main(string[] args)
        {
            int num_leds = 8;
            int num_channels = 4;
            //TcpListener server = null;
            ArduinoDMX arduino_dmx = new ArduinoDMX();
            //OpenDMX open_dmx = new OpenDMX();

            arduino_dmx.start();
            //Int32 port = 13000;
            //IPAddress localAddr = IPAddress.Parse("127.0.0.1")
                ;

            //server = new TcpListener(localAddr, port);
            //server.Start();

            //TcpClient client = server.AcceptTcpClient();
            //Console.WriteLine("CLIENT CONNECTED");
            //NetworkStream stream = client.GetStream();

            //while (!client.Connected) ;
            /*
            while (client.Connected)
            {
                int buf_len;
                byte[] buf = new byte[1024];

                if ((buf_len = stream.Read(buf, 0, buf.Length))
                    > 0)
                {
                    string str = Encoding.UTF8.GetString(buf, 0,
                        buf_len).ToString();
                    int channel = int.Parse(str.Split(new char[]
                        { '=' })[0]);
```

```
                byte value = byte.Parse(str.Split(new char[]
                    { '=' })[1]);
                arduino_dmx.setDmxValue(channel, value);
            }

        }
        */

        for (int i = 0; i < (num_leds * num_channels); i +=
            num_channels) { arduino_dmx.setDmxValue(i, 200);
            }

        while (true)
        {
            char key = Console.ReadKey().KeyChar;
            if (key == 'u') { for (int i = 0; i < (num_leds
                * num_channels); i += num_channels) {
                arduino_dmx.setDmxValue(1 + i, 255); } }
            if (key == 'j') { for (int i = 0; i < (num_leds
                * num_channels); i += num_channels) {
                arduino_dmx.setDmxValue(1 + i, 200); } }
            if (key == 'n') { for (int i = 0; i < (num_leds
                * num_channels); i += num_channels) {
                arduino_dmx.setDmxValue(1 + i, 100); } }
            if (key == 'k') { for (int i = 0; i < (num_leds
                * num_channels); i += num_channels) {
                arduino_dmx.setDmxValue(1 + i, 0); } }
            if (key == 'r') { for (int i = 0; i < (num_leds
                * num_channels); i += num_channels) {
                arduino_dmx.setDmxValue(3 + i, 170); } }
            if (key == 'f') { for (int i = 0; i < (num_leds
                * num_channels); i += num_channels) {
                arduino_dmx.setDmxValue(3 + i, 80); } }
            if (key == 'v') { for (int i = 0; i < (num_leds
                * num_channels); i += num_channels) {
                arduino_dmx.setDmxValue(3 + i, 30); } }
            if (key == 'd') { for (int i = 0; i < (num_leds
                * num_channels); i += num_channels) {
                arduino_dmx.setDmxValue(3 + i, 0); } }
        }

        arduino_dmx.stop();
    }
  }
}
```

D MATLAB TCP/IP Receiver Code

```
clear server

server = tcpserver("0.0.0.0", 2000);
configureTerminator(server, "LF");
configureCallback(server, "terminator", @server_callback);

function server_callback(src, ~)
    disp("-----------------------")
    base64 = readline(src);
    decoded = matlab.net.base64decode(char(base64));
    bytes = uint8(transpose(decoded));
end
```

E ESP32 Transmitter Code

```
#include <WiFi.h>
#include <ESPmDNS.h>
#include <WiFiUdp.h>
#include <ArduinoOTA.h>
#include <ESP32DMASPISlave.h>

#include <string.h>
#include <stdint.h>
#include "util.h"

#define SERVER_IP "10.1.1.187"
#define SERVER_PORT 2000
#define BUFFER_LENGTH 128

uint8_t* buffer;

ESP32DMASPI::Slave slave;
WiFiClient client;

constexpr uint8_t CORE_TASK_SPI_SLAVE {0};
constexpr uint8_t CORE_TASK_PROCESS_BUFFER {0};

static TaskHandle_t task_handle_wait_spi = 0;
static TaskHandle_t task_handle_process_buffer = 0;

void task_wait_spi(void* pvParameters) {
  while (1) {
    ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
    slave.wait(buffer, BUFFER_LENGTH);
    xTaskNotifyGive(task_handle_process_buffer);
  }
}

void task_process_buffer(void* pvParameters) {
  while (1) {
    ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
```

```
    if (client.connected()) { client.write(buffer, slave.
        available()); }
    slave.pop();
    xTaskNotifyGive(task_handle_wait_spi);
  }
}

void setup() {
  Serial.begin(115200);
  Serial.println("Booting");

  buffer = slave.allocDMABuffer(BUFFER_LENGTH);
  delay(1000);

  slave.setDataMode(SPI_MODE0);
  slave.setMaxTransferSize(BUFFER_LENGTH);
  slave.begin(HSPI);

  xTaskCreatePinnedToCore(task_wait_spi, "task_wait_spi", 2048,
      NULL, 2, &task_handle_wait_spi, CORE_TASK_SPI_SLAVE);
  xTaskNotifyGive(task_handle_wait_spi);
  xTaskCreatePinnedToCore(task_process_buffer, "
      task_process_buffer", 2048, NULL, 2, &
      task_handle_process_buffer, CORE_TASK_PROCESS_BUFFER);

  WiFi.mode(WIFI_STA);
  WiFi.begin(SSID, PASS);

  while (WiFi.waitForConnectResult() != WL_CONNECTED) {
    Serial.printf("Connection␣Failed!␣Rebooting...␣\r\n");
    delay(5000);
    ESP.restart();
  }

  OTA_setup();
  ArduinoOTA.begin();
}

void loop() {
  ArduinoOTA.handle();

  if (!client.connected()) {
    if (client.connect(SERVER_IP, SERVER_PORT)) {
      Serial.printf("Connected␣to␣%s␣on␣tcp␣port␣%u␣\r\n",
          SERVER_IP, SERVER_PORT);
    }
```

```
    else {
      Serial.printf("Failed␣to␣connect␣to␣%s␣on␣tcp␣port␣%u␣\r\n
          ", SERVER_IP, SERVER_PORT);
      delay(1000);
    }
  }
```

F PIC32 Code

```
// PIC32MX775F512H Configuration Bit Settings

// 'C' source line config statements

// DEVCFG3
// USERID = No Setting
#pragma config FSRSSEL = PRIORITY_7     // SRS Select (SRS
    Priority 7)
#pragma config FMIIEN = OFF             // Ethernet RMII/MII
    Enable (RMII Enabled)
#pragma config FETHIO = OFF             // Ethernet I/O Pin
    Select (Alternate Ethernet I/O)
#pragma config FCANIO = OFF             // CAN I/O Pin Select (
    Alternate CAN I/O)
#pragma config FUSBIDIO = ON            // USB USID Selection (
    Controlled by the USB Module)
#pragma config FVBUSONIO = ON           // USB VBUS ON Selection
     (Controlled by USB Module)

// DEVCFG2
#pragma config FPLLIDIV = DIV_10        // PLL Input Divider (10
    x Divider)
#pragma config FPLLMUL = MUL_16         // PLL Multiplier (16x
    Multiplier)
#pragma config UPLLIDIV = DIV_6         // USB PLL Input Divider
     (6x Divider)
#pragma config UPLLEN = ON              // USB PLL Enable (
    Enabled)
#pragma config FPLLODIV = DIV_8         // System PLL Output
    Clock Divider (PLL Divide by 8)

// DEVCFG1
#pragma config FNOSC = PRIPLL           // Oscillator Selection
    Bits (Primary Osc w/PLL (XT+,HS+,EC+PLL))
```

```
#pragma config FSOSCEN = OFF            // Secondary Oscillator
    Enable (Disabled)
#pragma config IESO = OFF               // Internal/External
    Switch Over (Disabled)
#pragma config POSCMOD = HS             // Primary Oscillator
    Configuration (HS osc mode)
#pragma config OSCIOFNC = OFF           // CLKO Output Signal
    Active on the OSCO Pin (Disabled)
#pragma config FPBDIV = DIV_1           // Peripheral Clock
    Divisor (Pb_Clk is Sys_Clk/1)
#pragma config FCKSM = CSDCMD           // Clock Switching and
    Monitor Selection (Clock Switch Disable, FSCM Disabled)
#pragma config WDTPS = PS1048576        // Watchdog Timer
    Postscaler (1:1048576)
#pragma config FWDTEN = OFF             // Watchdog Timer Enable
     (WDT Disabled (SWDTEN Bit Controls))

// DEVCFG0
#pragma config DEBUG = OFF              // Background Debugger
    Enable (Debugger is disabled)
#pragma config ICESEL = ICS_PGx1        // ICE/ICD Comm Channel
    Select (ICE EMUC1/EMUD1 pins shared with PGC1/PGD1)
#pragma config PWP = OFF                // Program Flash Write
    Protect (Disable)
#pragma config BWP = OFF                // Boot Flash Write
    Protect bit (Protection Disabled)
#pragma config CP = OFF                 // Code Protect (
    Protection Disabled)

// #pragma config statements should precede project file
    includes.
// Use project enums instead of #define for ON and OFF.

#include <xc.h>
#include "user.h"

int main (void) {
    init();

    while (1) {
        run();
    }

    // Should never reach this
    return 0;
}
```

```
#ifndef _SINE_H
#define _SINE_H

#ifdef __cplusplus
extern "C" {
#endif

void init(void);
void run(void);

#ifdef __cplusplus
}
#endif

#endif /* _SINE_H */
#ifndef _SINE_H
#define _SINE_H

#ifdef __cplusplus
extern "C" {
#endif

void init(void);
void run(void);

#ifdef __cplusplus
}
#endif

#endif /* _SINE_H */


#include <xc.h>
#include <stdint.h>
#include <string.h>

#include "user.h"
#include "util.h"
#include "ESP32.h"
#include "ADS1294R.h"

struct {
    uint32_t ECG:16;
    uint32_t RSP:16;
    uint32_t EMG:16;
```

```
    uint32_t BPM:16;
} packet;

ChannelData ch;

void init() {
//    ADC_init();
    ESP32_init();
    ADS1294R_init();
}

void run() {
//    if (data_ready()) {
//        read_data(&ch);
//        debug(
//            "HEADER: 0x%06X \n"
//            "CH1: %u \n"
//            "CH2: %u \n"
//            "CH3: %u \n"
//            "CH4: %u \n",
//            ch.HEAD, ch.CH1, ch.CH2, ch.CH3, ch.CH4
//        );
//    }
    debug("A");
//    delay(500);
}


#ifndef _UTIL_H_
#define _UTIL_H_

#include <xc.h>

// Print 8-bit binary number using printf
// usage: debug("NUM: "BYTE_TO_BINARY_PATTERN, BYTE_TO_BINARY(
    binary_number));
#define BYTE_TO_BINARY_PATTERN "%c%c%c%c%c%c%c%c"
#define BYTE_TO_BINARY(byte)  \
  ((byte) & 0x80 ? '1' : '0'), \
  ((byte) & 0x40 ? '1' : '0'), \
  ((byte) & 0x20 ? '1' : '0'), \
  ((byte) & 0x10 ? '1' : '0'), \
  ((byte) & 0x08 ? '1' : '0'), \
  ((byte) & 0x04 ? '1' : '0'), \
  ((byte) & 0x02 ? '1' : '0'), \
  ((byte) & 0x01 ? '1' : '0')
```

```
// SYS_FREQ = CRYSTAL_FREQ / 10 * 16 / 8
//#define SYS_FREQ 5000000

// May not be exactly actually since scope says different but
   close enough
#define DELAY_CONST 2.5

void delay_us(unsigned int us) {
    // Convert microseconds us into how many clock ticks it will
        take
    // Doing this causes an overflow I think... better to
       precalc and put in magic number
//    us *= SYS_FREQ / 1000000 / 2;   // Core Timer updates
   every 2 ticks
    us *= DELAY_CONST;
    _CP0_SET_COUNT(0);              // Set Core Timer count to 0
    while (us > _CP0_GET_COUNT());  // Wait until Core Timer
       count reaches the number we calculated earlier
}

void delay_ms(int ms) {
    delay_us(ms * 1000);
}

void delay(int ms) {
    delay_ms(ms);
}

void ADC_init() {
    AD1CON1bits.ADSIDL = 0;
    AD1CON1bits.SIDL = 0;
    AD1CON1bits.ASAM = 1;   // auto sampling
    AD1CON1bits.CLRASAM = 0; // overwrite buffer
    AD1CON1bits.FORM = 0b000; // integer 16-bit output
    AD1CON1bits.SSRC = 0b111; // auto convert
    AD1CON1bits.ADON = 1;
    AD1CON1bits.ON = 1;
    AD1CON1bits.SAMP = 1;

    AD1CHSbits.CH0SA = 0b1111;
    AD1CHSbits.CH0NA = 0;
    AD1CHSbits.CH0SB = 0b0000;
    AD1CHSbits.CH0NB = 0;
}
#endif // _UTIL_H_
```

```
#ifndef _ESP32_H_
#define _ESP32_H_

#include <xc.h>
#include <stdint.h>
#include <stdio.h>
#include <stdarg.h>
#include <string.h>

uint32_t ESP32_SPI_write(uint32_t data) {
    // Low-level SPI driver
    SPI2BUF = data;                 // Place data we want to
       send in SPI buffer
    while(!SPI2STATbits.SPITBE);    // Wait until sent status
       bit is cleared
    uint32_t read = SPI2BUF;        // Read data from buffer to
       clear it

    delay_us(5000);
    return read;
}

void ESP32_SPI_write_4byte(uint8_t b1, uint8_t b2, uint8_t b3,
   uint8_t b4) {
    uint32_t word = ((uint32_t)b1 << 24) | ((uint32_t)b2 << 16)
       | ((uint32_t)b3 << 8) | (uint32_t)b4;
    ESP32_SPI_write(word);
}

void ESP32_SPI_write_byte(uint8_t data) {
    ESP32_SPI_write_4byte(data, 0, 0, 0);
}

void ESP32_SPI_write_array(uint8_t *array, size_t len) {
    for (size_t i = 0; i < len; i++) {
        ESP32_SPI_write_byte(array[i]);
    }
}

void write_packet(uint8_t* buf, size_t len) {
    uint8_t mod_table[] = {0, 2, 1};
    char encoding_table[] = {   'A', 'B', 'C', 'D', 'E', 'F', 'G
       ', 'H',
                                'I', 'J', 'K', 'L', 'M', 'N', 'O
                                   ', 'P',
```

```
                                    'Q', 'R', 'S', 'T', 'U', 'V', 'W
                                        ', 'X',
                                    'Y', 'Z', 'a', 'b', 'c', 'd', 'e
                                        ', 'f',
                                    'g', 'h', 'i', 'j', 'k', 'l', 'm
                                        ', 'n',
                                    'o', 'p', 'q', 'r', 's', 't', 'u
                                        ', 'v',
                                    'w', 'x', 'y', 'z', '0', '1', '2
                                        ', '3',
                                    '4', '5', '6', '7', '8', '9', '+
                                        ', '/'
    };

    size_t output_length = 4 * ((len + 2) / 3);
    char encoded_data[output_length];

    for (int i = 0, j = 0; i < len;) {
        uint32_t octet_a = i < len ? buf[i++] : 0;
        uint32_t octet_b = i < len ? buf[i++] : 0;
        uint32_t octet_c = i < len ? buf[i++] : 0;

        uint32_t triple = (octet_a << 0x10) + (octet_b << 0x08)
            + octet_c;

        encoded_data[j++] = encoding_table[(triple >> 3 * 6) & 0
            x3F];
        encoded_data[j++] = encoding_table[(triple >> 2 * 6) & 0
            x3F];
        encoded_data[j++] = encoding_table[(triple >> 1 * 6) & 0
            x3F];
        encoded_data[j++] = encoding_table[(triple >> 0 * 6) & 0
            x3F];
    }

    for (int i = 0; i < mod_table[len % 3]; i++) {
        encoded_data[output_length - 1 - i] = '=';
    }

    ESP32_SPI_write_array(encoded_data, output_length);
    ESP32_SPI_write_byte('\n');
}

void debug(const char *fmt, ...) {
    va_list args;
    char str[1024];
```

```
    va_start(args, fmt);
    vsprintf(str, fmt, args);
    va_end(args);

    write_packet(str, strlen(str));
}

void ESP32_IO_init() {
    TRISBbits.TRISB2 = 0;       // Set ESP32 EN pin as output
    PORTBbits.RB2 = 1;          // Set ESP32 EN pin high
}

void ESP32_SPI_init() {
    SPI2CONbits.ON = 0;         // Turn off SPI2 before
       configuring
    SPI2CONbits.FRMEN = 0;      // Framed SPI Support (SS pin
       used)
    SPI2CONbits.MSSEN = 1;      // Slave Select Enable (SS
       driven during transmission)
    SPI2CONbits.ENHBUF = 0;     // Enhanced Buffer Enable (
       disable enhanced buffer)
    SPI2CONbits.SIDL = 1;       // Stop in Idle Mode
    SPI2CONbits.DISSDO = 0;     // Disable SDOx (pin is
       controlled by this module)
    SPI2CONbits.MODE32 = 1;     // Use 32-bit mode
    SPI2CONbits.MODE16 = 0;     // Do not use 16-bit mode
    SPI2CONbits.SMP = 0;        // Input data is sampled at the
       end of the clock signal
    SPI2CONbits.CKE = 1;        // Data is shifted out/in on
       transition from idle (high) state to active (low) state
    SPI2CONbits.SSEN = 1;       // Slave Select Enable (SS pin
       used by module)
    SPI2CONbits.CKP = 0;        // Clock Polarity Select (clock
       signal is active low, idle state is high)
    SPI2CONbits.MSTEN = 1;      // Master Mode Enable
    SPI2CONbits.STXISEL = 0b01; // SPI Transmit Buffer Empty
       Interrupt Mode (generated when the buffer is completely
       empty)
    SPI2CONbits.SRXISEL = 0b11; // SPI Receive Buffer Full
       Interrupt Mode (generated when the buffer is full)
    SPI2BRG = 50;
    SPI2CONbits.ON = 1;         // Configuration is done, turn
       on SPI2 peripheral
}
```

```
void ESP32_init() {
    ESP32_IO_init();
    ESP32_SPI_init();
}

#endif /* _ESP32_H_ */


#ifndef _ADS1294R_H_
#define _ADS1294R_H_

#include <xc.h>
#include <stdint.h>

#include "util.h"
#include "ESP32.h"


/* Pin Mapping */

// Test points
#define TP6_PIN PORTDbits.RD4
#define TP7_PIN PORTDbits.RD5
#define TP8_PIN PORTDbits.RD6

// Controls
#define DRDY_PIN PORTDbits.RD7
#define CLKSEL_PIN PORTDbits.RD8
#define CS_PIN PORTDbits.RD9
#define START_PIN PORTDbits.RD10


/* Register Addresses */

// Device settings (READ-ONLY)
#define ID          0x00

// Global Settings across channels
#define CONFIG1     0x01
#define CONFIG2     0x02
#define CONFIG3     0x03
#define LOFF        0x04

// Channel-specific settings
#define CH1SET      0x05
#define CH2SET      0x06
```

```
#define CH3SET      0x07
#define CH4SET      0x08
#define RLD_SENSP   0x0D
#define RLD_SENSN   0x0E
#define LOFF_SENSP  0x0F
#define LOFF_SENSN  0x10
#define LOFF_FLIP   0x11

// Lead-off status registers (READ-ONLY)
#define LOFF_STATP  0x12
#define LOFF_STATN  0x13

// GPIO and other registers
#define GPIO        0x14
#define PACE        0x15
#define RESP        0x16
#define CONFIG4     0x17
#define WCT1        0x18
#define WCT2        0x19


/* SPI Command Definitions */

// System commands
#define WAKEUP      0x02
#define STANDBY     0x04
#define RESET       0x06
#define START       0x08
#define STOP        0x0A

// Data read commands
#define RDATAC      0x10
#define SDATAC      0x11
#define RDATA       0x12


/* Chip info */

// Channel definitions
#define NUMBER_OF_CHANNELS 4
#define BYTES_PER_CHANNEL 3
#define BYTES_TO_READ (NUMBER_OF_CHANNELS * BYTES_PER_CHANNEL)

#define CS_DELAY 0
```

```
/* Channel data struct */
typedef struct {
    uint32_t HEAD:24;
    uint32_t CH1:24;
    uint32_t CH2:24;
    uint32_t CH3:24;
    uint32_t CH4:24;
} ChannelData;

#define THREE_BYTE(B1, B2, B3) ((B1 << 16) | (B2 << 8) | B3)


/* Low-level driver */

uint8_t ADS1294R_write(uint8_t data) {
    // Low-level SPI driver
    SPI3BUF = (uint32_t)data;           // Place data we want to
        send in SPI buffer
    while(!SPI3STATbits.SPITBE);        // Wait until sent
       status bit is cleared
    return (uint8_t)SPI3BUF;            // Read data from buffer
        to clear it
}

uint8_t ADS1294R_read() {
    return ADS1294R_write(0x00);
}

void write_cmd(uint8_t cmd) {
    CS_PIN = 0;
    ADS1294R_write(cmd);
    CS_PIN = 1;
}

/* Register drivers */

uint8_t read_register(uint8_t reg) {
    static uint8_t read_register_cmd = 0x20;
    static uint8_t read_register_mask = 0x1F;

    uint8_t first_byte = read_register_cmd | (reg &
       read_register_mask);
    uint8_t second_byte = 0x00; // only ever read a single
       register

    CS_PIN = 0;
```

```
    ADS1294R_write(first_byte);
    ADS1294R_write(second_byte);
    ADS1294R_read();
    uint8_t ret = ADS1294R_read();
    CS_PIN = 1;

    return ret;
}

void write_register(uint8_t reg, uint8_t data) {
    static uint8_t write_register_cmd = 0x40;
    static uint8_t write_register_mask = 0x1F;

    uint8_t first_byte = write_register_cmd | (reg &
       write_register_mask);
    uint8_t second_byte = 0x00; // only ever write a single
       register

    CS_PIN = 0;
    ADS1294R_write(first_byte);
    ADS1294R_write(second_byte);
    ADS1294R_write(data);
    CS_PIN = 1;
}


/* ADS1298RADS1294R init */

void ADS1294R_GPIO_init() {
    // Not sure if any of these work...
    TRISDbits.TRISD4 = 0;       // TP6 as output    -  Pin 52
    TRISDbits.TRISD5 = 0;       // TP7 as output    -  Pin 53
    TRISDbits.TRISD6 = 0;       // TP8 as output    -  Pin 54

    TRISDbits.TRISD7 = 1;       // nDRDY as input   -  Pin 55
    TRISDbits.TRISD8 = 0;       // CLKSEL as output -  Pin 42
    TRISDbits.TRISD9 = 0;       // CS as output     -  Pin 43
    TRISDbits.TRISD10 = 0;      // START as output  -  Pin 44

    TP6_PIN = 0;
    TP7_PIN = 0;
    TP8_PIN = 0;

    CLKSEL_PIN = 0;
    CS_PIN = 1;
    START_PIN = 0;
```

```
}

void ADS1294R_SPI_init() {
    SPI3CONbits.ON = 0;          // Turn off SPI2 before
       configuring
    SPI3CONbits.FRMEN = 0;       // Framed SPI Support (SS pin
       used)
    SPI3CONbits.MSSEN = 0;       // Slave Select Enable (SS
       driven during transmission)
    SPI3CONbits.ENHBUF = 0;      // Enhanced Buffer Enable (
       disable enhanced buffer)
    SPI3CONbits.SIDL = 1;        // Stop in Idle Mode
    SPI3CONbits.DISSDO = 0;      // Disable SDOx (pin is
       controlled by this module)
    SPI3CONbits.MODE32 = 0;      // Do not use 32-bit mode (8-bit
        mode)
    SPI3CONbits.MODE16 = 0;      // Do not use 16-bit mode (8-bit
        mode)

    // SMP = 1; data sampled at end of output time... SMP = 0;
       data sampled at middle of output time
    SPI3CONbits.SMP = 1;

    // CKE = 1; transition from active to idle... CKE = 0;
       transition from idle to active
    SPI3CONbits.CKE = 0;

    // CKP = 1; high is idle, low is active... CKP = 0; low is
       idle, high is active
    SPI3CONbits.CKP = 0;

    SPI3CONbits.SSEN = 0;        // Slave Select Enable (SS pin
       used by module)
    SPI3CONbits.MSTEN = 1;       // Master Mode Enable
    SPI3CONbits.STXISEL = 0b01; // SPI Transmit Buffer Empty
       Interrupt Mode (generated when the buffer is completely
       empty)
    SPI3CONbits.SRXISEL = 0b11; // SPI Receive Buffer Full
       Interrupt Mode (generated when the buffer is full)

    // SCLK period > 70ns
    // 70ns ~= 14.3MHz
    // F_SCK = 14MHz

    // Library uses 4MHz
```

```
    // BRG = (F_PB / 2 * F_SCK) - 1
    // BRG = 1.86
    // BRG >= 2

    SPI3BRG = 4;
    SPI3CONbits.ON = 1;          // Configuration is done, turn
        on SPI3 peripheral
}


/* Public Functions */

void ADS1294R_init() {
    ADS1294R_GPIO_init();
    ADS1294R_SPI_init();

    // Set CLKSEL pin = 1
    CLKSEL_PIN = 1;
    delay(1);

    write_cmd(RESET);
    delay(1);

    // Send Stop Data Continuous command
    write_cmd(SDATAC);
    delay(1);

//    write_register(GPIO, 0b11110000);

    // Write config registers
    write_register(CONFIG1, 0x86);  // 500 samples/s
    delay(1);
    write_register(CONFIG2, 0x00);  // Test signals disabled
    delay(1);
    write_register(CONFIG3, 0xC0);  // Enable internal reference
        buffer, no RLD
    delay(1);

    // Send Read Data Continuous command
    START_PIN = 1;
    delay(1);
    write_cmd(START);
    delay(1);
    write_cmd(RDATAC);
    delay(1);
}
```

```
void read_data(ChannelData* ch) {
    CS_PIN = 0;

    ADS1294R_read();    // read once to clear out previous
        buffer
//    ch->HEAD = THREE_BYTE(ADS1294R_read(), ADS1294R_read(),
    ADS1294R_read());
//    ch->CH1 = THREE_BYTE(ADS1294R_read(), ADS1294R_read(),
    ADS1294R_read());
//    ch->CH2 = THREE_BYTE(ADS1294R_read(), ADS1294R_read(),
    ADS1294R_read());
//    ch->CH3 = THREE_BYTE(ADS1294R_read(), ADS1294R_read(),
    ADS1294R_read());
//    ch->CH4 = THREE_BYTE(ADS1294R_read(), ADS1294R_read(),
    ADS1294R_read());

    uint8_t h1 = ADS1294R_read();
    uint8_t h2 = ADS1294R_read();
    uint8_t h3 = ADS1294R_read();

    ch->HEAD = (h1 << 16) | (h2 << 8) | h3;

    for (uint8_t i = 0; i < BYTES_TO_READ; i++) {
        ADS1294R_read();
    }

    CS_PIN = 1;
}

uint8_t data_ready() {
    return DRDY_PIN == 0;
}

#endif /* _ADS1294R_H_ */
```