Remote-Robot

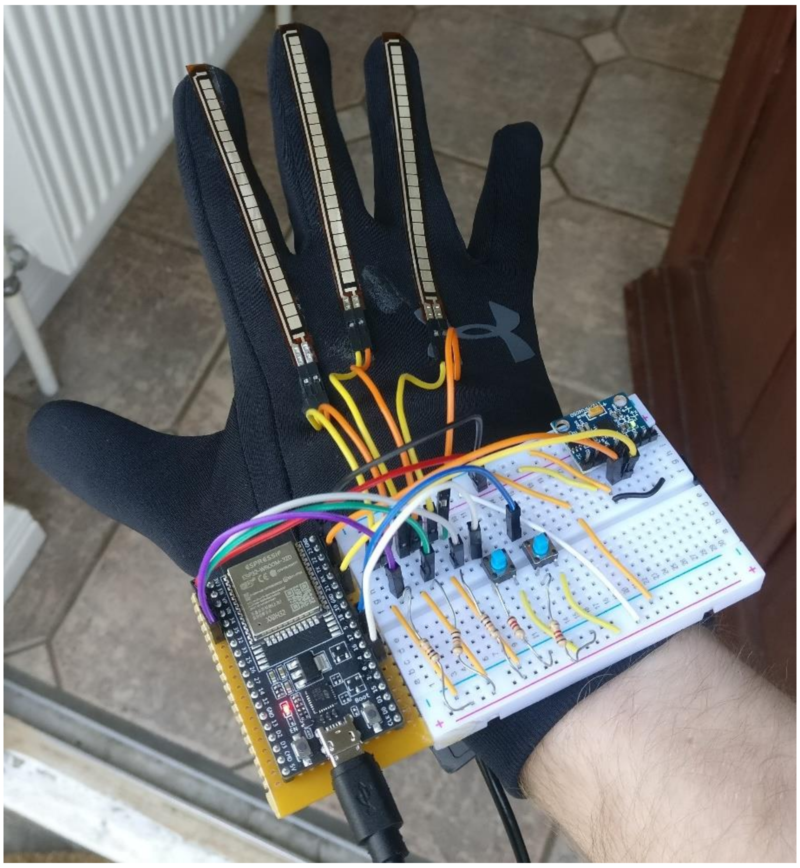
Final Year Project

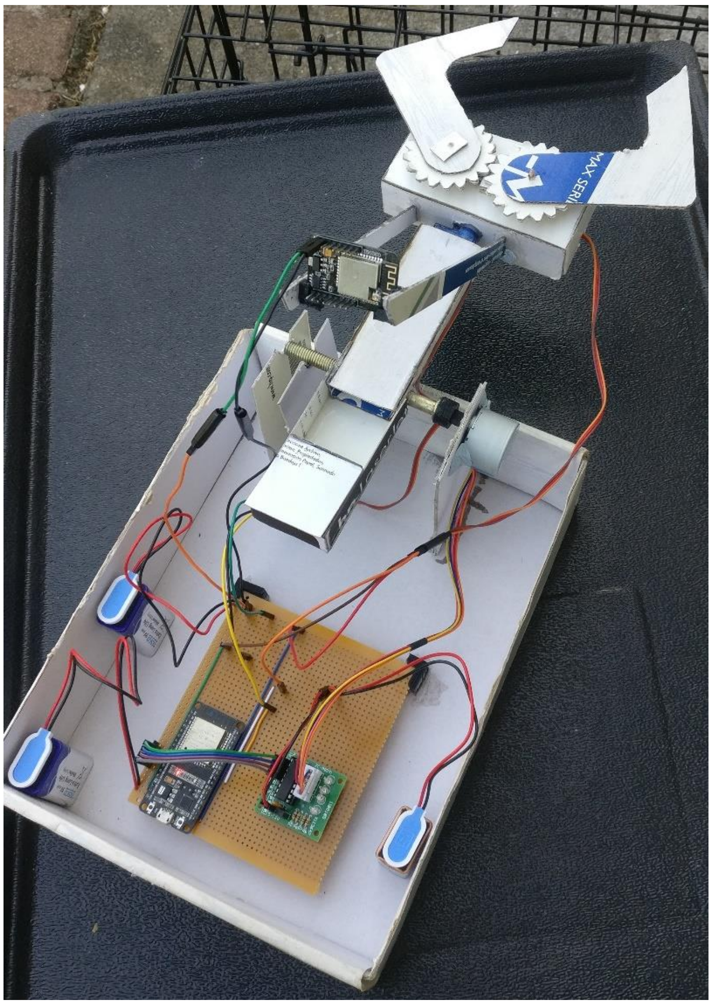
Ruairí Doherty

Bachelor of Engineering (Hons) in
Software & Electronic Engineering

Galway-Mayo Institute of Technology

2019/2020





Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Hons) in Software & Electronic Engineering at Galway-Mayo Institute of Technology.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

Ruairí Doherty

Acknowledgements

Paul Lennon, Lecturer, Software & Electronic Engineering, Galway-Mayo Institute of Technology.

Niall O’ Keeffe, Lecturer, Software & Electronic Engineering, Galway-Mayo Institute of Technology.

Brian O’ Shea, Lecturer, Software & Electronic Engineering, Galway-Mayo Institute of Technology.

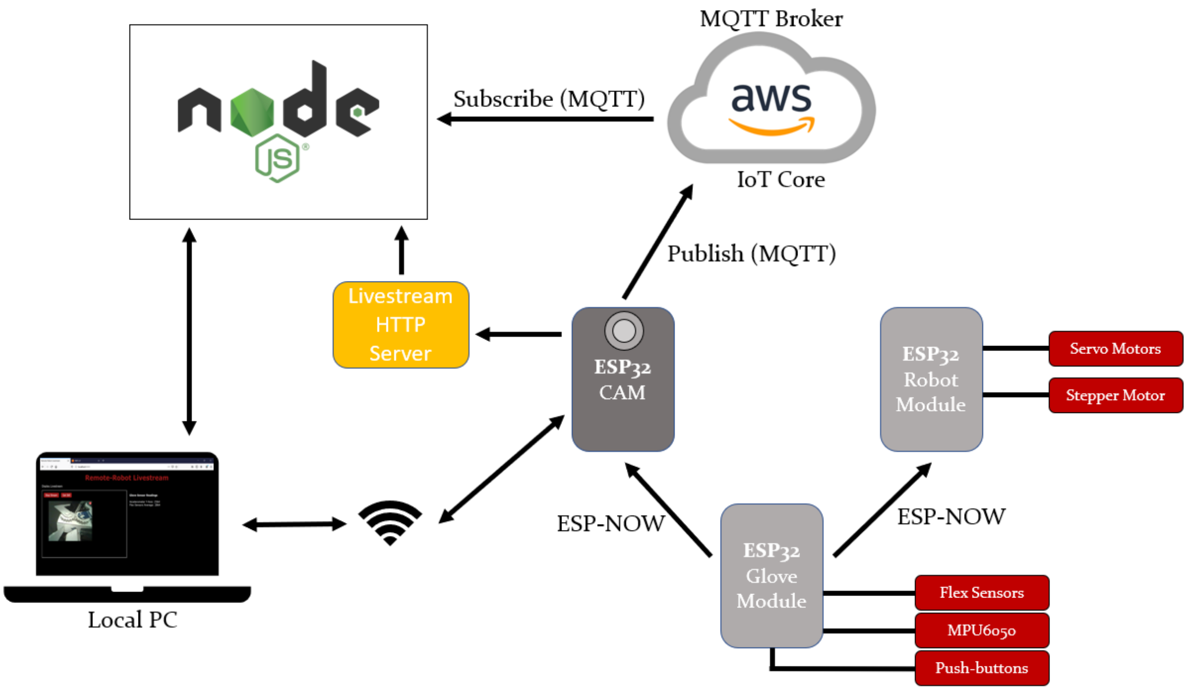
Table of Contents

1. Project Overview

The Remote-Robot is a wireless, hand-gesture controlled robot. It's purpose is for situations where precision is required, in environments that would be unsuitable for a human to operate in. For example – bomb disposal, radioactive environments or natural disaster relief. The project consists of two modules – the glove and robot arm. The glove uses flex sensors and an accelerometer to continuously monitor the user's hand movements. It also has two push buttons to control the pitch of the robot arm. The robot module consists of two servo motors and one stepper motor. The servo motors are controlled by the flex sensors and accelerometer on the glove. The stepper motor handles the pitch of the robot, controlled by the push-buttons on the glove. Both of these modules communicate wirelessly, allowing remote control of the robot arm. A camera module mounted on the robot's hand displays a livestream on a Node.js web-server, allowing the user to see what the robot can see. Real-time sensor data is also displayed on the server.

GitHub repository: https://github.com/ruairiD11/ProjectY4

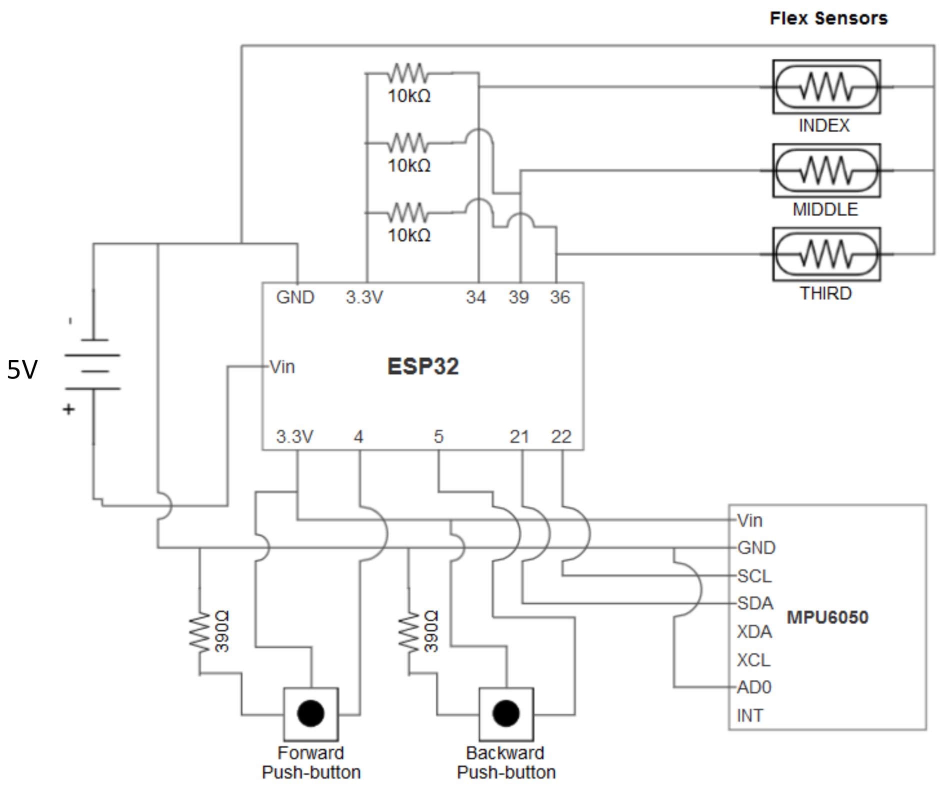
2. Project Architecture



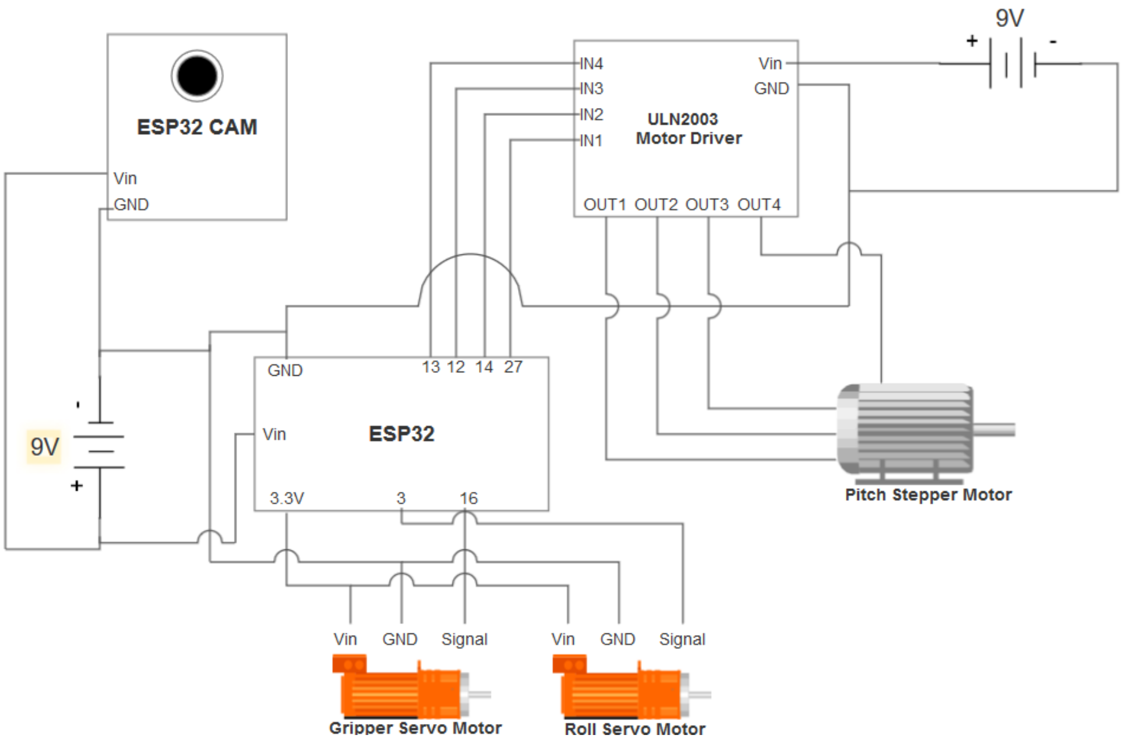
3. Schematics

Schematics were designed using: https://www.smartdraw.com/

Glove Module



Robot Module



4. Project Requirements

4.1 Functional

- The glove and the robot arm must communicate wirelessly.
- The glove module should be equipped with many sensors, to monitor the user's hand movements.
- The Remote-Robot should accurately imitate the user's hand movements.

4.2 Non-Functional

- The Remote-Robot should be able to be controlled from a significant distance (150 metres or more).
- The Remote-Robot should receive the glove sensor data every 50ms or less.
- The Remote-Robot should preserve as much power as possible whilst in Idle mode.

5. Introduction

For this project, I wanted to combine the software and hardware knowledge I've gained from my four years studying this course. I knew I wanted to have a glove module involved, as this would consist of many sensors. Initially, I thought of having a glove that could read sign language. It would convert sign language into speech through a speaker or text on a screen, but this idea didn't materialise. I enjoy the process of designing and building things, so I thought of building a robot arm that could be controlled by hand movements. I felt this idea would allow me to display my knowledge of both hardware and software, whilst also allowing me to design, prototype and build a robot arm.

The first semester of working on the project was mostly experimentation with different hardware components, and trying to establish what would end up in the final product. During this time, I was testing using the Arduino Uno development board, along with a 32-bit Arduino-compatible microcontroller called the Teensy LC. I was programming in the Arduino IDE, but was continuously looking to veer away from this environment. By the end of first semester, I had a prototype built of the robot arm and had it controlled by hand movements. I needed to have the two modules communicate wirelessly. I had experimented with different stand-alone Wi-fi modules such as the ESP8266 ESP-01 with no success. At my demonstration at the end of first semester, I was advised to switch over to using the ESP32 development board as it has built-in Wi-fi capabilities. So over the Christmas break I looked to do just that. This decision had many pros, with the only con being that I would now need to migrate all the work done from first semester on the Arduino Uno, over to the ESP32.

6. Work Breakdown Structure

Figure 6.1, on the next page, shows the breakdown of work for my second semester (January – April). As I needed to essentially restart the project with a new development board, I planned out one/two week sprints to make sure I got back on track as soon as possible. This is how I planned it out:

20th January – 3rd February (2 week sprint): For migrating all work accomplished from first semester into Eclipse, to have the same functionality with the ESP32.

3rd February – 17th February (2 week sprint): Research and implementation of wireless communication between the glove and robot arm.

17th February – 3rd March (2 week sprint): For optimizing the overall performance on my prototype – this included introducing task synchronisation/ communication using FreeRTOS communication objects and mechanisms such as queues, task notifications and event groups.

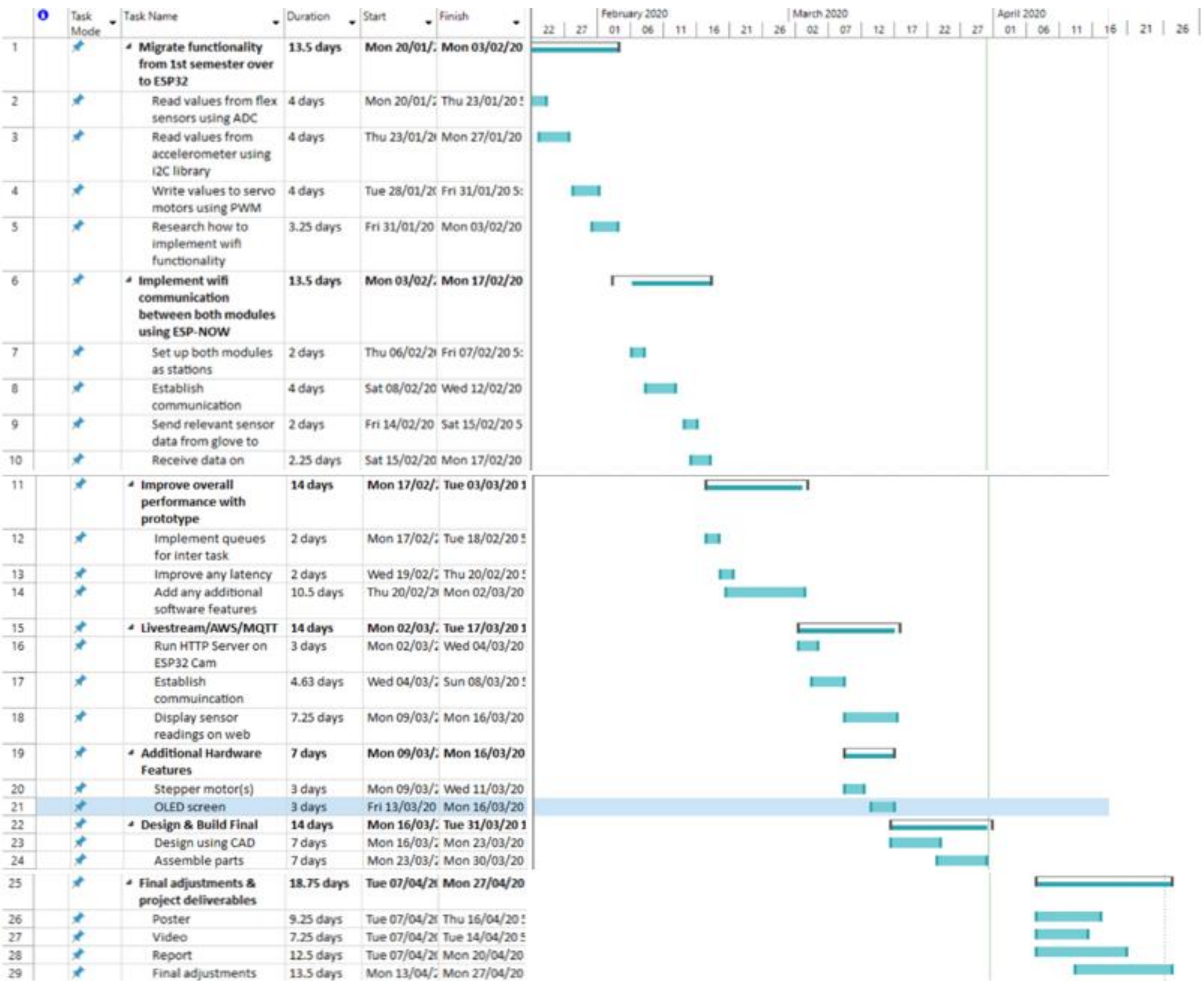
3rd March – 17th March (2 week sprint): Planned to set-up the Node.js server that displays sensor data and a livestream.

9th March – 16th March (1 week sprint): For adding any additional hardware features – stepper motors, OLED screen.

17th March – 31st March (2 week sprint): For designing and assembling the final robot arm.

7th April – 27th April: The final twenty days was for work on project deliverables and any final adjustments before the demonstration.

This plan helped me meet my deadlines on a regular basis, and in turn, get my project back on schedule.

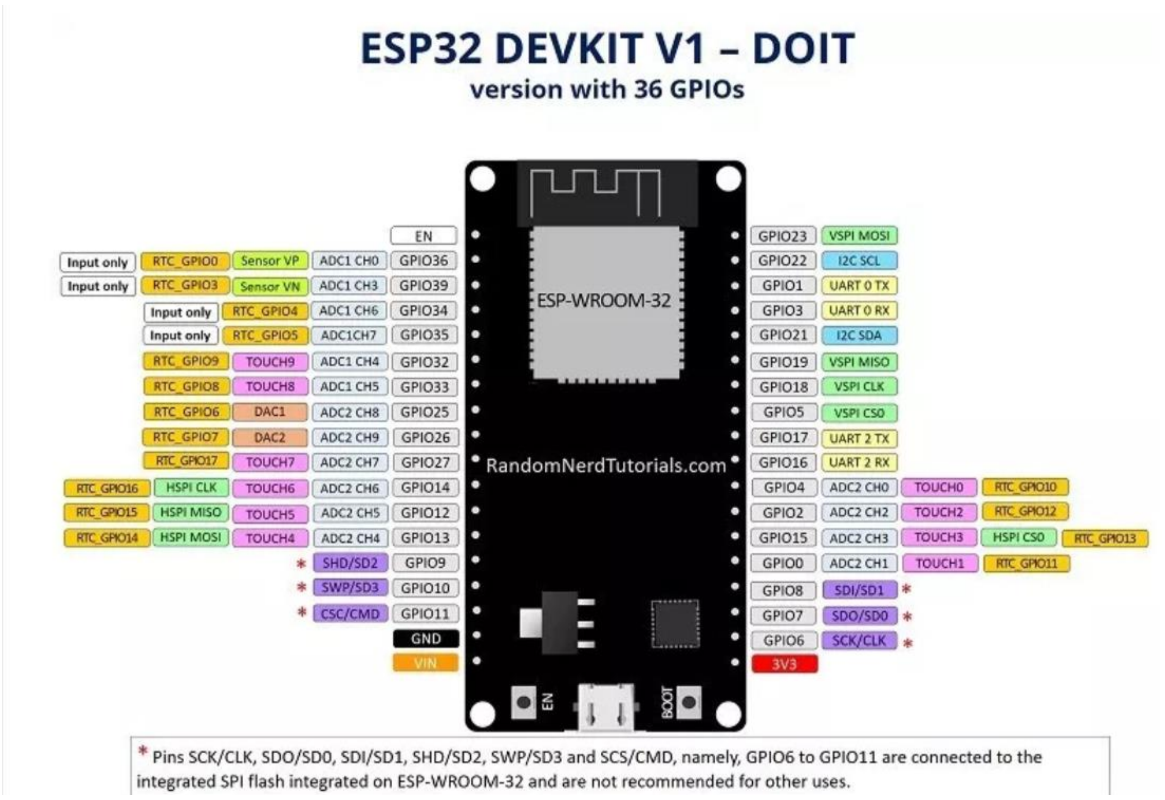


	Task Mode	Task Name	Duration	Start	Finish
1	📌	**Migrate functionality from 1st semester over to ESP32**	**13.5 days**	**Mon 20/01/**	**Mon 03/02/20**
2	📌	Read values from flex sensors using ADC	4 days	Mon 20/01/	Thu 23/01/20
3	📌	Read values from accelerometer using i2C library	4 days	Thu 23/01/2	Mon 27/01/20
4	📌	Write values to servo motors using PWM	4 days	Tue 28/01/2	Fri 31/01/20 5:
5	📌	Research how to implement wifi functionality	3.25 days	Fri 31/01/20	Mon 03/02/20
6	📌	**Implement wifi communication between both modules using ESP-NOW**	**13.5 days**	**Mon 03/02/**	**Mon 17/02/20**
7	📌	Set up both modules as stations	2 days	Thu 06/02/2	Fri 07/02/20 5:
8	📌	Establish communication	4 days	Sat 08/02/20	Wed 12/02/20
9	📌	Send relevant sensor data from glove to	2 days	Fri 14/02/20	Sat 15/02/20 5
10	📌	Receive data on	2.25 days	Sat 15/02/20	Mon 17/02/20
11	📌	**Improve overall performance with prototype**	**14 days**	**Mon 17/02/**	**Tue 03/03/20 1**
12	📌	Implement queues for inter task	2 days	Mon 17/02/	Tue 18/02/20
13	📌	Improve any latency	2 days	Wed 19/02/	Thu 20/02/20
14	📌	Add any additional software features	10.5 days	Thu 20/02/2	Mon 02/03/20
15	📌	**Livestream/AWS/MQTT**	**14 days**	**Mon 02/03/**	**Tue 17/03/20 1**
16	📌	Run HTTP Server on ESP32 Cam	3 days	Mon 02/03/	Wed 04/03/20
17	📌	Establish commuincation	4.63 days	Wed 04/03/	Sun 08/03/20
18	📌	Display sensor readings on web	7.25 days	Mon 09/03/	Mon 16/03/20
19	📌	**Additional Hardware Features**	**7 days**	**Mon 09/03/**	**Mon 16/03/20**
20	📌	Stepper motor(s)	3 days	Mon 09/03/	Wed 11/03/20
21	📌	OLED screen	3 days	Fri 13/03/20	Mon 16/03/20
22	📌	**Design & Build Final**	**14 days**	**Mon 16/03/**	**Tue 31/03/20 1**
23	📌	Design using CAD	7 days	Mon 16/03/	Mon 23/03/20
24	📌	Assemble parts	7 days	Mon 23/03/	Mon 30/03/20
25	📌	**Final adjustments & project deliverables**	**18.75 days**	**Tue 07/04/2**	**Mon 27/04/20**
26	📌	Poster	9.25 days	Tue 07/04/2	Thu 16/04/20
27	📌	Video	7.25 days	Tue 07/04/2	Tue 14/04/20
28	📌	Report	12.5 days	Tue 07/04/2	Mon 20/04/20
29	📌	Final adjustments	13.5 days	Mon 13/04/	Mon 27/04/20

Figure 6.1 – Remote-Robot Second Semester Work Breakdown Structure

7. ESP32 Overview

There are many variations of ESP32 development boards. I am using the ESP32 DevKit V1 provided by DOIT, which evaluates the ESP-WROOM-32 module. It uses the Tensilica 32-bit Single/Dual-Core CPU Xtensa LX6 microcontroller. The ESP32 boasts many features such as built-in Wi-fi, Bluetooth, Ethernet and Low-Power capabilities. Along with this, it has 36 GPIO pins, which includes 12 ADC pins, 2 DAC pins, and the capability to interface with protocols such as I2C, SPI and UART. With a clock speed of up to 240 MHz, this board is extremely powerful, very adaptable and cheap. I picked them up for less than €10 each. See Figure 7.1 for the pin-out diagram.


ESP32 DEVKIT V1 – DOIT
version with 36 GPIOs
EN
Input only | RTC_GPIO0 | Sensor VP | ADC1 CH0 | GPIO36
Input only | RTC_GPIO3 | Sensor VN | ADC1 CH3 | GPIO39
Input only | RTC_GPIO4 | ADC1 CH6 | GPIO34
Input only | RTC_GPIO5 | ADC1CH7 | GPIO35
RTC_GPIO9 | TOUCH9 | ADC1 CH4 | GPIO32
RTC_GPIO8 | TOUCH8 | ADC1 CH5 | GPIO33
RTC_GPIO6 | DAC1 | ADC2 CH8 | GPIO25
RTC_GPIO7 | DAC2 | ADC2 CH9 | GPIO26
RTC_GPIO17 | TOUCH7 | ADC2 CH7 | GPIO27
RTC_GPIO16 | HSPI CLK | TOUCH6 | ADC2 CH6 | GPIO14
RTC_GPIO15 | HSPI MISO | TOUCH5 | ADC2 CH5 | GPIO12
RTC_GPIO14 | HSPI MOSI | TOUCH4 | ADC2 CH4 | GPIO13
* SHD/SD2 | GPIO9
* SWP/SD3 | GPIO10
* CSC/CMD | GPIO11
GND
VIN
GPIO23 | VSPI MOSI
GPIO22 | I2C SCL
GPIO1 | UART 0 TX
GPIO3 | UART 0 RX
GPIO21 | I2C SDA
GPIO19 | VSPI MISO
GPIO18 | VSPI CLK
GPIO5 | VSPI CS0
GPIO17 | UART 2 TX
GPIO16 | UART 2 RX
GPIO4 | ADC2 CH0 | TOUCH0 | RTC_GPIO10
GPIO2 | ADC2 CH2 | TOUCH2 | RTC_GPIO12
GPIO15 | ADC2 CH3 | TOUCH3 | HSPI CS0 | RTC_GPIO13
GPIO0 | ADC2 CH1 | TOUCH1 | RTC_GPIO11
GPIO8 | SDI/SD1 *
GPIO7 | SDO/SD0 *
GPIO6 | SCK/CLK *
3V3
ESP-WROOM-32
RandomNerdTutorials.com
EN
BOOT

* Pins SCK/CLK, SDO/SD0, SDI/SD1, SHD/SD2, SWP/SD3 and SCS/CMD, namely, GPIO6 to GPIO11 are connected to the integrated SPI flash integrated on ESP-WROOM-32 and are not recommended for other uses.

Figure 7.1 – ESP32 DevKit V1 DOIT Pin-out Diagram, Source – [1]

8. Development Platform & Tools

As mentioned before, I wanted to veer away from programming my ESP32 boards within the Arduino IDE. This is because if I was to work within the **Eclipse IDE**, I could program in plain C, and this would result in better overall computational efficiency. Most of the libraries provided to you by Arduino are designed for quick development of code, and generally don't take performance into consideration. This is why I opted to use the Eclipse IDE, as it allows me to communicate with the hardware at a lower level and increase efficiency.

In order to program within Eclipse I needed to download Espressif's Integrated Development Framework (**ESP-IDF**) and install toolchains that allow me to compile, debug and flash my ESP32 from the Eclipse environment. These tools included **CMake, Ninja** build tool, cross-compilers and **OpenOCD** for debugging. The ESP-IDF itself consists of the API with all the libraries and source code required to interface with the ESP32. Espressif provides a thorough tutorial on the setup of their integrated development framework [2].

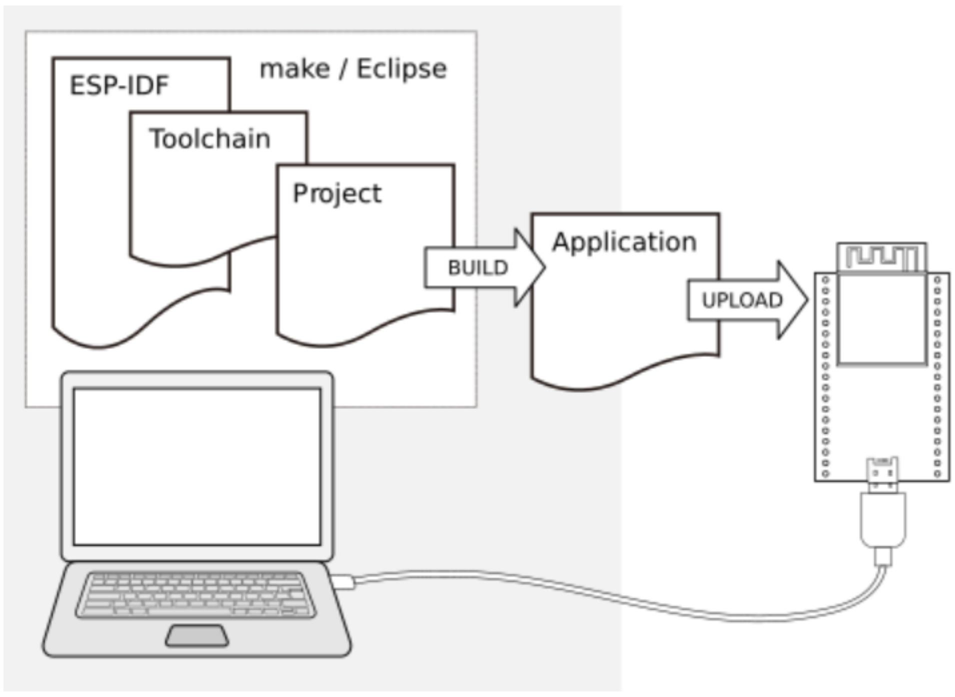


Figure 8.1 – "Development of applications for ESP32", Source – [2]

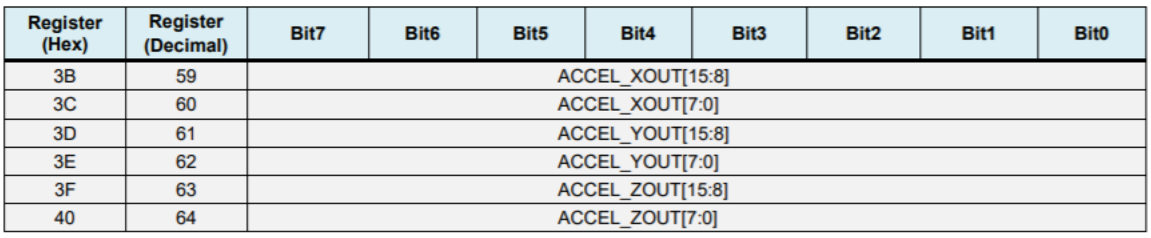
9. Glove Module

9.1 Flex Sensors

A flex sensor essentially acts as a variable resistor. The more the it is bent, the more internal resistance it has, thus reducing the amount of voltage output. We can monitor the voltage output of these sensors to determine how much they are being bent. It has two non-polarized terminals – one connected to a positive power source, and one connected to ground. These sensors are ideal for the glove module of my project. I have three of these sensors attached to the index, middle and third finger of my glove. Each are connected to a 10kΩ pull-up resistor and ground. I am using three separate ADC channels from the ESP32 to read the raw voltage values of the flex sensors. I calculate the average value of the three sensors, use the map function, and send this value as a task notification to the accelerometer task. I haven't got the strongest connection between the flex sensors and the ESP32, so I can get some inconsistent readings. I tested each sensor to see what output each was giving me. I noticed some irregularities with each particular sensor, and came up with a software solution for each. For example, the index finger's sensor would stay at 4095 (highest reading) when the finger was in the resting position. So I've used an if statement to check if it reads 4095, and if so, set the value to its lowest value which would be 2500. This isn't a problem if I physically apply pressure to the connections to the sensors – I receive a much cleaner output. I improved this again by storing the last value of each flex sensors, along with their current reading and getting a smoother average. I talk more about this in section 15 - Performance Improvements.

9.2 MPU6050

A major functional requirement of my project is to monitor the user's precise hand movements, so I needed to find a device that would allow me to do so. The MPU6050 is an Inertia Measurement Unit (IMU) with a built-in accelerometer and gyroscope. IMUs are common in real-world applications such as self-balancing robots, quadcopters and smartphones [3]. Not only does the MPU6050 produce accurate readings, but it is also very inexpensive. From my experience using it, I can safely say it was the correct choice. The MPU6050 uses the I2C protocol for communication. I made use of the accelerometer in particular for this project, in order to monitor the orientation of the user's hand in the y-axis. From my understanding, the accelerometer produces a change in output proportional to a change in the gravitational force on a particular axis (x,y or z). In order to interface with this component, I required the services of the I2C library provided in the ESP-IDF. I configured the device with the aid of Neil Kolban's technical tutorial on the MPU6050 [4]. In order to retrieve the data from the MPU6050, I need to read from the six accelerometer registers, as shown in Figure 9.2.1.



Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
3B	59	ACCEL_XOUT[15:8]							
3C	60	ACCEL_XOUT[7:0]							
3D	61	ACCEL_YOUT[15:8]							
3E	62	ACCEL_YOUT[7:0]							
3F	63	ACCEL_ZOUT[15:8]							
40	64	ACCEL_ZOUT[7:0]							

Figure 9.2.1 – MPU6050 Accelerometer Registers, Source - [5]

There is a high and low register for each of the x, y and z values. Beginning at ACCEL_XOUT[15:8], I loop through the registers down until ACCEL_ZOUT[7:0]. Once reading each register I'm left with six values. I then must use bit masking and shifting in order to combine the high and low values of each, and obtain the final x, y and z readings (the difference between the high and low values). I only require the y value, to control the roll servo of the robot arm.

10. Robot Module

10.1 Servo Motors

Inside a servo motor, you will find a small DC motor, a potentiometer and a control circuit. How it works is, as the DC motor rotates, so does the potentiometer, so the control circuit is able to decipher how much movement is occurring and in which direction. The servo has three inputs – Vin, GND and Signal. The signal input receives electrical pulses which determines the desired position of the motor. I have two SG90 servo motors on my robot module – the gripper and the roll servo. SG90 servo motors are small and cheap, so I bought them in bulk. The gripper servo's desired position is determined by the average value of the three flex sensors. The roll servo is determined by the y-axis value from the accelerometer. In order to interface with the servo motors I use Pulse Width Modulation (PWM). PWM is used to encode data in a signal, by using the duration that the signal is high, relative to the duration that the signal is low. The duty cycle is what determines the ratio of how long the signal is high versus low within that signal's frequency, and is measured in bits. So the duty cycle of the gripper servo is the average value of the flex sensors, and the duty cycle of the roll servo is the y value of my accelerometer. The ESP-IDF provides a library for implementing PWM called ledc, which stands for Light Emitting Diode Controller. It is the same principle applied if you are controlling the brightness of an LED, or in my case, positioning a servo motor. Once again, I used Neil Kolban's resources to aid me in the configuration of PWM [6].

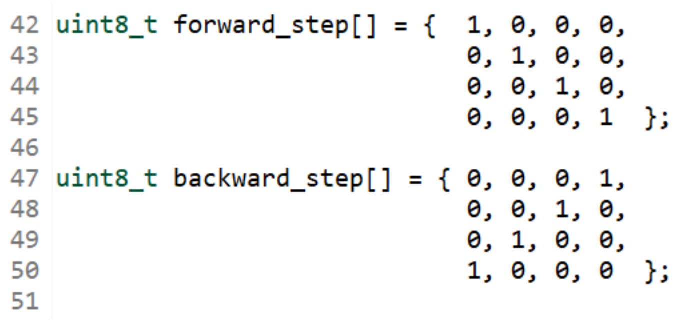
10.2 Stepper Motor

To add another degree of freedom (DoF) to the robot arm, I wanted to be able to control it's pitch. I could have added an additional servo motor to do this, but I came to the conclusion that this would not be a feasible option. Instead, I came up with a more appropriate solution:

The pitch does not need to be continuously monitored and updated, as this would unnecessarily require more of the CPU's resources. The glove module already has to continuously poll the readings from the accelerometer and flex sensors. I thought that having two push-buttons attached as interrupts would be a much more efficient way of controlling the pitch. The glove module has two push-buttons that are connected to the ESP32 as external interrupts – one to pitch forward and one to pitch backward. Whenever they are pressed, an interrupt occurs and changes the state of their respective variable from 0 to 1. These variables are sent as part of the ESP-NOW packet to the robot module. On the robot's side, it is continuously receiving these data packets including the state of the push-buttons. If it receives a value of 1 for one of these buttons, it recognizes that the button was pressed, and can perform the appropriate algorithm, as I will outline next.

Stepper motors consist of multiple coils that are grouped into phases or steps. If these phases are electrically powered in sequence, the motor will output a single step. I am using the 28BYJ-48 stepper motor along with the ULN2003 motor driver. The motor has 4 inputs (IN1, IN2, IN3, IN4) that connect to the motor driver. The driver is powered by its own independent 9V battery. With the knowledge that the motor will produce a single step if it's four inputs are powered in sequence, I came up with an algorithm to automate this process:

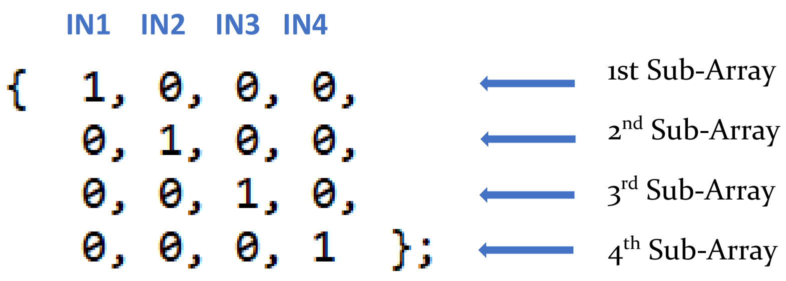
Firstly, I needed to create the sequence that the stepper should undertake when it is required to pitch forward or backward. I thought that creating an array for each the forward and backward sequence was a good way to do it. Imagine that each of these two arrays are broken into four sub-arrays:



```
uint8_t forward_step[] = {  1, 0, 0, 0,
                            0, 1, 0, 0,
                            0, 0, 1, 0,
                            0, 0, 0, 1  };

uint8_t backward_step[] = { 0, 0, 0, 1,
                            0, 0, 1, 0,
                            0, 1, 0, 0,
                            1, 0, 0, 0  };
```

Figure 10.2.1 – Forward & Backward Stepper Sequence

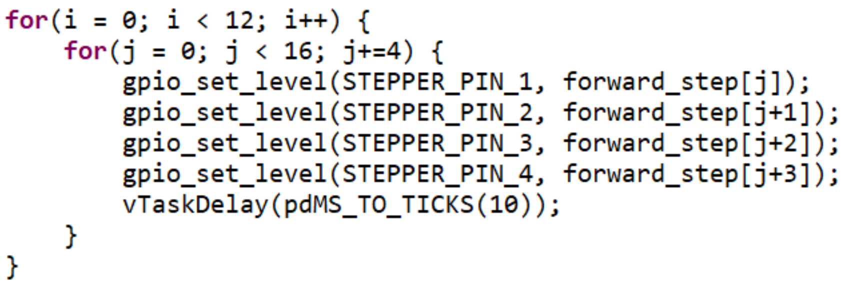
For example, if the stepper motor is told to pitch forward. It needs to loop through the forward_step array, assigning each of the stepper's four inputs (IN1 – IN4) to it's respective index in each sub-array.



```
{  1, 0, 0, 0,
   0, 1, 0, 0,
   0, 0, 1, 0,
   0, 0, 0, 1  };
```

Figure 10.2.2 – Sub-Arrays

After one iteration through the forward_step array, the stepper performs a single step. However, I noticed that a single step is extremely miniscule and hard to see. So I needed to create a nested for loop, that would perform an iteration through the array multiple times. I figured that twelve iterations produced a sufficient output:



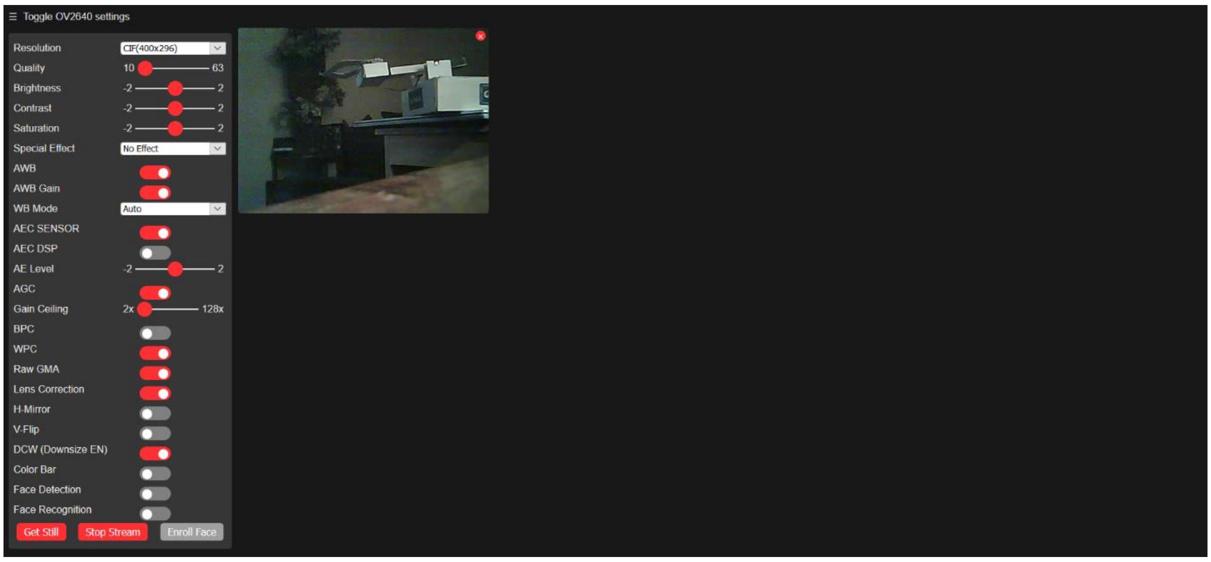
```
for(i = 0; i < 12; i++) {
    for(j = 0; j < 16; j+=4) {
        gpio_set_level(STEPPER_PIN_1, forward_step[j]);
        gpio_set_level(STEPPER_PIN_2, forward_step[j+1]);
        gpio_set_level(STEPPER_PIN_3, forward_step[j+2]);
        gpio_set_level(STEPPER_PIN_4, forward_step[j+3]);
        vTaskDelay(pdMS_TO_TICKS(10));
    }
}
```

Figure 10.2.3 – Nested For Loop

As can be seen in Figure 10.2.3, the inner for loop iterates through each sub-array, assigning each stepper input pin to its respective value. This for loop (without the outer loop) executes four times. The index variable, j, increments by four each time so that it begins at the first index of each sub-array, for each iteration. The outer loop then executes this whole sequence twelve times, with a very short delay of 10ms in between. This results in a significant pitch forward, performed very quickly.

10.3 ESP32 Camera

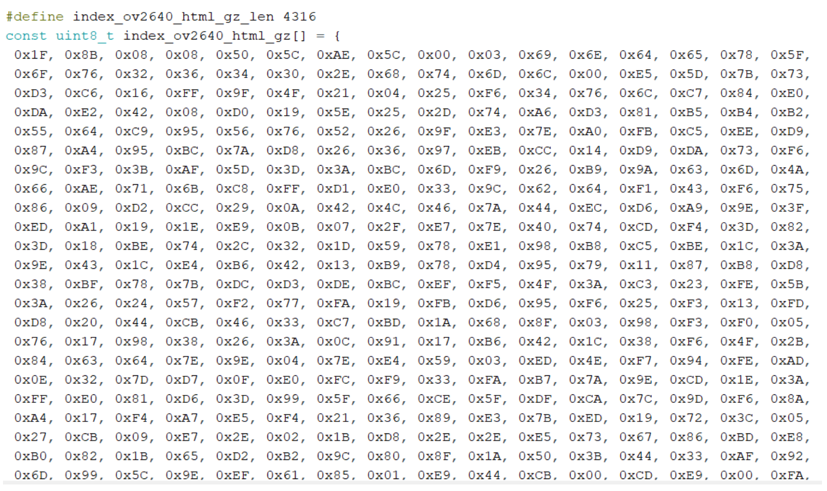
The ESP32 camera module has a lot of the same functionality as a regular ESP32 development board (built-in Wi-fi and Bluetooth, GPIO pins), with the added feature of the OV2640 camera. There is an example sketch in the ESP32 Arduino library that starts a HTTP web-server and displays a livestream. I can't take credit for creating this server and interfacing with the camera (I have referenced to it in the code), but I have stripped down the HTML so that I only have the elements I require for my project. As a quick reference, see Figure 10.3.1 for what the original web-page looks like.



Resolution	CIF(400x296)
Quality	10 — 63
Brightness	-2 — 2
Contrast	-2 — 2
Saturation	-2 — 2
Special Effect	No Effect
AWB	●
AWB Gain	●
WB Mode	Auto
AEC SENSOR	●
AEC DSP	○
AE Level	-2 — 2
AGC	●
Gain Ceiling	2x — 128x
BPC	○
WPC	●
Raw GMA	●
Lens Correction	●
H-Mirror	○
V-Flip	○
DCW (Downsize EN)	●
Color Bar	○
Face Detection	○
Face Recognition	○

Figure 10.3.1 – ESP32 Camera Web Server Original Web-Page

The original web page has many options for changing the configurations of the display, and also some face recognition capabilities. I didn't need all this functionality, so I edited the HTML code to just have a button to display the livestream. However, the HTML file is given as an array of hexadecimal values in a header file (See Figure 10.3.2). I needed to convert it back to a HTML format in order to edit it. For this, I used this web application called CyberChef used for code decoding and compression (https://gchq.github.io/CyberChef). Once I was happy with the HTML code, I needed to convert it back into an array of hex values, and place it into the camera_index.h file.



```
#define index_ov2640_html_gz_len 4316
const uint8_t index_ov2640_html_gz[] = {
 0x1F, 0x8B, 0x08, 0x08, 0x50, 0x5C, 0xAE, 0x5C, 0x00, 0x03, 0x69, 0x6E, 0x64, 0x65, 0x78, 0x5F,
 0x6F, 0x76, 0x32, 0x36, 0x34, 0x30, 0x2E, 0x68, 0x74, 0x6D, 0x6C, 0x00, 0xE5, 0x5D, 0x7B, 0x73,
 0xD3, 0xC6, 0x16, 0xFF, 0x9F, 0x4F, 0x21, 0x04, 0x25, 0xF6, 0x34, 0x76, 0x6C, 0xC7, 0x84, 0xE0,
 0xDA, 0xE2, 0x42, 0x08, 0xD0, 0x19, 0x5E, 0x25, 0x2D, 0x74, 0xA6, 0xD3, 0x81, 0xB5, 0xB4, 0xB2,
 0x55, 0x64, 0xC9, 0x95, 0x56, 0x76, 0x52, 0x26, 0x9F, 0xE3, 0x7E, 0xA0, 0xFB, 0xC5, 0xEE, 0xD9,
 0x87, 0xA4, 0x95, 0xBC, 0x7A, 0xD8, 0x26, 0x36, 0x97, 0xEB, 0xCC, 0x14, 0xD9, 0xDA, 0x73, 0xF6,
 0x9C, 0xF3, 0x3B, 0xAF, 0x5D, 0x3D, 0x3A, 0xBC, 0x6D, 0xF9, 0x26, 0xB9, 0x9A, 0x63, 0x6D, 0x4A,
 0x66, 0xAE, 0x71, 0x6B, 0xC8, 0xFF, 0xD1, 0xE0, 0x33, 0x9C, 0x62, 0x64, 0xF1, 0x43, 0xF6, 0x75,
 0x86, 0x09, 0xD2, 0xCC, 0x29, 0x0A, 0x42, 0x4C, 0x46, 0x7A, 0x44, 0xEC, 0xD6, 0xA9, 0x9E, 0x3F,
 0xED, 0xA1, 0x19, 0x1E, 0xE9, 0x0B, 0x07, 0x2F, 0xE7, 0x7E, 0x40, 0x74, 0xCD, 0xF4, 0x3D, 0x82,
 0x3D, 0x18, 0xBE, 0x74, 0x2C, 0x32, 0x1D, 0x59, 0x78, 0xE1, 0x98, 0xB8, 0xC5, 0xBE, 0x1C, 0x3A,
 0x9E, 0x43, 0x1C, 0xE4, 0xB6, 0x42, 0x13, 0xB9, 0x78, 0xD4, 0x95, 0x79, 0x11, 0x87, 0xB8, 0xD8,
 0x38, 0xBF, 0x78, 0x7B, 0xDC, 0xD3, 0xDE, 0xBC, 0xEF, 0xF5, 0x4F, 0x3A, 0xC3, 0x23, 0xFE, 0x5B,
 0x3A, 0x26, 0x24, 0x57, 0xF2, 0x77, 0xFA, 0x19, 0xFB, 0xD6, 0x95, 0xF6, 0x25, 0xF3, 0x13, 0xFD,
 0xD8, 0x20, 0x44, 0xCB, 0x46, 0x33, 0xC7, 0xBD, 0x1A, 0x68, 0x8F, 0x03, 0x98, 0xF3, 0xF0, 0x05,
 0x76, 0x17, 0x98, 0x38, 0x26, 0x3A, 0x0C, 0x91, 0x17, 0xB6, 0x42, 0x1C, 0x38, 0xF6, 0x4F, 0x2B,
 0x84, 0x63, 0x64, 0x7E, 0x9E, 0x04, 0x7E, 0xE4, 0x59, 0x03, 0xED, 0x4E, 0xF7, 0x94, 0xFE, 0xAD,
 0x0E, 0x32, 0x7D, 0xD7, 0x0F, 0xE0, 0xFC, 0xF9, 0x33, 0xFA, 0xB7, 0x7A, 0x9E, 0xCD, 0x1E, 0x3A,
 0xFF, 0xE0, 0x81, 0xD6, 0x3D, 0x99, 0x5F, 0x66, 0xCE, 0x5F, 0xDF, 0xCA, 0x7C, 0x9D, 0xF6, 0x8A,
 0xA4, 0x17, 0xF4, 0xA7, 0xE5, 0xF4, 0x21, 0x36, 0x89, 0xE3, 0x7B, 0xED, 0x19, 0x72, 0x3C, 0x05,
 0x27, 0xCB, 0x09, 0xE7, 0x2E, 0x02, 0x1B, 0xD8, 0x2E, 0x2E, 0xE5, 0x73, 0x67, 0x86, 0xBD, 0xE8,
 0xB0, 0x82, 0x1B, 0x65, 0xD2, 0xB2, 0x9C, 0x80, 0x8F, 0x1A, 0x50, 0x3B, 0x44, 0x33, 0xAF, 0x92,
 0x6D. 0x99. 0x5C. 0x9E. 0xEF. 0x61. 0x85. 0x01. 0xE9. 0x44. 0xCB. 0x00. 0xCD. 0xE9. 0x00. 0xFA.
```

Figure 10.3.2 – **camera_index.h** - Original HTML as an array of hexadecimal values

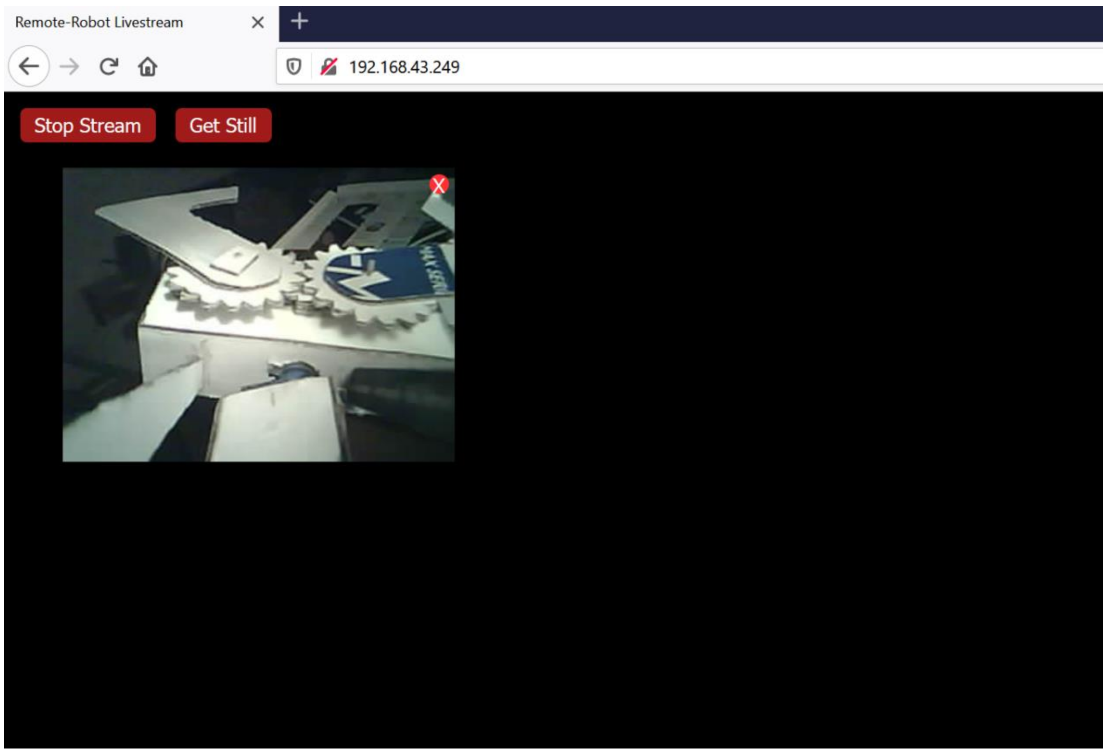


Figure 10.3.3 – The Edited Web-Page

The ESP32 camera module does three things:

1. Starts a HTTP webserver that displays a livestream.
2. Receives sensor data from the glove module through ESP-NOW.
3. Publishes the sensor data to the AWS IoT Core MQTT Broker.

To subscribe and publish the sensor data up to AWS, I am using the hornbill AWS IoT library [7] in Arduino. First of all, I need to specify the address of my AWS IoT host. The client ID is arbitrary, and I then must specify the topic I want to subscribe to.

```
char ssid[] = "OnePlus3";
char password[] = "12345678";
char AWS_HOST[] = "a2ot8lh5ttyzzl-ats.iot.us-east-1.amazonaws.com";
char CLIENT_ID[] = "espClient";
char TOPIC[] = "myESP32/esp32topic";
```

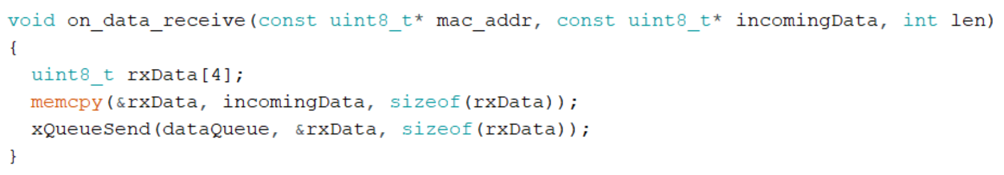
Figure 10.3.4 – AWS IoT Configuration

```
void topic_subscribe()
{
  if(hornbill.connect(AWS_HOST, CLIENT_ID) == 0)
  {
    Serial.println("Connected to AWS");
    delay(1000);

    if(0==hornbill.subscribe(TOPIC, sub_callback_handler))
    {
      Serial.println("Subscribe Successful");
      init_esp_now();
    }
    else
    {
      Serial.println("Subscribe Failed - check Thing Name and certs");
      while(1);
    }
  }
  else
  {
    Serial.println("AWS connection failed - check HOST address");
    while(1);
  }
}
```

Figure 10.3.5 – Function To Subscribe To My AWS MQTT Topic

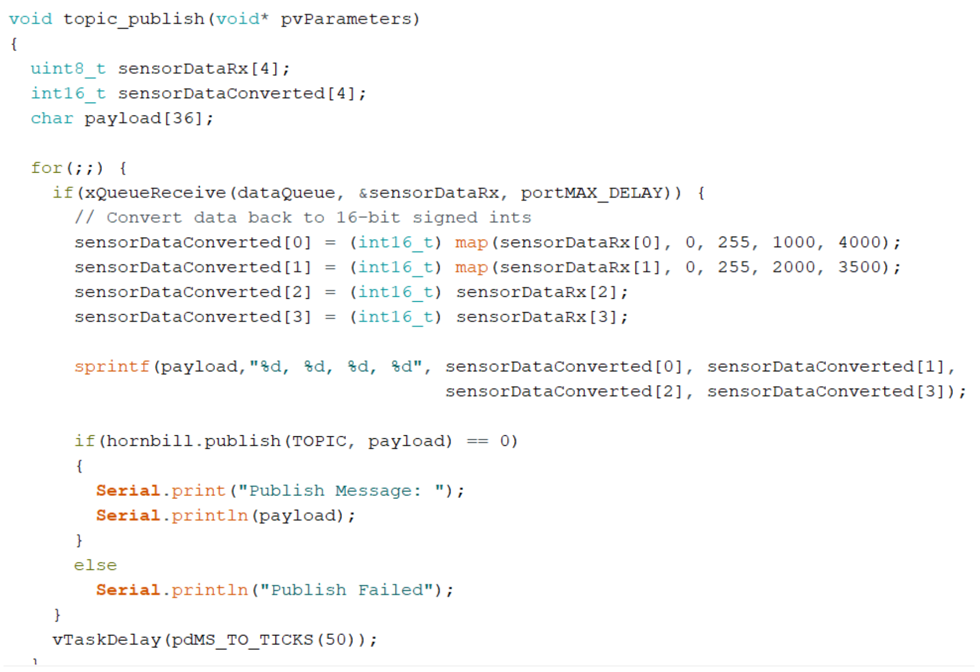
```
void init_esp_now()
{
  ESP_ERROR_CHECK(esp_now_init());
  esp_now_peer_info_t glove_peerInfo;
  memcpy(glove_peerInfo.peer_addr, gloveModuleAddress, 6);
  glove_peerInfo.ifidx = WIFI_IF_STA;
  glove_peerInfo.channel = 1;
  glove_peerInfo.encrypt = false;
  ESP_ERROR_CHECK(esp_now_add_peer(&glove_peerInfo));
  esp_now_register_recv_cb(on_data_receive);

  delay(1000);
}
```

Figure 10.3.6 – Function To Initialize ESP-NOW & Configure Peer



```
void on_data_receive(const uint8_t* mac_addr, const uint8_t* incomingData, int len)
{
  uint8_t rxData[4];
  memcpy(&rxData, incomingData, sizeof(rxData));
  xQueueSend(dataQueue, &rxData, sizeof(rxData));
}
```

Figure 10.3.7 – ESP-NOW Receiving Call-Back Function



```
void topic_publish(void* pvParameters)
{
  uint8_t sensorDataRx[4];
  int16_t sensorDataConverted[4];
  char payload[36];

  for(;;) {
    if(xQueueReceive(dataQueue, &sensorDataRx, portMAX_DELAY)) {
      // Convert data back to 16-bit signed ints
      sensorDataConverted[0] = (int16_t) map(sensorDataRx[0], 0, 255, 1000, 4000);
      sensorDataConverted[1] = (int16_t) map(sensorDataRx[1], 0, 255, 2000, 3500);
      sensorDataConverted[2] = (int16_t) sensorDataRx[2];
      sensorDataConverted[3] = (int16_t) sensorDataRx[3];

      sprintf(payload,"%d, %d, %d, %d", sensorDataConverted[0], sensorDataConverted[1],
                                        sensorDataConverted[2], sensorDataConverted[3]);

      if(hornbill.publish(TOPIC, payload) == 0)
      {
        Serial.print("Publish Message: ");
        Serial.println(payload);
      }
      else
        Serial.println("Publish Failed");
    }
    vTaskDelay(pdMS_TO_TICKS(50));
```

Figure 10.3.8 – Function To Publish Sensor Readings To AWS MQTT Topic

As the ESP32 camera module doesn't have a micro USB port, I needed to use a FTDI programmer in order to flash my code onto the board:

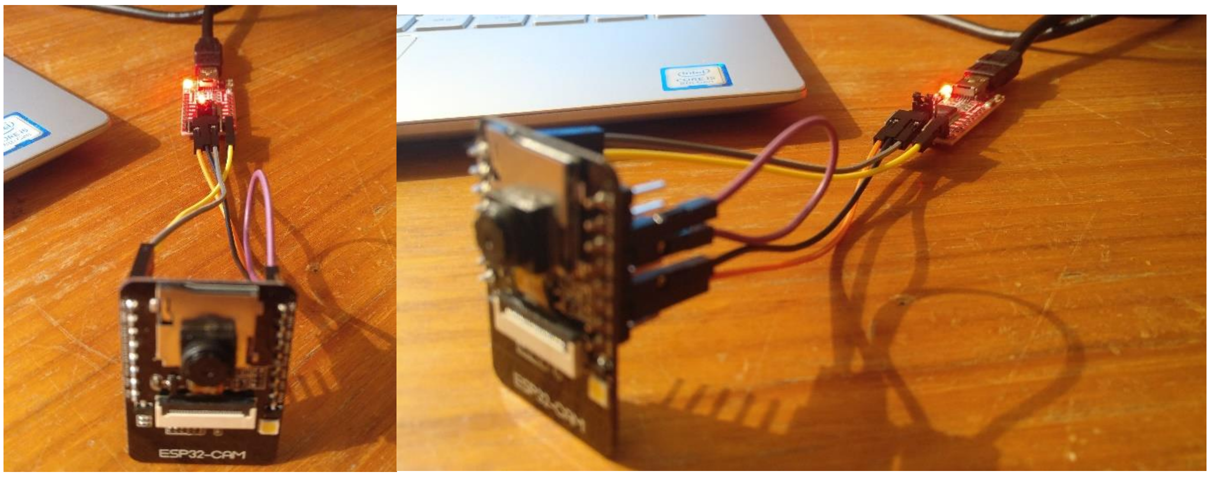
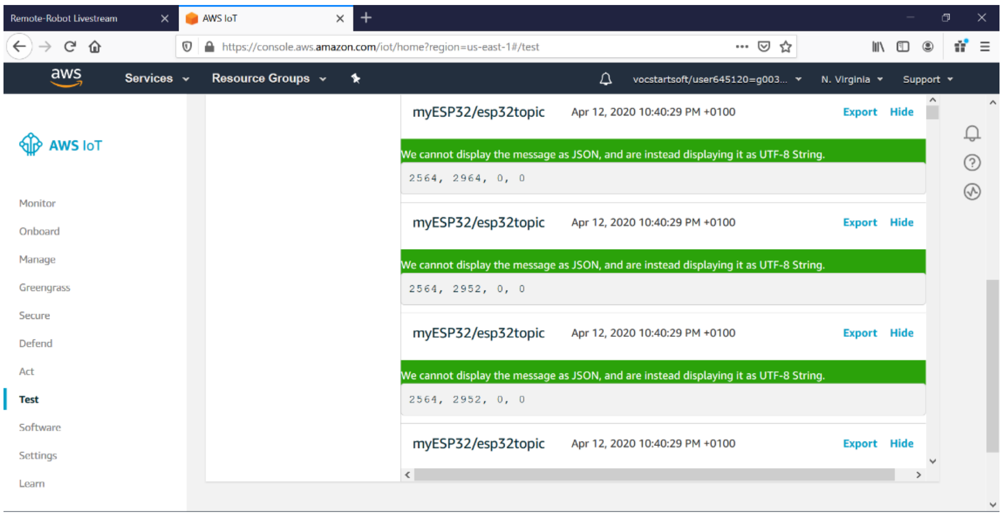


Figure 10.3.9 – FTDI Programmer

11. AWS IoT Core

I am using Amazon Web Service’s Internet of Things Core in order to route the sensor readings to my Node.js server. The ESP32 Camera module publishes the array of data as a string to AWS using the MQTT protocol. MQTT is a lightweight messaging protocol that uses the idea of publishing and subscribing to a particular broker. In my case, the broker is set up on the AWS IoT Core. The ESP32 camera module publishes the data. My Node.js server then subscribes to that particular topic on the broker, in order to retrieve the sensor data. From the AWS IoT console, I can subscribe to my topic (myESP32/esp32topic), and see the packet of data being received.



myESP32/esp32topic Apr 12, 2020 10:40:29 PM +0100

We cannot display the message as JSON, and are instead displaying it as UTF-8 String.

```
2564, 2964, 0, 0
```

myESP32/esp32topic Apr 12, 2020 10:40:29 PM +0100

We cannot display the message as JSON, and are instead displaying it as UTF-8 String.

```
2564, 2952, 0, 0
```

myESP32/esp32topic Apr 12, 2020 10:40:29 PM +0100

We cannot display the message as JSON, and are instead displaying it as UTF-8 String.

```
2564, 2952, 0, 0
```

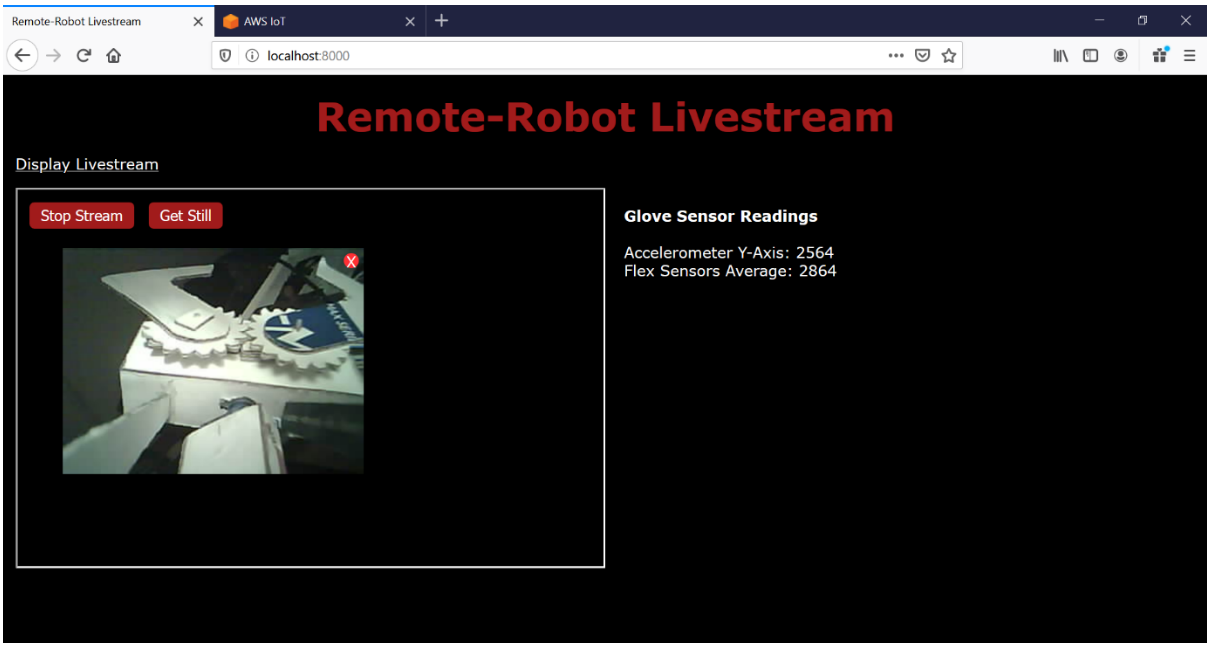
myESP32/esp32topic Apr 12, 2020 10:40:29 PM +0100

Figure 11.1 – AWS IoT Core MQTT Client

12. Node.js Express Server

A Node.js server runs locally on my computer. It displays a web-page with a livestream from the robot arm, plus some real-time sensor data. The livestream is displaying on a separate HTTP server running on the ESP32 camera module. Using a HTML iFrame, I can essentially embed the livestream web-page within my main web-page. If "Display Livestream" is clicked, it redirects to the livestream server (192.168.43.249), and displays it within the iFrame.

The node.js server subscribes to my MQTT topic on the AWS IoT Core, in order to pull the sensor data. The data is received as a string, so I need to parse each value from the string. Using AJAX (Asynchronous JavaScript And Xml), I can dynamically update the sensor readings, without the need to refresh the page. It uses a XMLHttpRequest to request the data. Inside my JavaScript code, I have it continuously sending these requests and updating the respective HTML element, with the new sensor value.



Glove Sensor Readings

Accelerometer Y-Axis: 2564
Flex Sensors Average: 2864

Figure 12.1 – Node.js Web-Page

13. Wireless Communication

13.1 ESP-NOW

In order to meet the requirement of allowing the robot arm to be controlled remotely, I needed to find a means of wireless communication between the glove and robot module. I discovered that Espressif had developed their own wireless protocol called ESP-NOW. ESP-NOW is a peer-to-peer wireless protocol, without the need for Wi-fi. It allows up to 250 Bytes of data to be sent at a time. It would be similar to the 2.4GHz wireless communication you would get with a wireless mouse or TV remote [8]. It is a very simple and lightweight means of communication between ESP devices. All that is required is knowledge of the MAC address of the device you want to talk to. Once a device is configured as a peer, you are free to send and receive data. The default bit rate of ESP-NOW is 1Mbps. It is a reliable and fast protocol, without the trouble of connecting to access points.

If the device is the transmitter, it must specify a sending call-back function that executes every time data is sent. The receiving device must specify a receiver call-back function every time data is retrieved.

The packet of data being sent from the glove to the robot is an array consisting of four unsigned 8 bit integers. These four values are the Y-axis value of the accelerometer, the average of the three flex sensors, the state of the forward pitch button, and the state of the backward pitch button. I need to convert the accelerometer and flex sensor average to unsigned 8 bit integers before transmitting, as it is a requirement when using ESP-NOW.

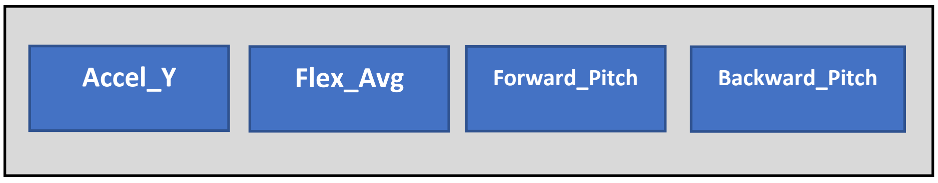


Figure 13.1.1 – My ESP-NOW Data Packet

13.2 Range

I tested the range of communication between the glove and robot arm, and found that the robot can be controlled from at least 215 metres away. It could actually be remotely controlled at a further distance, once there is a minimal amount of obstacles in between both modules.

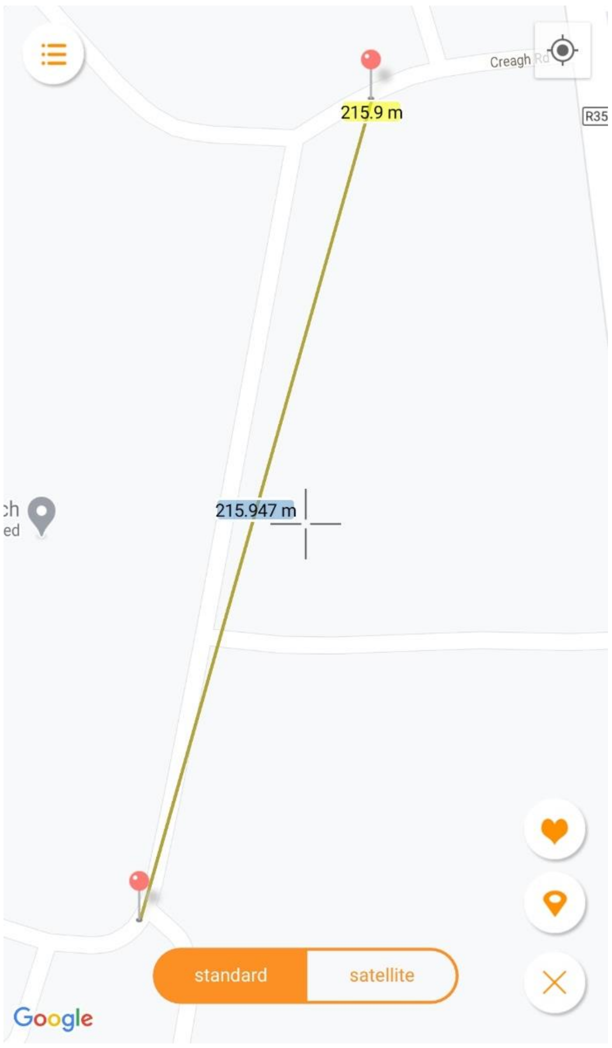
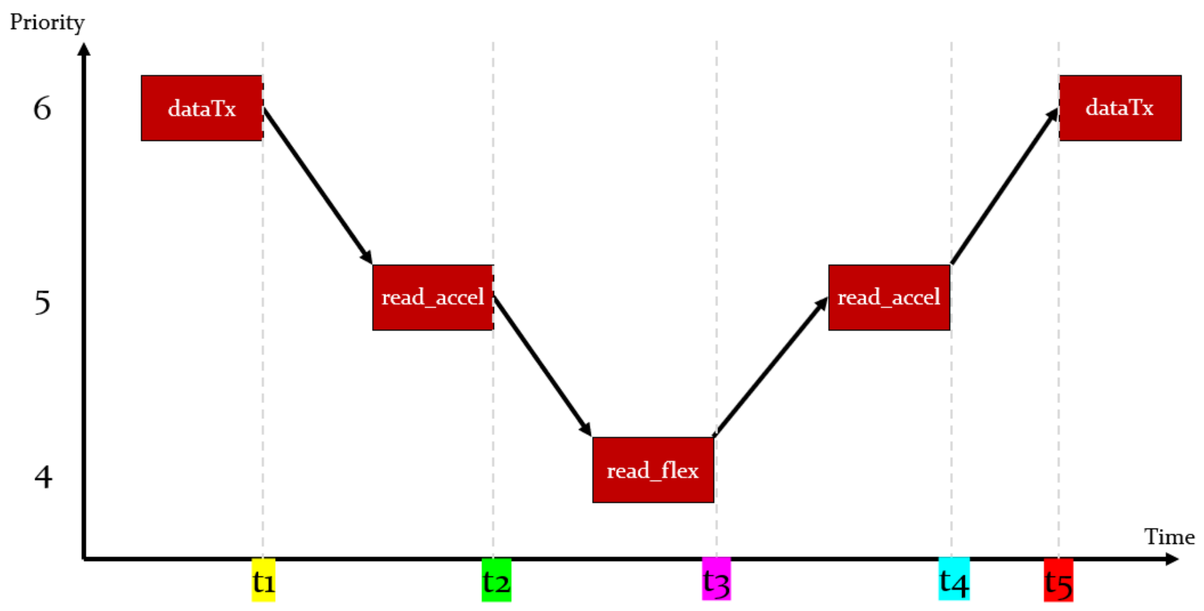


Figure 13.2.1 – Range Of Communication Between Glove and Robot Arm

14. FreeRTOS Task Synchronization

The ESP-IDF implements a version of Amazon's FreeRTOS, which I would have some experience using in first semester. With FreeRTOS, you can achieve real-time task synchronization using communication objects such as queues, task notifications, event groups and semaphores. I have my programs configured for pre-emptive scheduling. Pre-emptive scheduling is where the execution of the program is decided by the priority of tasks. If a higher priority task is ready to run, it will pre-empt any lower priority tasks. This allows faster response times for tasks that are of higher importance.

14.1 Glove Module Task Timing Diagram



t1 – **dataTx** begins executing but is blocked waiting for an incoming queue containing the sensor data from the accelerometer task.

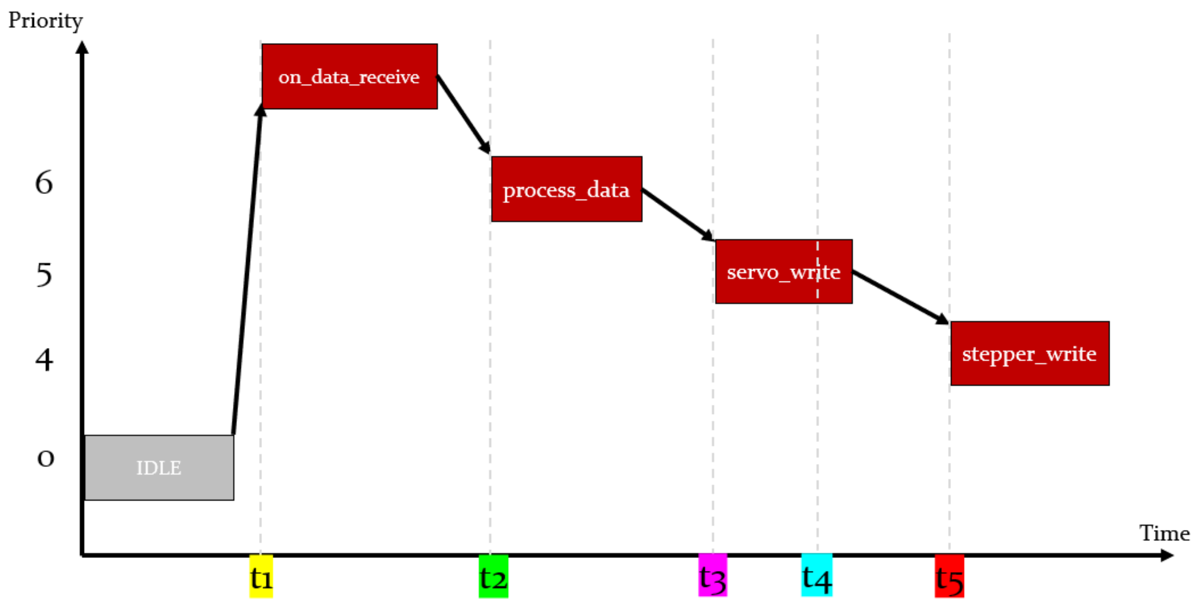
t2 – **read_accel** is the next highest priority task, so it begins execution. It reads the y-axis value from the accelerometer and then blocks waiting for a task notification from the flex sensor task.

t3 – **read_flex** retrieves the average value of the three flex sensors and sends this value as a task notification back to the accelerometer task.

t4 – **read_accel** is now unblocked as it has received the notification from **read_flex**. The sensor data is then mapped down to the range 0-255 using the map() function, and then sent in a queue up to **dataTx**.

t5 – **dataTx** is then unblocked after receiving the packet of sensor data. It can then send the data as an array of unsigned 8-bit integers to the robot and camera modules using ESP-NOW.

14.2 Robot Module Task Timing Diagram



t1 – **IDLE** task is running as all tasks are in a blocked state. At t1, the glove module is powered on and begins transmitting data to the robot module. **on_data_receive** is a call-back function that runs when data is received through ESP-NOW. It defers the processing of the data to another task by sending the it in a queue. This is to make sure the execution time of the call-back function is kept as short as possible.

t2 – **process_data** unblocks after receiving the queue. It converts the data back into 16-bit signed integers and sends a queue to the servo task.

t3 – servo_write uses Pulse Width Modulation to update the duty cycle of the signal being sent to both of the servo motors. It also checks if the state of the forward/backward push-button has changed to 1.

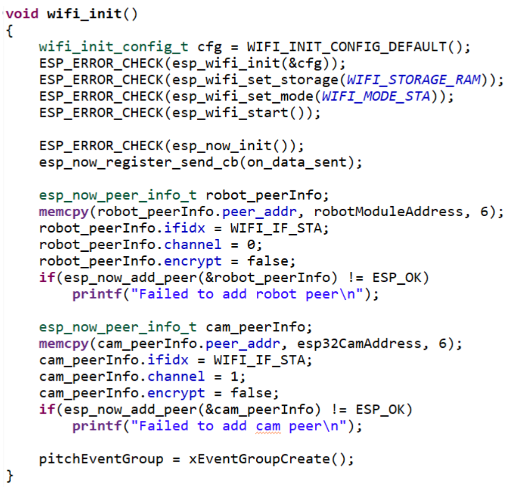
t4 – The forward pitch button is pressed on the glove module. This causes an interrupt on the glove's side, and sends the state of the button to the robot. In **servo_write**, it notices that the forward_state variable has changed from 0 to 1. This causes a flag in an event group to be set.

t5 – **stepper_write** unblocks when it recognises that the pitch forward flag in the event group has been set. This then performs the operation to pitch the robot arm forward.

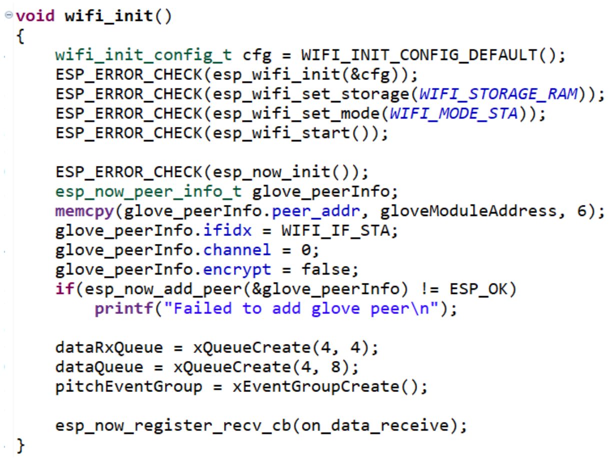
14.3 Other Functions

I will breakdown the other functions/handlers (not FreeRTOS tasks) that appear in my code.

wifi_init() – This function is common to both the robot and glove module, and is the first function that executes. It initializes Wi-Fi and ESP-NOW. It then configures the other devices as peers, to allow for ESP-NOW communication.

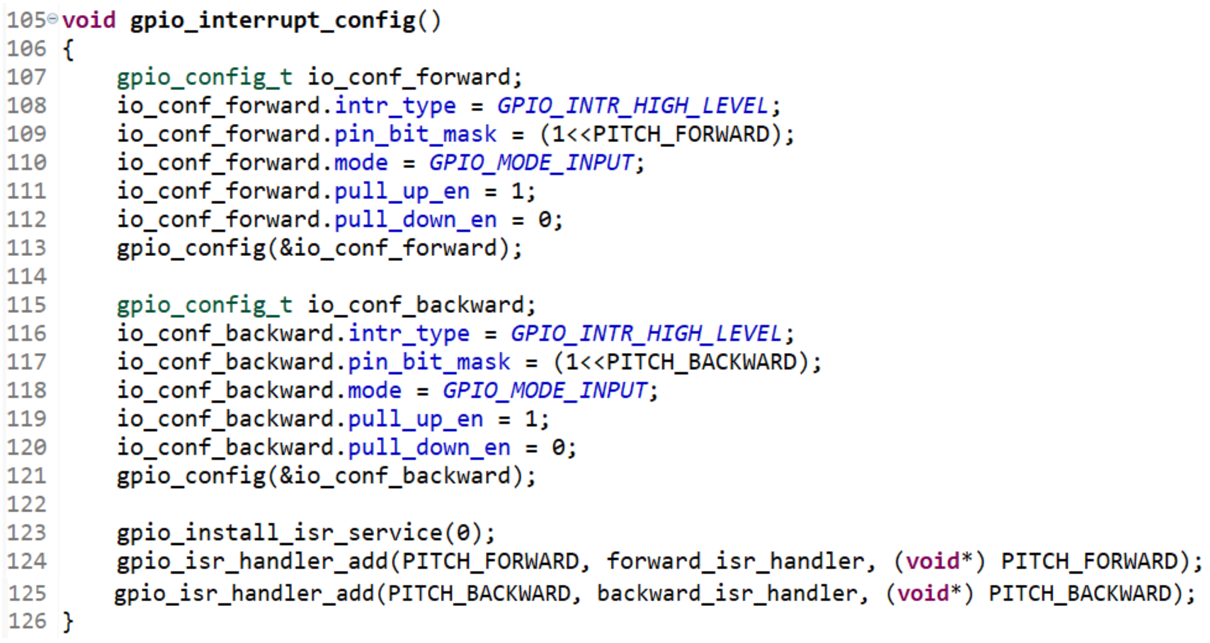


```
void wifi_init()
{
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));
    ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_RAM));
    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
    ESP_ERROR_CHECK(esp_wifi_start());

    ESP_ERROR_CHECK(esp_now_init());
    esp_now_register_send_cb(on_data_sent);

    esp_now_peer_info_t robot_peerInfo;
    memcpy(robot_peerInfo.peer_addr, robotModuleAddress, 6);
    robot_peerInfo.ifidx = WIFI_IF_STA;
    robot_peerInfo.channel = 0;
    robot_peerInfo.encrypt = false;
    if(esp_now_add_peer(&robot_peerInfo) != ESP_OK)
        printf("Failed to add robot peer\n");

    esp_now_peer_info_t cam_peerInfo;
    memcpy(cam_peerInfo.peer_addr, esp32CamAddress, 6);
    cam_peerInfo.ifidx = WIFI_IF_STA;
    cam_peerInfo.channel = 1;
    cam_peerInfo.encrypt = false;
    if(esp_now_add_peer(&cam_peerInfo) != ESP_OK)
        printf("Failed to add cam peer\n");

    pitchEventGroup = xEventGroupCreate();
}
```



```
void wifi_init()
{
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));
    ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_RAM));
    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
    ESP_ERROR_CHECK(esp_wifi_start());

    ESP_ERROR_CHECK(esp_now_init());
    esp_now_peer_info_t glove_peerInfo;
    memcpy(glove_peerInfo.peer_addr, gloveModuleAddress, 6);
    glove_peerInfo.ifidx = WIFI_IF_STA;
    glove_peerInfo.channel = 0;
    glove_peerInfo.encrypt = false;
    if(esp_now_add_peer(&glove_peerInfo) != ESP_OK)
        printf("Failed to add glove peer\n");

    dataRxQueue = xQueueCreate(4, 4);
    dataQueue = xQueueCreate(4, 8);
    pitchEventGroup = xEventGroupCreate();

    esp_now_register_recv_cb(on_data_receive);
}
```

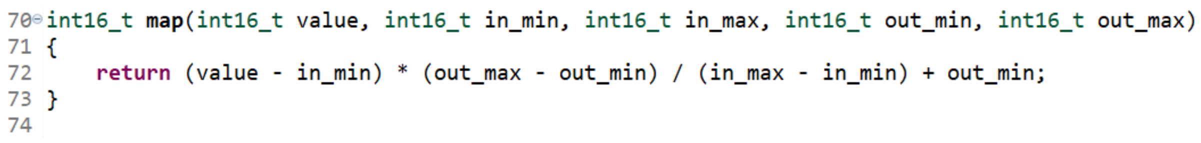
gpio_interrupt_config() – (Exclusive to Robot Module). Configures the two push-buttons as external interrupts (GPIO4: Forward, GPIO5: Backward). They are set-up to occur on a rising-edge as these buttons are active-high. It also configures the interrupt service routine handler for each button.



```
void gpio_interrupt_config()
{
    gpio_config_t io_conf_forward;
    io_conf_forward.intr_type = GPIO_INTR_HIGH_LEVEL;
    io_conf_forward.pin_bit_mask = (1<<PITCH_FORWARD);
    io_conf_forward.mode = GPIO_MODE_INPUT;
    io_conf_forward.pull_up_en = 1;
    io_conf_forward.pull_down_en = 0;
    gpio_config(&io_conf_forward);

    gpio_config_t io_conf_backward;
    io_conf_backward.intr_type = GPIO_INTR_HIGH_LEVEL;
    io_conf_backward.pin_bit_mask = (1<<PITCH_BACKWARD);
    io_conf_backward.mode = GPIO_MODE_INPUT;
    io_conf_backward.pull_up_en = 1;
    io_conf_backward.pull_down_en = 0;
    gpio_config(&io_conf_backward);

    gpio_install_isr_service(0);
    gpio_isr_handler_add(PITCH_FORWARD, forward_isr_handler, (void*) PITCH_FORWARD);
    gpio_isr_handler_add(PITCH_BACKWARD, backward_isr_handler, (void*) PITCH_BACKWARD);
}
```

map() – (Also used on both modules). This algorithm, provided by Arduino [9], allows me to map a variable to a more desirable range. This helps for mapping my 16-bit signed integer values (y-axis accelerometer and flex sensor average) down to the range of an unsigned 8-

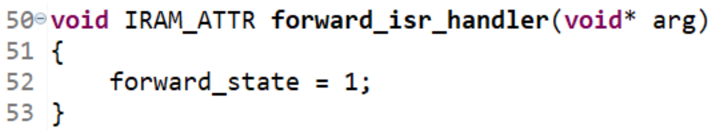
bit integer (0-255). I need to do this before transmitting the data using ESP-NOW. The first parameter it takes is the variable you want to map. The second and third are the minimum and maximum range of that variable. The last two parameters are the minimum and maximum range that you want to map to.



```
int16_t map(int16_t value, int16_t in_min, int16_t in_max, int16_t out_min, int16_t out_max)
{
    return (value - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}

```

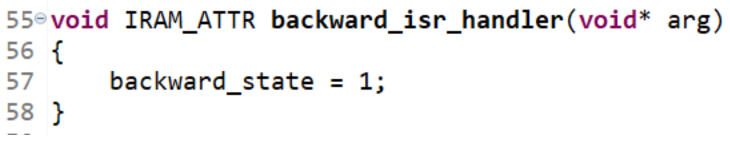
The glove module has two interrupt handlers:

forward_isr_handler() –Interrupt handler for the pitch forward button (GPIO4). When an interrupt occurs, it changes the global variable, forward_state, to 1.



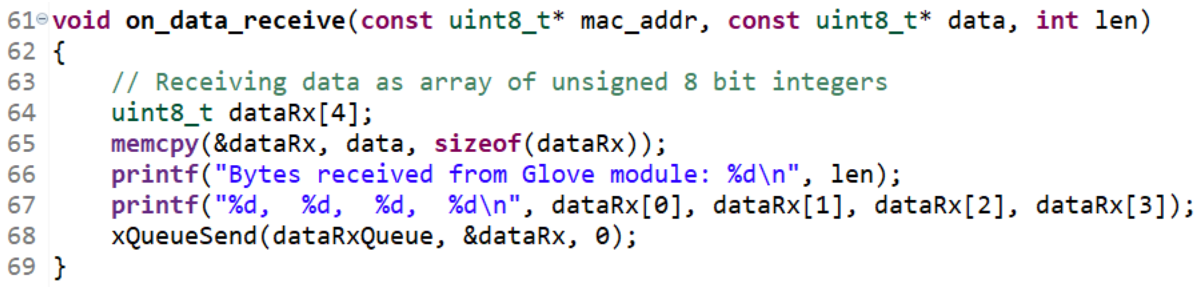
```
void IRAM_ATTR forward_isr_handler(void* arg)
{
    forward_state = 1;
}
```

backward_isr_handler() - Interrupt handler for the pitch backward button (GPIO5). When an interrupt occurs, it changes the global variable, backward_state, to 1.



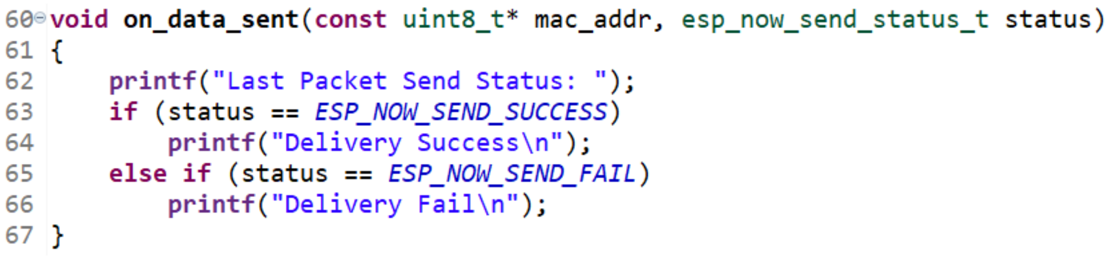
```
void IRAM_ATTR backward_isr_handler(void* arg)
{
    backward_state = 1;
}
```

on_data_receive() – (Robot module). ESP-NOW call-back function that executes whenever data is received. It uses a queue to send the received data to another task for processing. This is to keep the call-back function as short as possible, as it interrupts the rest of the code.



```
void on_data_receive(const uint8_t* mac_addr, const uint8_t* data, int len)
{
    // Receiving data as array of unsigned 8 bit integers
    uint8_t dataRx[4];
    memcpy(&dataRx, data, sizeof(dataRx));
    printf("Bytes received from Glove module: %d\n", len);
    printf("%d,  %d,  %d,  %d\n", dataRx[0], dataRx[1], dataRx[2], dataRx[3]);
    xQueueSend(dataRxQueue, &dataRx, 0);
}
```

on_data_sent() – (Glove Module). ESP-NOW call-back function that executes whenever data is sent. It displays whether or not the data was transmitted successfully.



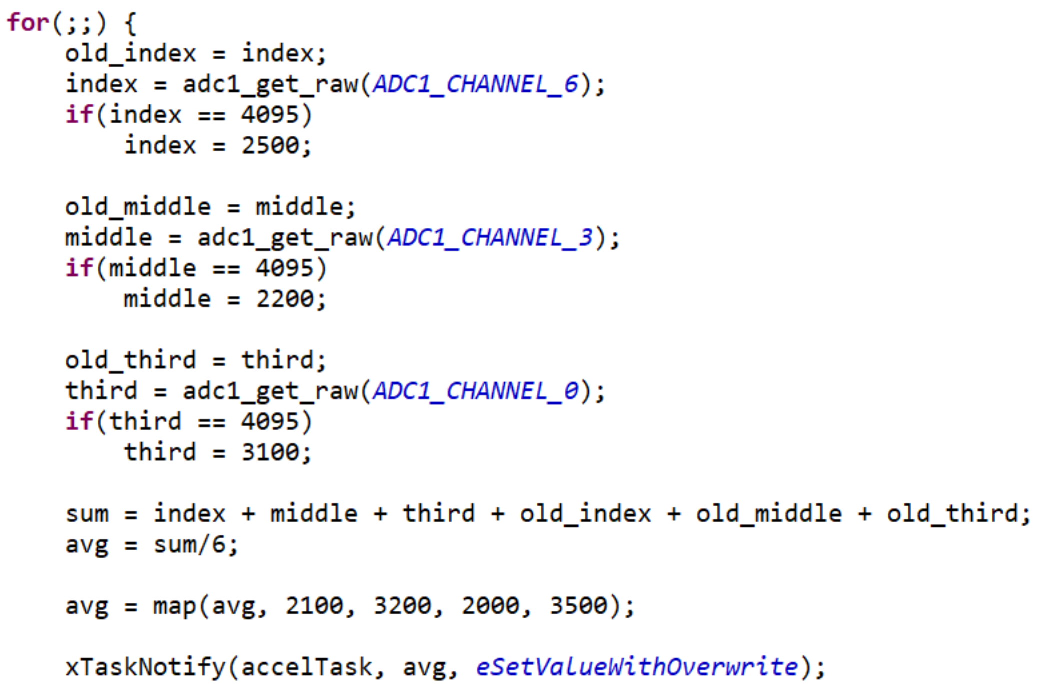
```
void on_data_sent(const uint8_t* mac_addr, esp_now_send_status_t status)
{
    printf("Last Packet Send Status: ");
    if (status == ESP_NOW_SEND_SUCCESS)
        printf("Delivery Success\n");
    else if (status == ESP_NOW_SEND_FAIL)
        printf("Delivery Fail\n");
}
```

15. Performance Improvements

15.1 Flex Sensor Smoothing Average

A problem I’ve had throughout this whole project, was that I was getting an inconsistent output from my flex sensors. This was to be expected as I am reading the average value of three sensors, rather than one, meaning it can be more prone to a sporadic output. Another reason why this was happening is because the hardware connections attached to the sensors are not the best, so they can momentarily disconnect causing spikes/drops in the output. I couldn’t really do much about the hardware connections as they were glued to the glove, and I didn’t want to have to get a new glove with new flex sensors and connections. So I had to come up with a software solution to the problem. The first thing I did was isolate each sensor and test it to see if it was producing sporadic readings. I noticed each sensor had its own unique defect – for example, the sensor on the index finger sensor produced a value of 4095 (maximum value) when the finger is in the resting position (it should be at its minimum value). So I needed to add an if statement to check if it is at 4095, then change it back to 2500 (minimum value). This helped, but there was still room for improvement.

I researched the idea of a Moving Average Filter (MAF) which essentially regulates a set of data. This gave me the idea to store the last reading of each sensor, along with its current reading. This meant I now had six values (last reading and current reading for the three flex sensors). I would then calculate the average of these six values, resulting in a much smoother output on the gripper servo motor.



```
for(;;) {
    old_index = index;
    index = adc1_get_raw(ADC1_CHANNEL_6);
    if(index == 4095)
        index = 2500;

    old_middle = middle;
    middle = adc1_get_raw(ADC1_CHANNEL_3);
    if(middle == 4095)
        middle = 2200;

    old_third = third;
    third = adc1_get_raw(ADC1_CHANNEL_0);
    if(third == 4095)
        third = 3100;

    sum = index + middle + third + old_index + old_middle + old_third;
    avg = sum/6;

    avg = map(avg, 2100, 3200, 2000, 3500);

    xTaskNotify(accelTask, avg, eSetValueWithOverwrite);
```

Figure 15.1.1 – Flex Sensor Smoothing Average

15.2 Dual Core Utilization

As the ESP32 boasts two cores (APP CPU and PRO CPU), I felt it would be a good idea to take advantage of this. With FreeRTOS, you can pin certain tasks to a particular core. The last parameter in xTaskCreatePinnedToCore is for specifying the core you wish to pin that task to (Core 0 – PRO CPU, Core 1 – APP CPU):

```
xTaskCreatePinnedToCore(&data_Tx, "data_Tx", 2048, NULL, 6, NULL, 0);
xTaskCreatePinnedToCore(&read_accelerometer, "read_accelerometer", 2048, NULL, 5, &accelTask, 1);
xTaskCreatePinnedToCore(&read_flex, "read_flex", 2048, NULL, 4, &flexTask, 1);
```

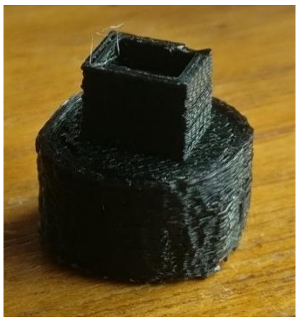
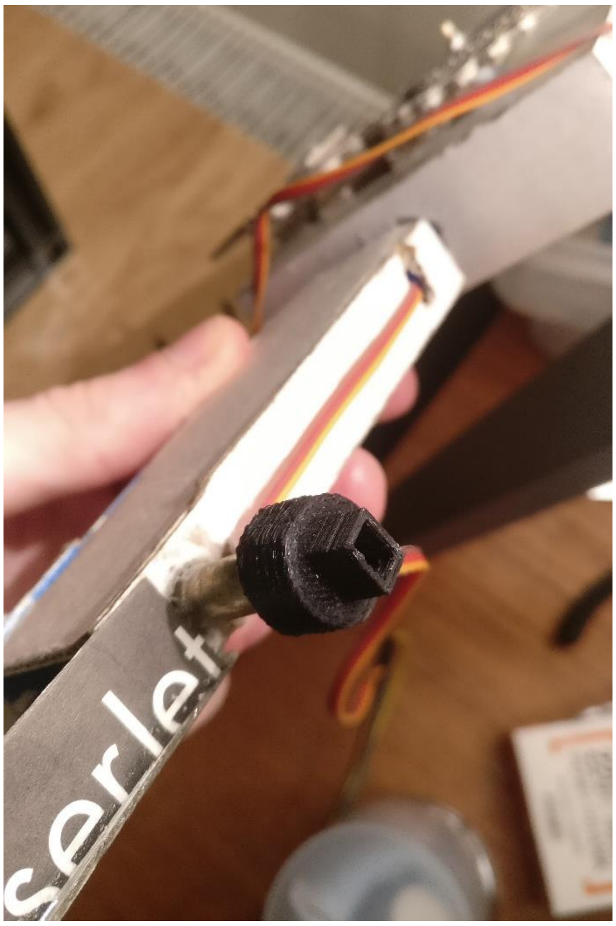
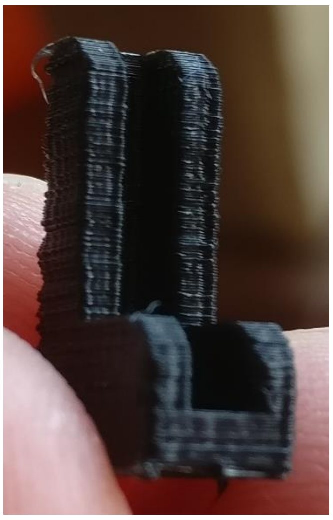
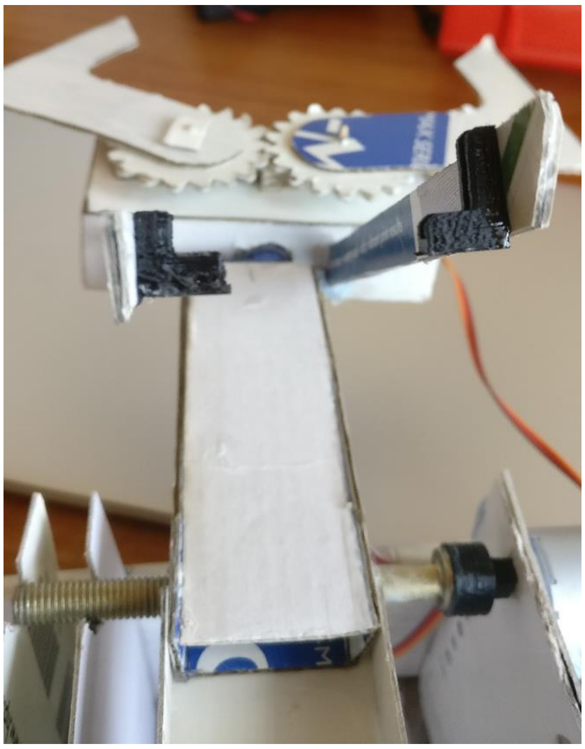
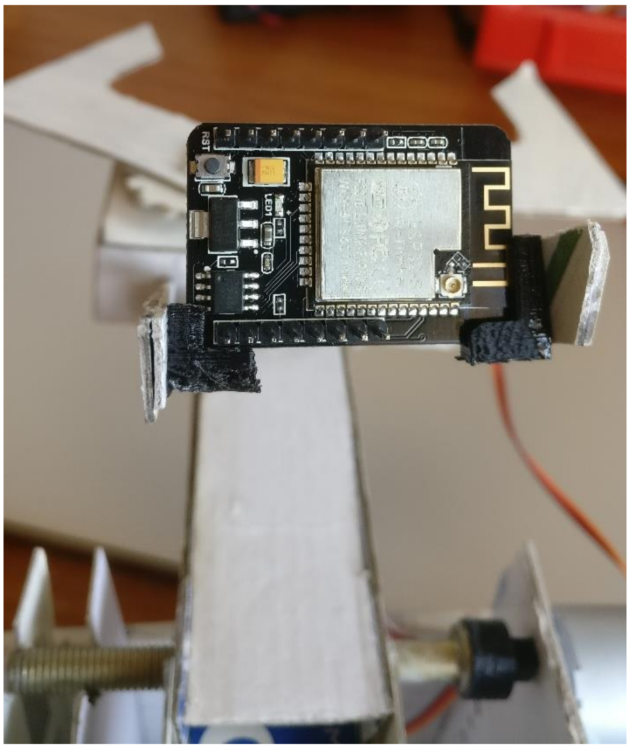
Figure 15.2.1 – Glove Module Tasks

```
xTaskCreatePinnedToCore(&servo_write, "servo_write", 2048, NULL, 5, NULL, 1);
xTaskCreatePinnedToCore(&stepper_write, "stepper_write", 2048, NULL, 4, NULL, 1);
xTaskCreatePinnedToCore(&process_data, "process_data", 2048, NULL, 6, NULL, 0);
```

Figure 15.2.1 – Robot Module Tasks

This allows tasks to execute concurrently, and as a result optimize the overall performance.

16. 3-D Printed Parts

17. Conclusion

Overall, I am happy with the result of my project. It allowed me to further my knowledge of embedded programming (especially working with FreeRTOS), which is the area I would like to get into after college. Working with AWS was also a big bonus. I also got to improve my skills with testing and designing circuits.

I had hoped to eventually have a fully 3-D printed robot arm. I received a faulty cable with my 3-D printer which meant that I essentially had no extruder motor. So, if I wanted to print something, it had to be a very small piece because I need to manually feed the filament into the printer. Due to COVID-19, this cable never arrived. However, I felt the prototype I had built from first semester was good enough to demonstrate the concept of the Remote-Robot. If I was to do this project again, I would have made the robot arm mobile, and have it fitted with a GPS tracker.

A big thing this project thought me was organization. I followed what you would see in an Agile environment, by setting one or two week deadlines to complete certain features. This allowed me extra time at the end of the project for any additional features, and for project deliverables.

18. References

[1] – R. Santos. (December, 2016). "Getting started with the ESP32 Development Board". Random Nerd Tutorials. [Online]. Available: https://randomnerdtutorials.com/getting-started-with-esp32

[2] - Espressif Systems (Shanghai) CO. LTD. "ESP-IDF Programming Guide". docs.espressif.com. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started

[3] – A. Sanjeev. (2018). "How to Interface Arduino and the MPU 6050 Sensor". Maker Pro. [Online]. Available: https://maker.pro/arduino/tutorial/how-to-interface-arduino-and-the-mpu-6050-sensor

[4] – N. Kolban. (February, 2017). "ESP32 Technical Tutorials: MPU6050 Accelerometer". YouTube. [Online]. Available: https://www.youtube.com/watch?v=HwurH20jsvw&list=PLB-czhEQLJbWMOl7Ew4QW1LpoUUE7QMOo&index=15&t=368s

[5] – InvenSense Inc. (August, 2013). "MPU-6000 and MPU-6050 Register Map and Descriptions". Revision 4.2. invensense.tdk.com. [Online]. Available: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf

[6] – N. Kolban. (January, 2017). "ESP32 Technical Tutorials: ESP32 and Pulse Width Modulation". YouTube. [Online]. Available: https://www.youtube.com/watch?v=rarE-WI_Y0A&list=PLB-czhEQLJbWMOl7Ew4QW1LpoUUE7QMOo&index=24

[7] – S. Bagwan. (Explore Embedded). (July, 2017). "Arduino-ESP32-AWS-IoT". GitHub. [Online]. Available: https://github.com/ExploreEmbedded/Hornbill-Examples/tree/master/arduino-esp32/AWS_IOT

[8] – Espressif Systems (Shanghai) CO. LTD. "ESP-NOW Overview". espressif.com. [Online]. Available: https://www.espressif.com/en/products/software/esp-now/overview

[9] – Arduino. "map() function". arduino.cc. [Online]. Available: https://www.arduino.cc/reference/en/language/functions/math/map/