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)

October 16, 2023

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 create 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 addition of lighting control. Hardware for the project had been developed, but this hardware had only been programmed with minimal testing software. This hardware consists of a PIC32, as the primary microcontroller; an ESP32, acting as a wireless transmitter; and an ADS1294R, which is a specialized analog to digital converter for biosignals. This thesis focuses on creating functional firmware for the various microcontrollers present on the previously developed hardware, as well as implementing lighting control as an additional output for the system.

The project began with the implementation of the lighting controller. This block allows the system to connect to standard performance lighting fixtures and send arbitrary control messages based on acquired biosignals. The lighting output uses an off-the-shelf controller, allowing high reliability and effective compliance. Additionally, a prototyping fixture was developed, allowing fast and cost effective testing without the need for an expensive fixture. With these additions, the system is able to map collected signals into interesting lighting effects.

Further project developments involved the previously designed hardware. The hardware had been programmed to ensure it was wired correctly, with the PIC32 being functional but unconfigured, and the ESP32 being programmed intermittently. The first step was to ensure all devices could be programmed consistently, starting with the ESP32. The intermittent programming issue was solved by implementing over-the-air updates, allowing the device to be programmed over WiFi without needing reset the device into a specific boot mode. Next, the PIC32 was configured correctly to allow hardware debugging.

Once the devices could be consistently programmed, device-to-device communication was established. This started with the PIC32 communicating with the ESP32 over SPI. There were a number of challenges with this, but eventually consistent communication was established. Then, a similar process was undertaken for communication between the PIC32 and the ADS1294R. This had significant problems. Firstly, the communication lines of the devices had no test points, so there was not way to determine where a number of faults were occurring. Secondly, the ADS1294R was only responding to single-byte commands, and because of the lack of test points, it was not immediately apparent why. Lastly, there was a difference between

the schematic designs and the manufactured design that meant the device would not respond exactly as expected, even when functional. These issues were overcome and communication was successfully established.

The final stage of the project was system integration. An off-body PC was setup, and wireless communication was established to the ESP32 using TCP/IP. Then, communication with the off-body PC and the PIC32, through the ESP32 acting as a wireless bridge, was confirmed. An additional debugging tool was implemented here, to allow arbitrary text transmission between the PIC32 and the off-body PC using a modified version of printf. The final stage of system integration was collecting sensor readings from the ADS1294R and transmitting them wirelessly to the off-body PC for processing. However, at this point in the project, the previously designed hardware started having power issues. This is most likely due to the high impedance of the board power lines, and with the addition of all of the devices running together, the current draw causes too much of a voltage drop across the high impedance lines.

Overall, the project has made vast headway's in the development of a performance augmentation platform. While the device was not made fully functional, communication issues between all elements of the on-body device were solved, and additional recommendations can be implemented into a second version of the board.

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 [58], they are used in the body to detect various biological events such as muscle contractions and heartbeats [20]. These signals can be detected using various types of sensors, including electric, mechanical, acoustic, and infrared sensors [31].

Music has been a form of human expression for over 40,000 years [33]. 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' [14] [59]. 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 [40] and require a large number of electrodes [54]. 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][50], which is a common and painless measurement that is used to monitor the

heart [41]. Electromyography (EMG) [62][69], which measures electrical activity due to the response of muscles [66]. Galvanic skin response (GSR) [35], which can show the intensity of emotional changes due to the change in conductance of the skin [21]. And respiratory rate [8], which measures the number of breaths per minute [67]. These signals have a degree of predictability [61] 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 [44], and 'Stethophone' by Nerness and Fuloria [46]. 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' [22], and 'Emovere' by Jaimovich [27]. 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[68]. 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 [15][65]. Additionally, software developed in MATLAB has been developed that processes the incoming signals and generates music in real-time [11][47]. 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.
  - The hardware has been successfully powered-on.
  - Programmer communication with the microcontrollers has been established.
  - One of the microcontrollers has been connected 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 once the project began was the lighting controller. This part of the project was built from scratch, as there was no prior work that had been done in this area. This made it perfect for the first half of the year, as other sections of the project required hardware that was currently being developed by the students finishing mid-year.

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 whatever lights are connect to it.

Different protocols for controlling various lighting fixtures exist. To determine which should be used for this project, a decision matrix, shown in Table 2.1 and Table 2.2 was used. For this project, DMX512 (DMX) was determined to be the most appropriate.

TABLE 2.1: Lighting protocol decision matrix

	DMX512	RDM	Modbus	0-10V
Simplicity (0.13)	0.80	0.40	0.80	0.90
Expense (0.14)	1.00	1.00	0.70	1.00
Scalability (0.10)	0.90	1.00	0.90	0.20
Adoption (0.16)	0.90	0.80	0.50	0.30
Usability (0.16)	1.00	0.70	0.40	0.20
Documentation (0.13)	0.90	0.90	0.90	0.30
Licensing (0.18)	1.00	1.00	1.00	1.00
Total (1.00)	0.94	0.83	0.73	0.58

To control DMX fixtures, there are two options, using an off-the-shelf controller or creating a custom controller. As shown in this decision matrix, the more viable option was to purchase an off-the-shelf controller.

TABLE 2.2: Lighting protocol decision matrix (continued)

	EnOcean	TCP/IP	DALI	BACnet
Simplicity (0.13)	0.40	0.20	0.30	0.30
Expense (0.14)	0.40	0.30	0.20	0.10
Scalability (0.10)	0.60	0.80	0.70	0.70
Adoption (0.16)	0.20	0.40	0.30	0.30
Usability (0.16)	0.40	0.30	0.30	0.30
Documentation (0.13)	0.80	0.80	0.10	0.10
Licensing (0.18)	0.00	0.00	0.00	0.00
Total (1.00)	0.37	0.36	0.25	0.23

2.1 Prototyping Fixture

To aid in the development of the lighting controller, a prototyping fixture was developed. This fixture just requires an Arduino and a NeoPixel LED strip to function. This has the major benefit over a real fixture of being extremely cost effective, as well as being small, and powered over USB.

The prototyping fixture is made up of 8 NeoPixel LEDs with 4 DMX channels each. The DMX channels are intensity, red channel, green channel, blue channel. This makes a total of 32 channels (4×8) for the 8 LEDs.

2.1.1 DMX

The fixture connects to the off-body PC over USB and communicates via serial. While the device does not strictly require DMX frames in order to function, DMX was still implemented in software in order to better understand the protocol as well as unify the prototyping fixture with the real controller.

2.1.2 NeoPixels

The prototyping fixture contains an array of NeoPixel LEDs. These LEDs are controlled using the Adafruit NeoPixel library [1]. The LEDs can then be colored independently using red, green, and blue values. Since each color is represented using a byte, it can be mapped directly to a DMX channel, since those are also a byte long.

2.1.3 Integration

To implement DMX intensity control, the LED brightness needs to be mapped using the the code shown in Listing 2.1.

LISTING 2.1: Prototyping fixture DMX color intensity mapping

```
(intensity * color) >> 8;
```

In this case bit shifting by 8 is equivalent to dividing by 255. This means that at maximum intensity the result of this calculation is the color value. While at the minimum intensity the calculation becomes 0.

The value of each LED is encoded across 4 bytes. So the main DMX processing loop increments by 4 each time, and extracts the discrete bytes, setting the base address of the LED to the derived color. Once this is done for all the LEDs, they are updated using the NeoPixel show() function.

With this, the prototyping fixture can respond to DMX channels in the same way you would expect a conventional DMX fixture to.

2.2 OpenDMX Controller

For use in professional lighting environments, it is important to use compliant hardware, since many of the devices in these environments come with a substantial cost.

Therefore, for this project a third-party device was used in order to mitigate the risk associated towards these devices. The third-party device that was chosen was the ENTTEC OpenDMX USB device [19], as it was within the system budget, has an open source driver, and maintained compliance.

2.2.1 Interface

The driver provided by the manufacturer was written in C#. Since existing code for this project had been written in MATLAB. An interface between the device and the existing code had to be created. The simplest way to do this is to create a TCP/IP port that connects to a C# server. That way, any high level language can connect and control the DMX output device, including MATLAB.

The code for doing this is shown in Listing 2.2

LISTING 2.2: C sharp DMX server receiver code

```
static void Main(string[] args)
{
        TcpListener server = null;
        OpenDMX open_dmx = new OpenDMX();

        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]);
                open_dmx.setDmxValue(channel, value);
             }
        }
}
```

3 On-Body Device

The on-body device was developed by William Tran in 2022 [65] and consists of two microcontrollers, a PIC32 and a ESP32, as well as 24-bit ADC for taking biosignal measurements.

At the middle of the year when the project was handed over, the ESP32 had been programmed to connect to WiFi, but was only programming intermittently, and the PIC32 had bare-bones programming on it to enable the ESP32 via its enable pin.

The only other contributions were from Craig Dawson who supplied information and 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 ESP32

The ESP32 on the board is the ESP32-WROOM-32D. This device is used as a wireless bridge between the on-body device and the off-body PC. It connects to the PIC32 through a Serial Peripheral Interface (SPI) bus. It is programmed using the Arduino software environment.

3.1.1 Power

As stated, the ESP32 was only programming intermittently. To determine what the cause of this unwanted behavior was, the board 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.

From this testing, it was discovered that the ESP32 programs successfully when it is being adequately powered and the boot select switch is pressed as the device is being powered on.

The additional power requirement is not due to any external limitations with the lab power supply, as the current draw that the supply is measuring is significantly less than the set current limit. Additionally, there is a reduction in current draw from the first supply once the additional power supply is added. This means that the load is being shared between the two supplies, as opposed to the first supply being at its maximum output and the second supply providing necessary additional power.

What his means more broadly is that the problem with programming the device comes from the power distribution of the on-body device. This can be further verified by measuring the voltage at points on the on-body device. All of the ICs on the device operate at a 3.3V power level. When measuring the voltage at the input pins of the ICs and at headers around the board, the voltage appears to be closer to 2.6V. As we add additional voltage connections, we can observe the voltage rise. Although the voltage does not reach 3.3V, the increase in voltage appears to be enough for the ICs to remain powered on. This appears to be because the power traces on the board are not wide enough. Because of this, the traces have significant resistance which causes a voltage drop to occur across them once the higher IC currents begin to flow.

For the ESP32, the lower voltage causes the brownout detection feature of the device to activate, causing it to reset. The effect that has on the device is that it will reset itself out of programming mode, causing the programming to fail. This is because the ESP32 must be put into programming mode by holding down the boot select switch while the device is initially powered on. So, even in instances when the device has been successfully put into this mode, the device reset caused by the brownout detector reverts it out of this mode.

Once the power supply was supplemented with additional power supplies, this issue became less problematic. However, additional issues arose as the ESP32 has to be placed into programming mode as it is powered. With the addition of these power supplies, there is coordination required in order to get it into this mode. Since the power is also required to keep the ESP32 from resetting, all of the supplies need to be disconnected and reconnected at the same time, while the boot select switch is pressed. Additionally, with this setup there is not way to validate the correct entry into programming mode, meaning there is not way to know if it has actually been placed into the correct mode until the programming fails. For prototyping, this becomes extremely cumbersome because this process must be repeated every time there is a change. As well as whenever the programming fails due to an unexpected reset.

3.1.2 Over-The-Air Programming

Over-The-Air (OTA) programming is a technique used to program a device wirelessly [56]. When compared to the alternative, OTA programming has a number of benefits.

- The device can be programmed regardless of the boot mode.
- The device can be programmed without the use of a UART converter.
- The device can be programmed without a physical connection to a PC.

The downsides to this programming mode is that it adds additional compilation time, as well as minor overhead during run-time. However, the compilation time is already in the range of 40 to 50 seconds, and the addition is less than 5 seconds, and the run-time overhead can be mitigated by running OTA code in its own thread. Considering these benefits and drawbacks, it was determined that this programming mode should be implemented.

The first step in implementing this feature was to connect the device to the network. The ESP32 is programmed using the Arduino IDE, which contains a WiFi library for the ESP32. The code for simple WiFi device bring-up is shown in Listing 3.1.

LISTING 3.1: Arduino code for connected ESP32 to WiFi

```
#include <WiFi.h>

void setup() {
```

```
    Serial.begin(115200);
    Serial.println('Booting');
    WiFi.mode(WIFI_STA);
    WiFi.begin(SSID, PASS);

    while (WiFi.waitForConnectResult() != WL_CONNECTED) {
      Serial.println(''Connection Failed! Rebooting...'');
      delay(5000);
      ESP.restart();
    }

  }
```

Once the device has been connected to the network, OTA programming can be implemented using the ArduinoOTA library. This consists of including the ArduinoOTA.h header file, configuring OTA updates using the code shown in Listing 3.2, and calling ArduinoOTA.handle() in the primary loop of the program.

LISTING 3.2: Arduino code for configuring OTA updates

```
  ArduinoOTA
  .onStart([]() {
    String type;

    if (ArduinoOTA.getCommand() == U_FLASH)
      type = 'sketch';
␣␣␣␣else␣//␣U_SPIFFS
␣␣␣␣␣␣␣type␣=␣'filesystem';

␣␣␣␣␣␣␣Serial.println('Start␣updating␣' + type);
    })

    .onEnd([]() {
      Serial.println('\nEnd');
␣␣␣␣})

␣␣␣␣.onProgress([](unsigned␣int␣progress,␣unsigned␣int␣total)␣{
␣␣␣␣␣␣␣Serial.printf('Progress: \%u\%\%\\r',␣(progress␣/␣(total␣/
   ␣100)));
␣␣␣␣})

␣␣␣␣.onError([](ota_error_t␣error)␣{
␣␣␣␣␣␣␣Serial.printf('Error[\%u]:␣', error);
      if (error == OTA_AUTH_ERROR) Serial.println('Auth Failed')
          ;
```

```
␣␣␣␣␣␣else␣if␣(error␣==␣OTA_BEGIN_ERROR)␣Serial.println('Begin
    Failed');
␣␣␣␣␣␣else␣if␣(error␣==␣OTA_CONNECT_ERROR)␣Serial.println('
    Connect␣Failed');
       else if (error == OTA_RECEIVE_ERROR) Serial.println('
          Receive␣Failed');
       else if (error == OTA_END_ERROR) Serial.println('End
          Failed');
␣␣␣␣});
```

After implementing these libraries, the ESP32 can be programmed using OTA by selecting the relevant device in the Arduino IDE. One caveat of this is that the OTA updates will only be pushed as long as the OTA handler is called. So, for instances where the ESP32 is in an infinite loop, or when there is a substantially long delay in the primary loop, the ESP32 will need to be programmed using the original hardware method.

3.2 PIC32

The PIC32 on the board is the PIC32MX775F512H. It connects to sensors, the 24-bit ADC, and the ESP32. It is the main processor on the on-body device and it is programmed in C using MPLAB X v5.00 with an ICD 3. The ICD 3 must be supplying 3.3V to on-body device in order for the PIC32 to be programmed successfully.

3.2.1 Programming Configuration

When the board was being manufactured, there was a global supply chain issue, as mentioned in Tran's thesis [65]. Because of this, the schematic design of the board was different to what was assembled on the board. This caused issues when programming the PIC32, because the specific model of PIC had changed. So, when the programmer connected to the device, it would read a different device ID from what was expected, and the programming would fail. Updating the project to use the PIC32MX775F512H solved this programming issue.

The other hurdle in programming the PIC32 was in the clock configuration. With incorrect clock configuration, the device still programs, but it is not able to be put into debug mode. Additionally, all delays that are added are based on the clock speed. So it must be set correctly for the delays to be correct.

The clock configuration uses PLL [39]. It has been configured with a input divider of 10, a multiplier of 16, and an output divider of 8. With a 16 MHz crystal, the system clock frequency becomes $(16MHz/10*16)/8 = 3.2MHz$.

The system clock frequency was verified by measuring the delay between pulses using an oscilloscope. The code for delays in the system is shown in Listing 3.3, and code for generating output pulses is shown in Listing 3.4

LISTING 3.3: PIC32 code for adding delays in code execution (in microseconds)

```
void delay_us(unsigned int us) {
  us *= DELAY_CONST;
  _CP0_SET_COUNT(0);
  while (us > _CP0_GET_COUNT());
}

void delay(int ms) {
  delay_us(ms * 1000);
}
```

LISTING 3.4: PIC32 code for measuring system clock speed using delayed pulses

```
#define TP7 PORTDbits.RD5 // Pin definition for test point 7

void run() {
  TP7 = 1;
  delay(100);
  TP7 = 0;
  delay(100);
}
```

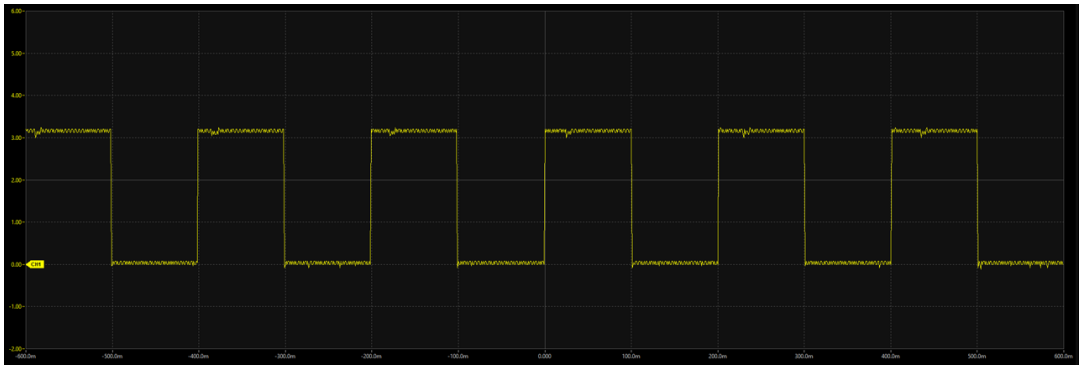
The measured results for this are shown on the oscilloscope image seen in Figure 3.1. This figure shows a 100 ms delay between pulses. This is exactly what the software delay has been set to, which verifies that the system clock frequency has been set correctly, because the delay relies on the system frequency to calculate the desired amount of clock ticks to wait. The equation for calculating the number of clock ticks is $F_{sys}/1000000/2$.

With the system clock frequency set correctly, debugging works on the device and the CPU can be halted and stepped through instructions as the program is running.

3.2.2 ESP32 Control

The ESP32 is controlled by the PIC32 in two ways.

FIGURE 3.1: Oscilloscope measurement of 100ms delay between pulses



- The ESP32 enable pin is connected to pin 14 (RB2) of the PIC32.
- The ESP32 HSPI bus is connected to SPI2 of the PIC32

The enable pin of the ESP32 just needs to be asserted high, which can be done by the PIC32 using the code shown in Listing 3.5

LISTING 3.5: PIC32 code for enabling the ESP32

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

Without this assertion, the ESP32 is unresponsive to programming and does not perform code execution.

The other form of control is through the respective SPI lines of the two microcontrollers. The SPI driver for the PIC32 is simple, as that device acts as the 'master', which is the typical mode for a microcontroller to operate in. The PIC32 is configured as a 32-bit master operating in SPI mode 3 [5]. The baudrate 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, they can be easily raised by decreasing the value of BRG, once the system has been well tested. Finally, the low-level SPI driver is implemented in Listing 3.6

LISTING 3.6: PIC32 low-level SPI driver

```
uint32_t ESP32_SPI_write(uint32_t data) {
  SPI2BUF = data;
  while(!SPI2STATbits.SPITBE);
  uint32_t read = SPI2BUF;
```

```
    delay_us(5000);
    return read;
}
```

The function of the code is relatively straightforward. The memory location of the SPI2BUF variable is mapped to the SPI 2 peripheral. When written to, the data at this address is written into a transmit buffer that queues the data for SPI transmission. When read from, data that has been received by the SPI peripheral is taken from the receive buffer. Between these two operations the processor waits for the SPITBE status flag to be set. This flag corresponds to the transmission buffer being empty, and is set once transmission has been completed and data has been received.

Additionally, a 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 functional. and thus the driver was implemented using significantly slower delay.

The ESP32 SPI configuration was less straightforward. As microcontrollers are typically the devices that coordinate communication between various 'dumb' sensors, they almost always act as the SPI master. However, since the on-body device uses multiple microcontrollers communicating via SPI, one of these devices needs to act as a slave. Since the PIC32 is what communicates to the sensors, and the ESP32 only acts as a wireless transmitter, the PIC32 was configured as the SPI master, leaving the ESP32 as an SPI slave. There no official support for this in the native development environment, so a third party library was used.

The library that was used is the ESP32DMASPI library [25]. This library is based on the official driver from the manufacture [60].

SPI was implemented using task based DMA receiving. In this setup, tasks run in separate threads, which frees up main thread to only process

OTA updates. One task runs continuously, waiting for SPI transmission Listing 3.7.

LISTING 3.7: ESP32 code for receiving SPI data

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

Once data has been received, a different task handles processing the incoming data Listing 3.8.

LISTING 3.8: ESP32 code for processing SPI data

```
void task_process_buffer(void* pvParameters) {
  while (1) {
    ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
    print_array(buffer, slave.available());
    slave.pop();
    xTaskNotifyGive(task_handle_wait_spi);
  }
}
```

This should have been the end of the SPI implementation, however the SPI communication did not work with just this code. To get the communication working some additional functions were required on the PIC. For example, the only way to send data was to write a single byte at a time, despite the fact that the SPI module is in 32-bit (4 byte) mode. What is interesting about this is that the single byte must be located at the front of the 4 byte word. This implies that the ESP32 was not able to receive 32-bits, and instead was just receiving the first 8-bit word. However, if the SPI peripheral of the PIC32 was put into 8-bit mode (and all respective code changed to fit that mode), the ESP32 would receive nothing. So, the ESP32 required the PIC32 to send 32 SPI clock pulses but would only receive the first 8 data bits. This same behavior was present when using both a task based approach or a polling approach and when DMA was or was not in use. It is a substantial issue because it adds a 75% overhead to the communication system. This, and the required delay in the low-level driver, are the primary candidates for future optimizations. The code for embedding data into a 4 byte word is shown in Listing 3.9.

LISTING 3.9: PIC32 additional SPI functions

```
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);
}
```

Despite significant performance issues, the drivers function together. Allowing arbitrary bytes to be shared between the two devices.

3.3 ADS1294R

The ADS1294R is a 4 channel, 24-bit, delta-sigma analog-to-digital converter with additional features to support electrocardiogram and electroencephalogram measurements. This device is the primary sensor front end. It communicates with the PIC32 via SPI as well as several hardware control lines.

3.3.1 Design Differences

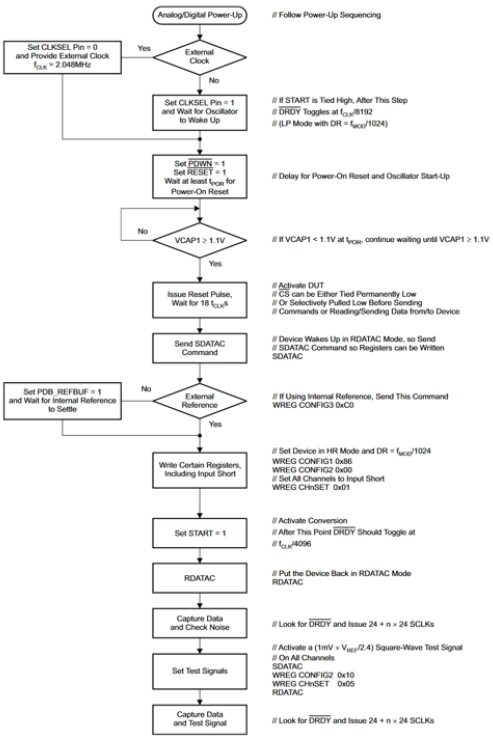
Like the PIC32, the schematic for the ADS1294R were slightly different to what was assembled due to chip shortages. The schematic showed the devices as an ADS1298R, which is the 8 channel version of the device. This is significant because the device requires a specific number of SPI clock pulses to be sent that correlates to the number of channels on the device. So sending the wrong number of clock pulses will generate undesired results. Additionally, the device ID is different which can cause some confusion when initially trying to configure the device.

3.3.2 Startup Procedure

The startup procedure of the device is shown in Figure 3.2. When implementing this startup procedure, a similar SPI driver to what was seen in Listing 3.6 was used. The key difference is that a higher delay was used for testing to ensure communication was well below data sheet limitations [2]. When trying to follow this procedure, there were a number of key differences between

what was expected and what was measured. It appeared that no matter what command was sent, the response was always the binary number 01100000. The data sheet specified responses for different registers, but regardless of the register the response was always the same. The device also did not appear to respond to any two byte commands. However, the device did respond to single byte commands.

FIGURE 3.2: ADS1294R startup sequence



The single byte commands were verified by monitoring the state of the data ready pin. Since this pin was connected to the PIC and the PIC has general purpose test points on a number of pins; the read value of the data ready pin can be written to one of the test points. This effectively 'passes through' the state of the pin. The caveat being that any delays to the PIC will cause a delay between the data ready value and the test point. As well as that it strips the signal of all information aside from if it has crossed the PIC's logic threshold. However, the limited information it provides is still useful in this context. Using an oscilloscope connected to the 'pass through' test point, it is shown that the data ready pin is continually asserted. This changes after calling single byte commands that change the state of the ADS1294R. For example, if the reset command is continually send, the data ready pin will never be asserted, once the command stops being sent, the data ready pin is reasserted. Similarly, if the standby or stop commands are sent, the pin

is not asserted until the respective wakeup or start commands are sent. The exact same behavior is observed when the physical reset pin is held down, or when the start pin is asserted by the PIC. What this says is that the device is powered and the transmit side of the SPI bus is connected correctly. Since the SPI binary response is always 01100000, it can be assumed that the receive side of the SPI bus is also connected correctly. This is because the response does not change and is both non-zero and also not continually driven high. If either of those were true, the SPI bus could be picking up noise, or the receive pin may not be getting asserted. But because of this response it must be being controlled by the ADS1294R SPI module, since there is no other part of that device that is connected to the SPI clock and the response is the same even at higher and lower SPI clock speeds.

As the SPI hardware is assumed to be connected correctly, and the low-level SPI driver is confirmed to work with the ESP32, the fault is assumed to be in the SPI configuration and startup procedure on the PIC. Changing the clock polarity or clock phase causes the single byte commands to fail. On top of that, changing the SPI interface off of 8-bit mode to 32-bit or 16-bit mode also caused the commands to fail. The only change that could be made was the data sampling time, as this effected nothing to do with the transmission. This change caused the received bits to change position, but they were still not correct and different commands still caused the device to respond with the same response each time. Therefore, the place in the program that was most widely explored was the code for the startup procedure.

The most obvious difference is that the startup procedure suggests using the hardware reset, but that pin is tied to a physical switch instead of directly connected to the PIC. Because of this, there is no way to automatically hardware reset the device in software, so the reset has previously been achieved by sending the reset command. To achieve a true hardware reset, a long delay was added to the startup sequence so that the physical switch could be pressed with the correct timing. To aid with the timing, one of the test points was also driven high at the start of this timing window and low once the window had passed. Then, using an oscilloscope to verify the timing window, the ADS1294R could be hardware reset at the correct point in the boot-up process. This unfortunately did not solve the issue and the device still maintained the same issue as before.

Next, the register read and register write functions were investigated. Since we had verified that the low-level driver was functional, and that same driver handled both the reading and writing, it was unlikely that this was

the issue, as this driver is so simple there are not many places errors could propagate.

The code for writing to the register is shown in Listing 3.10.

LISTING 3.10: PIC32 code for writing ADS1294R register

```
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

  ADS1294R_write(first_byte);
  ADS1294R_write(second_byte);
  ADS1294R_write(data);
}
```

As the write commands were verified to be working, the changes to this function were around the register command byte, register mask byte, and the addition of delays between writing the bytes. However, these changes did not have any effect on the communication issues with the device. We then turned our attention to the read register function shown in Listing 3.11

LISTING 3.11: PIC32 code for reading ADS1294R register

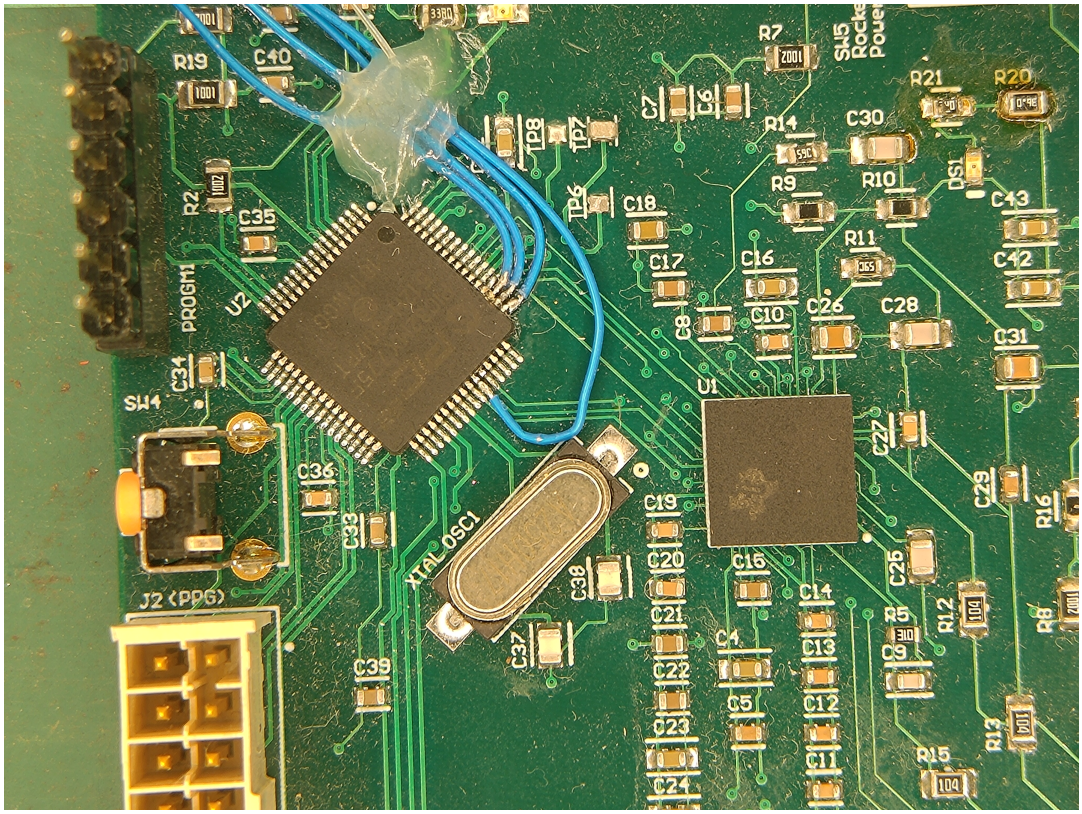
```
uint8_t read_register(uint8_t reg) {
  static uint8_t read_register_cmd = 0x20;
  static uint8_t read_register_mask = 0x1F;

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

  ADS1294R_write(first_byte);
  ADS1294R_write(second_byte);
  return ADS1294R_read();
}
```

Similar changes and additions were added to this function, with the same result of the communication not working correctly.

At this point all of the command and register definition were double checked and the register read and write functions were stepped through to verify that the bytes bit by bit, and all confirmed to be correct.

Since all logical software changes had been made and the system was still not functional, it was decided that the SPI pins should be broken out so the communication could be analyzed using an oscilloscope. Craig Dawson from engineering services helped in this aspect of the project, since the pitch of the pins that needed to be soldered required specialized tools and expertise. The device modifications are shown in Figure 3.3.

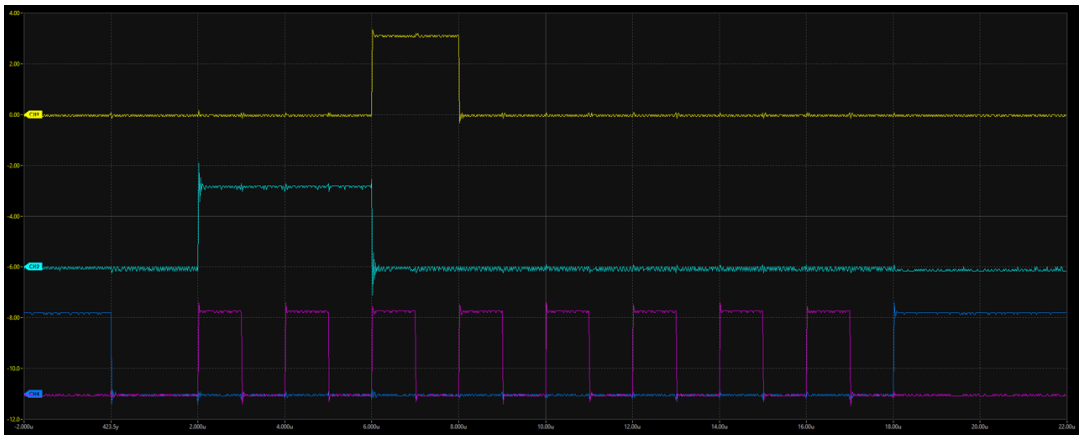
FIGURE 3.3: PIC32 board attached debugging wires



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 3.4. Since the chip select pin was being controlled by the PIC SPI module, it was being automatically driven low during transmission and then high again after transmission completed. This would have been the desired behavior, and it indeed was for single byte commands, but for multi-byte commands the chip select line needed to stay asserted until all of the bytes had been transferred and received.

The solution to this is to disable chip select control on the SPI module and manually set and reset the pin in software. To accomplish this, the code was changed as shown in Listing 3.12 and Listing 3.13

FIGURE 3.4: SPI communication between PIC32 and ADS1294R



LISTING 3.12: PIC32 code for writing single-byte commands to the ADS1294R

```
void write_cmd(uint8_t cmd) {
  CS_PIN = 0;    // Chip select pin is active low
  ADS1294R_write(cmd);
  CS_PIN = 1;    // Chip select pin in inactive high
}
```

LISTING 3.13: PIC32 code for writing single-byte commands to the ADS1294R

```
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;
}
```

With these changes, communication with the device is established and the device ID can be read successfully.

4 Integration

4.1 Off-Body PC

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 previous usage on this project.

4.1.1 Standalone Setup

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. To achieve this, the number of samples added to the front of the buffer need to be removed from the rear of the buffer each time a new sample is received. An 'offline' version of this program can be made without the need for full system integration using previously saved sampled data. Since this data is the same as what we will be eventually expecting to see, The processing code that we write for this test data will also function similarly on the real data.

For testing, an electrocardiogram (ECG) data-set from a previous iteration of the project was used. The code for bringing that into the workspace is shown in Listing 4.1. The load command will load a number of different ECG data-sets with that all have different beats per minute (BPM), and the desired ECG signal for testing can be selected by changing the the assignment of the ECG variable. For example, to load a 60 BPM ECG signal, the assignment

could be changed to ECG = ecgdata360Hz_hrmean60. The specific data-sets that are available can be seen in the workspace tab of MATLAB once the load command has been executed. Additionally, a loop rate is calculated to simulate the sensor sampling rate. Since the testing data-set sample rate is known, it can be easily calculated from that.

LISTING 4.1: MATLAB electrocardiogram packet test setup

```
load('./Previous/MatFiles/ex_ecgdata360Hz.mat');
ECG = ecgdata360Hz_hrmean220;
RATE = 1 / 360;
```

Once the data-set has been imported, the program can loop through and generate example packets for testing. The code to do this is shown in Listing 4.2.

LISTING 4.2: MATLAB electrocardiogram packet translation

```
PACKET_LENGTH = 1;
DATA_LENGTH = 1024;

packet_index = 1;
data = zeros(DATA_LENGTH, 1);

while true
    packet_index = packet_index + PACKET_LENGTH;

    %%% bounds check
    if packet_index + PACKET_LENGTH > length(ECG)
        packet_index = 1;
    end

    packet = ECG(packet_index:packet_index + PACKET_LENGTH);

    data(1:end-PACKET_LENGTH) = data(1+PACKET_LENGTH:end);
    data(end-PACKET_LENGTH:end) = packet(1:end);

    plot(data)
    axis([0 DATA_LENGTH -2 2]);

    pause(RATE);
end
```

The first thing that is done is the constants and variables definitions. In this context, packet refers to the small subset of the ECG signal that is going to be added to the larger fixed length array. That fixed length array is labelled 'data' in this example.

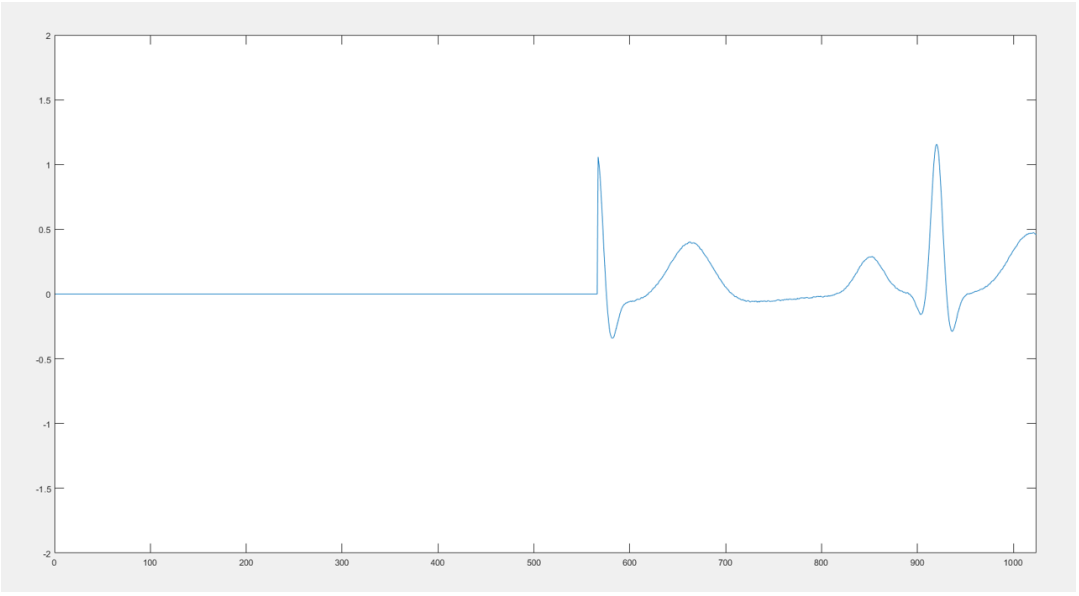
The packet index is the data position from the ECG data-set where data will be sampled from. Packet length determines how many samples at a time are shifted in and out of the data array, this is to simulate the device transmitting multiple samples at a time.

When the packet index reaches the end of the ECG data-set, the index resets back to the beginning. This is so the test program can run continuously, regardless of how big the test data-set actually is. When the index resets, there is a break in continuity of the signal. However, this is a known issue and can be easily detected and ignored.

The fixed data array can then be updated by first shifting the current data to the left, disposing of packet length worth of data at the beginning. Then, the same amount of data, from the packet, can be appended to the end of the array. Plots of this data are shown in Figure 4.1 andFigure 4.2.

With this setup, the off-body PC is ready to process an incoming signal as a fixed sized array of data. An example of potential signal analysis is shown in Listing 4.3.

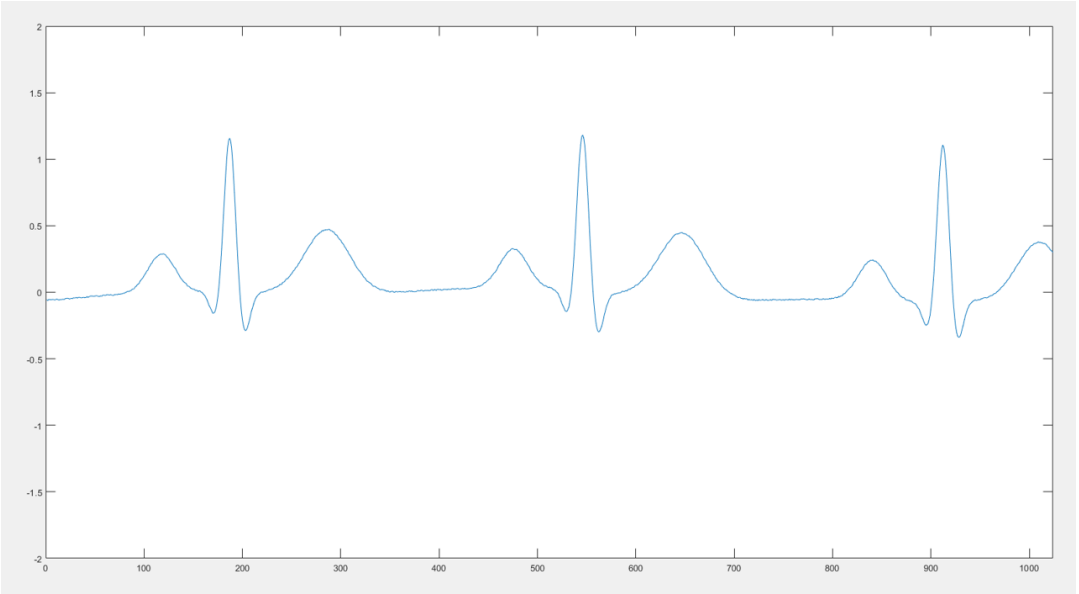
FIGURE 4.1: Fixed length data buffer visualization half filled



LISTING 4.3: MATLAB signal analaysis example

```
%%% server side signal analaysis
[peaks, locations, widths, prominances] = findpeaks(data);
threshold = max(prominances) * .60;
ecg_peak_locations = locations(prominances > threshold);

peak_gap = zeros;
for i = 2:length(ecg_peak_locations)
```

FIGURE 4.2: Fixed length data buffer visualization fully filled




```
        peak_gap(i-1) = ecg_peak_locations(i) -
            ecg_peak_locations(i-1);
    end

    BPM = round(360 / mean(peak_gap) * 60, 0);
```

This code calculates the BPM of the signal by measuring the gaps between the peaks of the ECG QRS signals. It can be verified by comparing the measured BPM value with the given value from the data-set. When the system is integrated, as long as the data length matches and the sensor data is accurate, this same code will produce similar results.

4.2 System Integration

4.2.1 Wireless Transmission

With all the block of the system functional, the first step of system integration is to establish communication between the off-body PC and on-body device.

On the on-body device, the ESP32 acts as the wireless bridge. As this device requires a connection to WiFi for OTA updates, the wireless protocol that was chosen to be implemented was TCP/IP. This is because the device already has a TCP/IP requirement and the only other choice for wireless communication was Bluetooth, which does not have the range or speed that TCP/IP has. The code for connecting the ESP32 to the off-body PC is shown in Listing 4.4 and the corresponding MATLAB code is shown in Listing 4.5.

LISTING 4.4: ESP32 code for connecting to TCP/IP client

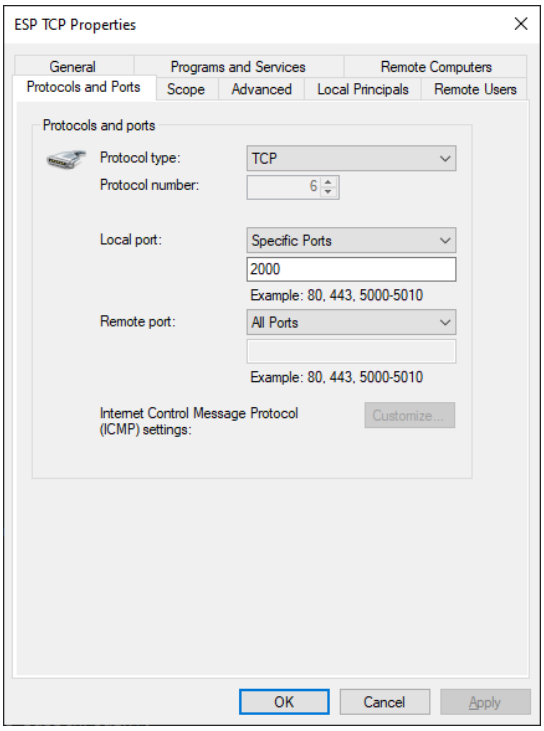
```
#include <WiFi.h>
#include <WiFiUdp.h>

WiFiClient client;

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

  while (WiFi.waitForConnectResult() != WL_CONNECTED) {
    Serial.println(``Connection Failed! Rebooting...'');
    delay(5000);
    ESP.restart();
  }

  if (client.connect(SERVER_IP, SERVER_PORT)) { Serial.println('
     Connection␣success!'); }
  else { Serial.println('Connection␣failed!'); }
}

loop() {
  if (client.connected()) {
    client.write('T');
    client.write('E');
    client.write('S');
    client.write('T');
  }

  delay(500);
}
```

LISTING 4.5: MATLAB TCPIP receive code

```
clear server

server = tcpserver("0.0.0.0", 2000);
configureCallback(server, "byte", 4, @server_callback);

function server_callback(src, ~)
    bytes = read(src, src.BytesAvailableFcnCount, "uint8");
    text = char(bytes);
    disp(text)
end
```

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 4.3.

FIGURE 4.3: TCP/IP firewall port settings



With data being sent between the ESP32 and the off-body PC, the next device to connect is the PIC32 to the off-body PC. Since the SPI communication between the PIC and the ESP has already been established, the process buffer task can be updated to pass-through the received data to the off-body PC, this is shown in Listing 4.6. The other change to this SPI code is to add the client connection setup code from Listing 4.4.

LISTING 4.6: ESP32 code for transmitting received SPI data

```
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);
  }
}
```

Then, the PIC can communicate with the off-body PC as shown in Listing 4.7.

LISTING 4.7: PIC32 code for sending data to off-body PC through the ESP32

```
void init() {
  ESP32_init();
}

void run() {
  ESP32_SPI_write_byte('T');
  ESP32_SPI_write_byte('E');
  ESP32_SPI_write_byte('S');
  ESP32_SPI_write_byte('T');
  ESP32_SPI_write_byte('\n');

  delay(500);
}
```

4.2.2 Base64

Sending individual bytes between devices is now possible. Currently, this only works as long as the value fits into a predetermined number of bytes. Additionally, once the system is running with non-known values, there is no easy way to keep the data synchronized. For instance, if a byte of data was lost, the received data would be in the incorrect place and there would be no way of knowing. This could be solved by sending a known byte sequence at the beginning of transmission. However, there would be no differentiation between that byte sequence and any arbitrary data that may be being sent. Another approach is to limit the character set so that specific byte values can correspond to 'special' characters. For example, the decimal value of 10, which corresponds to the newline character, could be used to mark the end of transmission. That way, even if the transmitter and receiver become out of sync, the system has a way of recovering. There are a number of character sets that have already been developed. For this project, base64 was implemented due to its lower overhead when compared to ASCII, its human readability to aid debugging, and its good documentation.

The code to transmit base64 encoded data from the PIC32 to the off-body PC is shown in Listing 4.8.

LISTING 4.8: PIC32 code for transmitting base64 encoded data

```
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');
}
```

This code takes the inputted data and limits it to only use characters from the encoding table. After the data has been encoded, it is sent to the ESP32 for writing to the off-body PC. That device then receives the data using the code shown in Listing 4.9.

LISTING 4.9: Off-body PC code for receiving and decoding base64 data

```
base64 = readline(src);
decoded = transpose(matlab.net.base64decode(char(base64)));
```

With this implemented, data that is sent to the off-body PC is consistently received correctly and the devices do not get out of sync. This encoding function allows for arbitrary data to be sent between the PIC32 and the off-body PC. One major benefit of this is that it allows custom wrapping of a printf function, along with the previously implemented packet writing function, to give the user a custom debugging function that allows any number or string to be formatted and sent to the off-body PC. The code to achieve this is shown in Listing 4.10.

LISTING 4.10: PIC32 printf debugging over WiFi

```
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));
}
```

The off-body PC can then decoded the received characters and print them to the console using the code in Listing 4.11.

LISTING 4.11: Off-body PC code for receiving custom printf debugging

```
y = char(decoded);
disp(y)
```

Then, the program can be debugged using the code shown in Listing 4.12. This allows the device to continue running while providing feedback, rather than needing to be put into debugging mode and stepped through manually. The primary benefit of this is that it provides information without disrupting

timings such as the use of breakpoints does, also this allows debugging in callback functions, which the debugger conventionally cannot step into.

LISTING 4.12: PIC32 printf debugging example

```
int i = 0;

void loop() {
  debug('Hello, World! \%u \n', i++);
  delay(500);
}
```

4.3 Testing and Validation

The final part of the project is the testing and validation. While all of the independent parts of the system had been tested in a vacuum, unfortunately when the system was fully operational, the device failed due to power supply issues. The issues are related to what was seen earlier with the ESP32 power issues. 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 fully 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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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).

[10] 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.

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

[12] 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.

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

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

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

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

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

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

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

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

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

[22] 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).

[23] 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.

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

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

[26] 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).

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

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

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

[35] 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.

[36] 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).

[37] 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.

[38] 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).

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

[40] 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.

[41] 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).

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

[43] 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.

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

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

[46] 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).

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

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

[54] 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.

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

[56] 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.

[57] 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).

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

[59] 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.

[60] 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).

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

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

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

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

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

[66] 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).

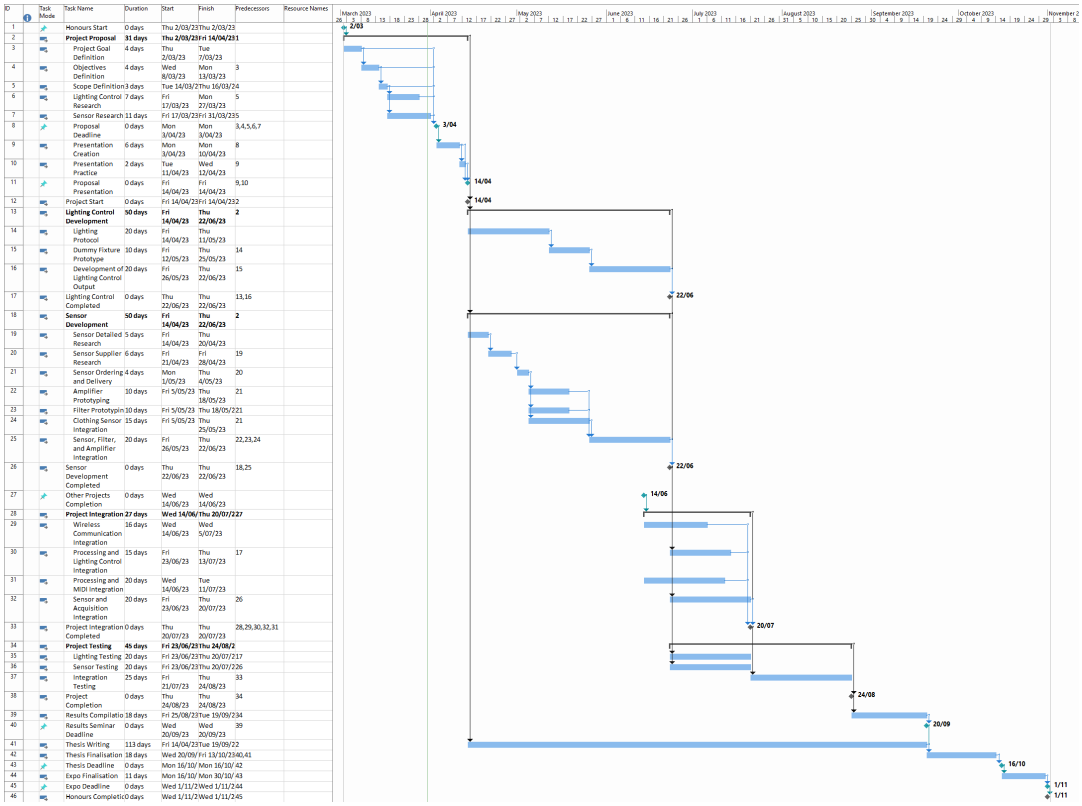
[67] The Johns Hopkins University, The Johns Hopkins Hospital, and Johns Hopkins Health System. *Vital Signs (Body Temperature, Pulse Rate, Respiration Rate, Blood Pressure)*. 2023. URL: https://www.hopkinsmedicine.org/health/conditions-and-diseases/vital-signs-body-temperature-pulse-rate-respiration-rate-blood-pressure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

[68] 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.

[69] 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.

[70] 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_ */
```