

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:

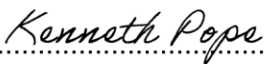
1. does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university
2. and the research within will not be submitted for any other future degree or diploma without the permission of Flinders University; and
3. to the best of my knowledge and belief, 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.....

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

Acknowledgements

Contents

Declaration of Authorship	i
Abstract	ii
Acknowledgements	iii
1 Introduction	1
1.1 Background	1
1.2 Aims	2
1.3 Literature Review	3
1.3.1 Introduction	3
1.3.2 Music from Brainwaves	3
1.3.3 Rhythmic Signal Approach	3
1.3.4 Wireless Solutions	4
1.3.5 The Music from Biosignals Project	4
1.3.6 Conclusion	4
1.4 Project Plan	6
1.4.1 Project Objectives	6
1.4.2 Assumptions and Constraints	6
1.4.3 Scope	7
2 Lighting Controller	9
2.1 Prototyping Fixture	9
2.1.1 DMX	9
2.1.2 NeoPixels	9
2.1.3 Integration	9
2.2 OpenDMX Controller	10
2.2.1 Interface	10
3 On-Body Device	11
3.1 ESP32	11

3.1.1	Power	11
3.1.2	Over-The-Air Programming	13
3.2	PIC32	14
3.2.1	Programming Configuration	15
3.2.2	ESP32 Control	16
3.3	ADS1294R	19
3.3.1	Design Differences	20
3.3.2	Startup Procedure	20
4	Integration	25
4.1	Off-Body PC	25
4.1.1	Non-Integrated Setup	25
4.2	System Integration	28
4.2.1	Wireless Transmission	28
4.2.2	Base64	30
4.3	Testing and Validation	32
5	Future Work	33
6	Conclusion	34

List of Figures

1 Introduction

This project proposes the use of biological signals, such as heart rate or muscle contractions, to generate musical signals. The project aims to provide a standard Musical Instrument Digital Interface (MIDI) for artists and performers to use. Additionally, the project aims to implement lighting control using the same biological signals to provide a novel interface for controlling lights.

1.1 Background

Music has been a form of human expression for over 40,000 years[26]. Throughout this time, 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. Previously, lighting control has been done 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.

A biosignal is a form of communication between biological systems[47], they are used in the body to detect various biological events such as muscle contractions and heartbeats[16]. These signals can be detected using various types of sensors, including electric, mechanical, acoustic, and infrared sensors[24].

1.2 Aims

These are the aims

I need something like an objective at the beginning though since that should be my metric for what to research in the literature review.

Maybe aims -> literature review -> objectives -> scope?

This is what we want, this is what already exists, this is what we want to do differently, this is what we will do?

1.3 Literature Review

1.3.1 Introduction

The use of biosignals to generate music has been an area of exploration and innovation in the field of music technology. In this literature review, we will examine various approaches and technologies that have been developed in the pursuit of creating music 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 are better suited to this project. Finally, we will investigate wireless solutions that enable the integration of biosensors into live performance devices and delve 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' [11] [48]. 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 [32] and require a large number of electrodes [44]. 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 electrocardiogram (ECG) [1] [41], electromyography (EMG) [50] [54], galvanic skin response (GSR) [28], and respiratory rate [5]. These signals have a degree of predictability [49] 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 [35], and ‘Stethophone’ by Nerness and Fuloria [37]. However, these applications are still limited in their flexibility and use due to the wired nature of these devices.

1.3.4 Wireless Solutions

Wireless biosensor based performance devices do exist. Examples of such systems are Yamaha AI’s ‘Transforms a Dancer into a Pianist’ [17], and ‘Emovere’ by Jaimovich [20]. There is not much documentation for these systems as they are still in use. But, from the small number of performance recordings of these systems it is clear that they are intended for experimental music. Therefore, further improvements on these devices can still be made in order to garner mass audience appeal through more conventional mainstream music generation.

1.3.5 The Music from Biosignals Project

The music from biosignals project has been ongoing for several years and attempts to integrate previously mentioned improvements. The project has developed an on-body device that acquires biosensors and allows them to be wirelessly transmitted to a PC for processing [12] [53]. Additionally, software developed in MATLAB has been developed that processes the incoming signals and generates music in real-time [8] [38]. Previously, the hardware and software design of the system has been separate and the two parts are yet to be integrated. This leaves potential future work open in connecting both sides of the project to create one cohesive whole.

1.3.6 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. The development of wireless solutions has facilitated the incorporation of biosensors into live performance settings, although further improvements are needed to enhance their mass audience appeal. The

Music from Biosignals project aims to incorporate these changes to create a device that can be used in a variety of live performance settings. By continuing to explore and refine the use of biosignals in music generation, we can unlock new possibilities for artistic expression and interactive musical experiences.

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 be composed of inexpensive parts
 - (a) Sensors must fall within the allocated budget
 - (b) PCBs must use off the shelf, easily sourced components
 - (c) Components must be easily replaceable for minimal cost if something were to fail
2. The system output must operate remotely to the acquired sensor data at a range of at least 30m
 - (a) The acquisition of sensor data must be completely 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 output MIDI messages
 - (a) The MIDI output port must be IEC 63035:2017 compliant
 - (b) The system must also provide MIDI over USB for greater compatibility with newer devices
4. The system must incorporate lighting control

1.4.2 Assumptions and Constraints

Assumptions:

- The users of the system have basic knowledge of music theory and performance.

- The system will be used in conjunction with a lighting console that maps all the fixture addresses correctly.
- 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 be operated by a skilled technician during live performances.

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 size and weight of the on-body subsystem must be wearable for several hours without fatigue.
- The system must be compatible with standard MIDI interfaces and protocols.

The given constraints are cost, size, weight, and compatibility. The cost is a constraint because the budget of the project is separate to the \$600 allocated limit. So, if the project goes over budget it may continue, but if it goes past the allocated limit, the project will not be able to continue. The size and weight are constraints because the project needs to be portable enough to effectively use. The system compatibility is a constraint because it is required for the system to correctly operate with other unknown systems.

1.4.3 Scope

In order to mitigate the risk of other concurrent projects running over scope, additional stretch goals have been added to the scope of this project. The stretch goals allow for increasing in scope while still strictly maintaining an achievable scope for this project. A breakdown of the scope of this project can be found in [Table 1.1](#), [Table 1.2](#), and [Table 1.3](#).

TABLE 1.1: Project in-scope list

Sensors	Clothing based biosignal acquisition
	Sensor gain and filtering
System Integration	Wireless communication
	System tunability
	System testing
MIDI	USB MIDI and physical port
Lighting Control	Lighting control over MIDI

TABLE 1.2: Project stretch-goal list

Sensors	Flex PCBs with flat battery
System Integration	Application testing
	Bi-directional bus communication
	Dynamic note magnitude
	PCB housing design for main board
MIDI	MIDI input support
	MIDI timecode quantisation
	MIDI output timecode
Lighting Control	Lighting standalone controller

TABLE 1.3: Project out-of-scope list

Sensors	Sensors as independent systems
System Integration	Sensor battery wireless charging
	Cableless system
	User interface for configuration
	Output based on change of frequency
MIDI	Wireless MIDI
	MIDI threshold points
Lighting Control	Programmable lighting controller
	Lighting controller input support

2 Lighting Controller

2.1 Prototyping Fixture

Small number of LEDs to verify correct control. 8 NeoPixel LEDs with 4 'dummy' DMX channels each. DMX channels are intensity, red channel, green channel, blue channel. Total of 32 channels (4×8)

2.1.1 DMX

Fixture uses Arduino along with `DMXSerial` library. Receives DMX frames using interrupts and stores them into array for processing. DMX frames are 512 bytes wide. A frame consists of the entire DMX 'universe' of channels. Individual channels are never written, instead the entire 'universe' is updated with each frame. Updates happen continually at a known rate. This way, fixture are aware if they lose connection to the controller, since they stop receiving frames.

2.1.2 NeoPixels

LEDs controlled using `NeoPixel` library. LEDs can be colored independently using red, green, and blue values. Each color is represented using a byte.

2.1.3 Integration

Both the DMX channel and color values are represented using bytes. No additional scaling is required.

Brightness needs to be mapped using

```
(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 values are sent over serial using Base64. This is so that the carriage return line can be used to mark the end of the incoming values. This was because the Arduino could become out of sync with what was being sent. If this happened there is no way of getting back in sync, because any special character could be interpreted as a regular value. This is why Base64 is necessary, because it allows for special characters that are separate from the designated regular character set. The values are sent as single decimal digits. So when they are received they must be decoded back into bytes.

The value of each LED is then encoded across 4 bytes. So the main program loops, incrementing 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 `show()` function.

2.2 OpenDMX Controller

When using expensive hardware in professional settings, it is important to use compliant hardware. While the prototyping fixture could be used for system verification, it was decided that existing hardware would be purchased for lighting control. **DECISION MATRIX HERE (MIGHT ALREADY HAVE ONE IN MY RESULTS POWER POINT?)**

2.2.1 Interface

The driver provided by the manufacturer was written in C#. Existing code for the project is written in MATLAB. We needed some way of controlling the device from MATLAB to utilise previous code. **MORE TO BE WRITTEN ABOUT THIS**

3 On-Body Device

The on-body device was developed by William Tran in 2021 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 the 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 **ESP MODEL**. 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 ICD3.
- 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 power supply, as the current draw that the supply is measuring is significantly less than the 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 max and the second supply provided necessary additional power.

What this 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 a 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, an 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

?? When compared to the alternative, Over-The-Air (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 run-time time ?. However, the compilation time is already in the range of 40 to 50 seconds, and the addition is less than 5 seconds. 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';

});
```

After implementing these libraries, the ESP32 can be programmed using OTA by selecting the relevant device. 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

The board designed by Tran (**REFERENCE HERE**) was designed in 2021. When the board was being manufactured, there was a global supply chain issue (**REF?**). 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 intentional delays are based on the clock speed so it must be set correctly for the delays to be correct.

The clock configuration uses PLL **??**. 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 **CALCULATE**. $(16\text{MHz}/10 * 16)/8 = 50\text{MHz}$

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;           // DELAY_CONST = SYS_FREQ /
    1000000 / 2
    _CPO_SET_COUNT(0);           // Reset Core Timer
    while (us > _CPO_GET_COUNT()); // Wait until Core Timer
    reaches desired number of clock ticks
}

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 OF SCOPE HERE)** 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.

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.

NEED TO INCLUDE THIS SOMEWHERE BUT PROBABLY NOT HERE OR LIKE THIS Used scope to measure space between pulses then set a specific pulse delay. I think previously the delay number was overflowing potentially because it was so large. Potentially the delay was way too short and the timing wasn't correct.

3.2.2 ESP32 Control

The ESP32 is controlled by the PIC32 in two ways.

- 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 0 ?? The baudrate generator value has been calculated using the equation $BRG = (F_{PB}/2 \times F_{SCK}) - 1$. **CALCULATE THIS: $BRG = 50$** . 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;                // 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);                // Required delay for data
    transmission
    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 prioritized over performance, 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 either a custom or third party library must be used. In the interest of getting as much of the project functional as possible, it was decided that a third party library would be used **DECISION MATRIX?**. However, there are potential performance improvements that could be achieved if a custom library was used, such as reducing the PIC transmission delays, so it is recommended that this option is looked into in the future.

The library that was used is the ESP32DMASPI ?? library. This library is based on the official driver from the manufacture ??.

How do I describe my workarounds in order to get this to work aside from just saying that that's what I did? SPI implemented using task based DMA receiving. 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);
    }
}
```

DESCRIBE MORE DETAIL ABOUT HOW THESE WORK.

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 [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);
}
```

Then, the only way to send data was to write a single byte at a time. What is interesting about this is that the single byte is 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 also present when using 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.

However, despite these 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 ???. 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. 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 that 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.

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, these limitations do not stop the device ? from providing important insight. 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 all ones. 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. Moreover, 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.

SPI SETUP IS AS SHOWN clock polarity 0, clock phase 1, output edge rising, data capture falling

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 hardware reset, a long delay was added to the startup sequence so that the physical switch could be pressed at the correct point. 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, 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 there was nothing about that driver that could be changed **ELABORATE**.

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, they were all 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.

ADD FIGURE OF BOARD WITH JUMPERS

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. **MAYBE FIGURE HERE OF SCOPE** 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](#)

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 is 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. **MATLAB DECISION MATRIX**

4.1.1 Non-Integrated 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 length. 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, we can create a circular loop of the data and have a test system that operates almost exactly as the eventual integrated program will.

For testing, an electrocardiogram (ECG) dataset 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 datasets 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 datasets 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 dataset 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 dataset 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 array is labelled 'data' in this example.

The packet index is the index into the ECG dataset 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 dataset, it is reset back to the beginning. There is no consideration for the continuity of the signal, it is a known issue and can be easily detected and ignored.

The packet is defined as the snippet that the packet index is pointing to in the dataset with the length defined by packet length. 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 can be appended to the end of the array. **FIGURE HERE SHOWING DATA ENTERING ON THE RIGHT AND EXITING ON THE LEFT** Plots of this data are shown in ?? and??.

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

LISTING 4.3: MATLAB signal analysis example

```
%%% server side signal analysis
[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)
    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 dataset. **FIGURE HERE SHOWING MATCHING BPMS** When the system is integrated, as long as the data length matches and the sensor data is clean ?, this same code will produce the same 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 is connected to WiFi for OTA updates, it was logically concluded that TCP/IP should be used as the communication **DECISION MATRIX**. 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');
    client.write('\n');
  }

  delay(500);
}
```

LISTING 4.5: MATLAB TCPIP receive code

```

clear server

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

function server_callback(src, ~)
    bytes = readline(src);
    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. **FIGURE SHOWING FIREWALL SETTING THIS IS ALSO NECESSARY FOR THE ESP32 OTA PROGRAMMING**

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 passthrough 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 connecting to TCP/IP client

```

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: ESP32 code for connecting to TCP/IP client

```

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);
}

```

MAY BE WORTH MENTIONING SOMEWHERE THAT THE ESP32 DID NOT START THIS SIMPLE IT BEGAN MUCH MORE COMPLICATED AND GAINED SIMPLICITY AS DEVELOPMENT CONTINUED

4.2.2 Base64

MAYBE CHANGE PREVIOUS TESTING CODE TO BYTE LENGTH BASED RECEIVING TO ILLUSTRATE POINT EVEN IF YOU CHANGE IT TO THE SINE EXAMPLE LIKE WHAT YOU ACTUALLY DID

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 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. **DECISION MATRIX FOR DIFFERENT CHARACTER SETS MAYBE MORE RESEARCH ON OTHER ONES/AT LEAST ON BASE64** Base64 can be decoded in MATLAB using the

```

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');
}

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

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));
}

y = char(decoded);
disp(y)

int i = 0;

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

4.3 Testing and Validation

What tests were performed and how that validates the system.

5 Future Work

What can be done going forward

6 Conclusion

Bibliography

- [1] V.X. Afonso et al. “ECG beat detection using filter banks”. In: *IEEE Transactions on Biomedical Engineering* (1999). DOI: [10.1109/10.740882](https://doi.org/10.1109/10.740882).
- [2] 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](https://doi.org/10.1016/j.bspc.2007.07.009). URL: <https://www.sciencedirect.com/science/article/pii/S1746809407000547>.
- [3] 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](https://doi.org/10.1016/S0010-4825(01)00009-9). URL: <https://www.sciencedirect.com/science/article/pii/S0010482501000099>.
- [4] 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](https://doi.org/10.5664/jcsm.6122).
- [5] 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](https://doi.org/10.1109/ICSENS.2011.6127168).
- [6] 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).
- [7] 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](https://doi.org/10.1109/86.867872).
- [8] Chen Chen. “Music from Bio-Signals”. Bachelor’s Thesis. Flinders University, 2016.

- [9] Amer Chlahawi 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](https://doi.org/10.1016/j.sbsr.2018.05.001).
- [10] Vinod Chouhan and Sarabjeet Mehta. "Total Removal of Baseline Drift from ECG Signal". In: 2007, pp. 512–515. DOI: [10.1109/ICCTA.2007.126](https://doi.org/10.1109/ICCTA.2007.126).
- [11] Carlos Conceição. *Alvin Lucier - "Music For Solo Performer" (1965)*. <https://www.youtube.com/watch?v=bIPU2ynqy2Y>. 2010.
- [12] Adam De Pierro. "Music from Biosignals". Bachelor's Thesis. Flinders University, 2019.
- [13] Digital Illumination Interface Alliance. *Introducing DALI*. 2021. URL: <https://www.dali-alliance.org/dali/> (visited on 03/24/2023).
- [14] Gilles Dubost and Atau Tanaka. "A Wireless, Network-based Biosensor Interface for Music". In: (2002).
- [15] EnOcean. *Smart Lighting*. 2023. URL: <https://www.enocean.com/en/applications/building-automation/lighting/> (visited on 03/24/2023).
- [16] Monty Escabí. "Chapter 11 - Biosignal Processing". In: *Introduction to Biomedical Engineering (Third Edition)* (2012), pp. 667–746.
- [17] 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).
- [18] 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](https://doi.org/10.1109/JSEN.2017.2684825).
- [19] 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).
- [20] Javier Jaimovich. *Emovere: Designing Sound Interactions for Biosignals and Dancers*. Tech. rep. Departamento de Música y Sonología Universidad de Chile, 2016.
- [21] 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>.

- [22] 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](https://doi.org/10.1109/ICRITO51393.2021.9596077).
- [23] 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](https://doi.org/10.1109/CSNT.2013.22).
- [24] Eugenijus Kaniusas. *Biomedical Signals and Sensors I*. Springer Berlin, 2012.
- [25] 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](https://doi.org/10.1177/0017896912472328).
- [26] Anton Killin. "The origins of music: Evidence, theory, and prospects". In: *Music & Science* 1 (2018), p. 1. DOI: [10.1177/2059204317751971](https://doi.org/10.1177/2059204317751971). URL: <https://doi.org/10.1177/2059204317751971>.
- [27] Neeraj Kumar, Imteyaz Ahmad, and Pankaj Rai. "Signal Processing of ECG Using Matlab". In: *International Journal of Scientific and Research Publications* 2 (2012).
- [28] 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](https://doi.org/10.1109/CBMS.2013.6627790).
- [29] 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).
- [30] 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](https://doi.org/10.1109/TBME.2010.2048568).
- [31] 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).

- [32] 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](https://doi.org/10.1109/ACCESS.2018.2842082).
- [33] Modbus Organization, Inc. *Modbus*. 2023. URL: <https://www.modbus.org/> (visited on 03/24/2023).
- [34] 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](https://doi.org/10.1109/BIBE.2012.6399657).
- [35] Teresa Nakra and Rosalind Picard. "The "Conductor's Jacket": A Device For Recording Expressive Musical Gestures". In: (1998).
- [36] Amine Naït-Ali, ed. *Advanced Biosignal Processing*. 1st ed. Berlin, Heidelberg: Springer, 2009.
- [37] 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).
- [38] Isaac Nicholls. "Music with Bio-signals". Bachelor's Thesis. Flinders University, 2019.
- [39] Miguel Ortiz et al. "Biosignal-driven Art: Beyond biofeedback". In: (2011).
- [40] 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.
- [41] 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](https://doi.org/10.1109/TBME.1985.325532).
- [42] 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](https://doi.org/10.1109/TSMCC.2009.2032660).

- [43] Hailong Peng et al. "Preparation of photonic-magnetic responsive molecularly imprinted microspheres and their application to fast and selective extraction of 17 β -estradiol". In: *Journal of Chromatography A* 1442 (2016), pp. 1–11. ISSN: 0021-9673. DOI: [10.1016/j.chroma.2016.03.003](https://doi.org/10.1016/j.chroma.2016.03.003). URL: <https://www.sciencedirect.com/science/article/pii/S0021967316302308>.
- [44] 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](https://doi.org/10.1109/EHB47216.2019.8969949).
- [45] RDM Task Group. *RDMnet*. 2023. URL: <https://www.rdmprotocol.org/rdm/rdmnet/> (visited on 03/24/2023).
- [46] 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).
- [47] John Semmlow. "Chapter 1 - The Big Picture: Bioengineering Signals and Systems". In: *Circuits, Signals and Systems for Bioengineers (Third Edition)* (2018), pp. 3–50.
- [48] 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](https://doi.org/10.1017/S135577181300037X).
- [49] Koray Tahiroğlu, Hannah Drayson, and Cumhur Erkut. "An Interactive Bio-Music Improvisation System". In: 2008.
- [50] Atau Tanaka and R. Knapp. "Multimodal Interaction in Music Using the Electromyogram and Relative Position Sensing". In: 2002.
- [51] The IES Controls Protocol Committee. *Lighting Control Protocols*. Illuminating Engineering Society of North America, 2011.
- [52] Matthew John Thies and Bruce Pennycook. *Controlling game music in real time with biosignals*. Tech. rep. The University of Texas at Austin, 2012.
- [53] William Tran. "Music from Biosignals". Bachelor's Thesis. Flinders University, 2022.
- [54] 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](https://doi.org/10.1109/TBME.2012.2232293).

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