MIE438 Project Report

Group 9:
Haoyang Deng 1006146957
Gurpreet Mukker 1005985971
Fangren Xu 1005696711

Link to the video demonstration of our project:
https://www.youtube.com/watch?v=mmCg1Bn_yoI
Link to code:
https://drive.google.com/drive/folders/1FIIog_HAKlyf5MacsGvps7gjHWu8m0FR?usp=drive_link

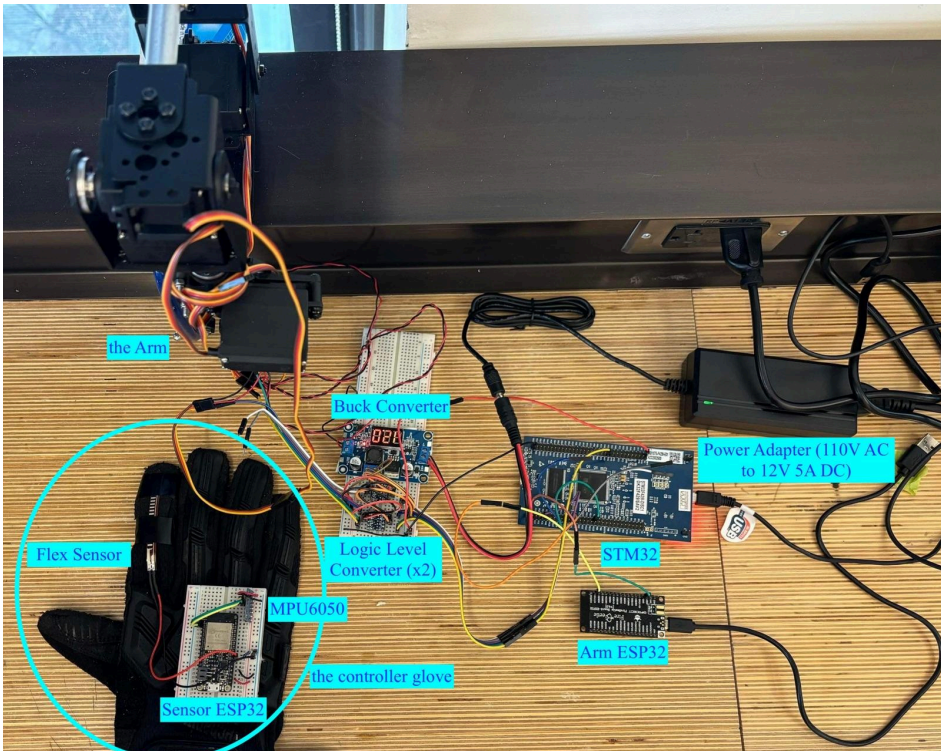

Figure 1. The robotic arm system and controller glove (with major parts marked)

Introduction

In our project proposal, we proposed to build a 5 Degree-of-Freedom (DoF) robotic arm controlled by hand gestures. Our initial goal was to control the robotic arm with hand gestures characterised by a controller glove, and implement additional features such as automatic object finding and a better user interface (LCD/touch screen). In the original proposal, the primary function of our robotic arm is to have 4 DoF (plus 1 DoF that rotates the gripper) and be able to pick up objects with its gripper. The secondary functions are to control the arm's motion based on gyroscope/accelerometer readings from hand gestures and the gripping is controlled by the flex sensor attached to a finger. Our objective was to enable wireless communication between the controller (glove) and robotic arm, implement touchscreen control and visualisation, and automatic object finding/picking.

The robotic arm that we actually built consists of four servo motors controlled by an STM32 and is able to move along six axes (±X, ±Y, ±Z) in a straight line. The STM32 is connected with an ESP32 with wires using serial communication and this ESP32 communicates with another ESP32 wirelessly via ESP NOW. The controller is based on an accelerometer, flex sensor, and an ESP32 for gesture characterization. The physical build of the robotic arm uses an off-shelf developer kit from Amazon [1]. In the final system, the gripper is removed because the servo motor is not powerful enough to operate it. The flex sensor is used to assist gesture characterization instead of controlling the gripper.

Overall, we were able to achieve: Gesture controlled robotic arm that is able to move in a straight line using inverse kinematics while implementing wireless communication between the control glove and the arm controller. Due to the reduction in our team size after the proposal was submitted, we did not have time for touch screen implementation and automatic object finding/picking.

The main body of our report describes the hardware used in the project, that includes the power supply structure, MCU selection and its peripheral setup, software implementation details of major parts using flow charts such as Inverse Kinematics and communication buffer setup. At last we evaluate the hardware and software choices that were made and reflect how they could be optimised in the future.

1. Microcontrollers, Sensors, and Communication

1.1 STM32 for the Robotic Arm

For our project we chose to use the SMT32F4 family of MCU. Specifically we use STM32F429I 144-Pin variant in a Discovery development kit. This controller was chosen for its high pin count that was required to implement all the features that were discussed in the project proposal. Since the final project design was trimmed down, much of the peripherals on the development kit were not used, such as LCD and touch Controllers. The number of pins on this MCU kit is much larger than what we need, if it's not for LCD or touch controllers, we would have chosen a STM32 with much fewer pins and peripherals (like the SMT32F4 family of chips on a Nucleo board), since we only need several pins for servo motor PWM and a couple sets of TX/RX pins to communicate with the Sensor ESP32.

From the developer's perspective, two major peripherals were used from STM32F429. A total of 4 channels were used from TIMER 2 and 3 to generate PWM signals. UART1 was used to communicate with the arm's ESP32 and PC.

The exact pins used for PWM channels and UART on STM32 can be found in appendix [A1].

Peripheral Setup

The Timers were set up as follows,

Peripheral Clock Frequency(F_IN)	72MHz
Prescaler(PSC)	10
AutoReload Register	65535

Table - 1 Timer SFR Register values to set 100Hz Frequency

$$PWM\ OUT(Hz) = \frac{F_{IN}}{(PSC+1)(AutoReload+1)} = 99.87Hz$$

UART5 was set up to communicate between the ESP32 and PC. The Rx channel of UART5 was connected to Tx of EPS32 to receive the data relating to the hand orientation and Tx pin of the UART5 was connected to the Rx of a USB to Serial Converter for debug messages.

Baud Rate	Word Length	Parity/Stop-Bit	DMA Stream	DMA Ch Direction
115200	8	0/1	0	Peripheral to Memory

Table - 2 UART Configuration

From a computational perspective, the FPU inside the MCU was very useful for many calculations that used the type ‘double’ variables for inverse kinematic equations. There was an extensive use of multiply,divide and square root operations which would have been very slow to compute without an FPU.

As STM32Cube doesn’t provide a library for operating/controlling general servo motors (which Arduino IDE offers), we had to measure the actual range of PWM signal (Duty Cycle) that the 996R servo motors would respond to, using square wave function generators and oscilloscopes. When we started testing the servo motors with STM32 to confirm the range of PWM signals, we noticed that the logic 1 of ~3V from STM32 is too low for our servo motors, making them only responsive in a range of 0 to 120° instead of a total range of ~200° range which was achieved when using 7.2V (peak to peak) PWM signals. To amplify the signal we used a 4-channel bi-directional logic level converter to convert the ~3V peak to peak PWM signal from STM32 to 7.2V peak to peak for the servo motors. On the logic converter, the reference for “voltage high” is supplied by the 7.2V output from the buck converter, and the “voltage low” is supplied by the ~3V output on STM32. Logic level shifters would have been obviously necessary for our system, had we noticed the logic 1 of both STM32 and ESP32 are around 3V. We didn’t notice this during the initial design (proposal) because the older versions of Arduino UNO (which we have worked with) use 5V as logic 1, which was proven to be sufficient for the peak to peak voltage of PWM signals for these servo motors (thus we falsely assumed the actual voltage of logic 1 doesn’t matter for PWM signals).

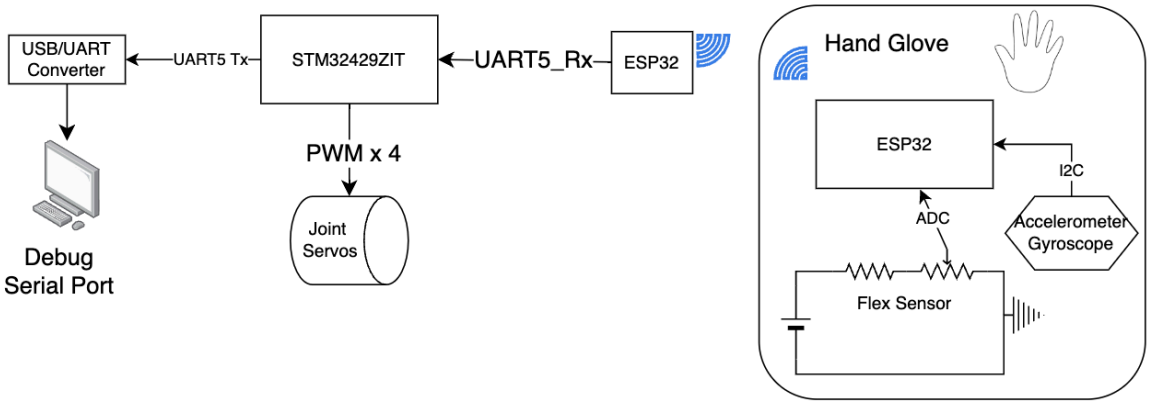

Figure 2. Communication Block Diagram

1.2 Two ESP32s for the glove and wireless communication with STM32

The main purpose of the MCU on the glove is to collect data from sensors and send it back to the STM32, which controls servo motors of the arm. ESP32-Wroom was chosen for its built-in WIFI and bluetooth hardware. ESP32 also provides the necessary ADC for flex sensor and I2C pins for MPU6050, in addition to access to libraries from Arduino IDE. We initially planned to use the HC-05 module to establish a bluetooth connection between STM32 and the Sensor ESP32. But due to its unavailability on the Digikey and mouser, we decided to instead use two ESP32s for wireless communication. We also decided to use ESP-NOW instead of the bluetooth protocol because of its ease of use in comparison to using bluetooth protocol.

ESP-NOW is a protocol developed by Espressif for direct wireless communication among ESP32 and ESP8266 boards without a router [2] [3]. The WiFi-based ESP-NOW can operate in 3 different modes of communication: one-way, two-way, and broadcasting. We used the broadcasting mode since it was the most simple out of all. In this mode, the sender will broadcast the message to the surroundings, and any ESP32 within the range can receive the message. For our project, the “eap_now.h” library in Arduino IDE made this wireless communication much easier to set up, especially in the following aspects:

1. This library allows the message transmitted to be of the same user defined structure (defined by a “struct”), which can contain a combination of different types of variables (up to 250 bytes), and the user doesn’t need to worry about how those messages are sent (as the function “esp_now_send” handles it) [3].
2. This library also simplified setting up actions that are triggered by sending/receiving a message, in the form of functions like “esp_now_register_send_cb” and “esp_now_register_recv_cb”.

1.3 Sensors

As shown in Figure 2, the sensors used in our project are: one MPU6050 (accelerometer, gyroscope, and temperature sensor) and one flex sensor; both are powered by the 3.3V from the Sensor ESP32.

The MPU6050 chip can measure angular velocity on three axes, linear acceleration on three axes, and temperature. For this project, the accelerometer (on MPU6050) is used to identify the orientation of the hand, the gyroscope and temperature sensor were not used. The MPU6050 communicates with the Sensor ESP32 via I2C (with libraries “Adafruit_MPU6050.h” and “Adafruit_Sensor.h”).

The flex sensor is connected to a 10kΩ resistor in a voltage divider setup, and the output voltage is read by A2 pin (12-bit ADC) of Sensor ESP32 into an integer. We tried using A0, but we noticed that ESP NOW communication disrupts (turns off) the ADC function on A0 (as the value from A0 is always 0 when ESP NOW is enabled).

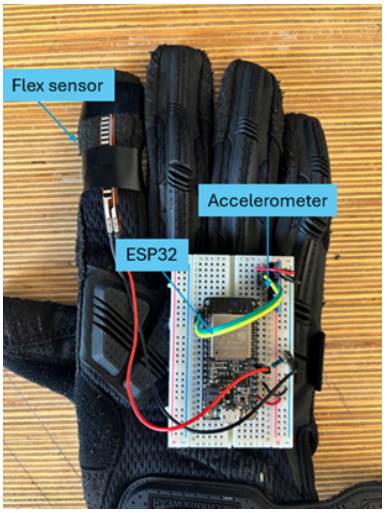

Figure 3. Glove and Sensors

1.4. Power Supply/Distribution

In our robotic arm system, other than the MCUs (one STM32 and two ESP32), the only components that require separate power supply are the 4 servo motors. Unlike the sensors (1 flex sensor and 1 accelerometer-gyroscope chip), which can be directly powered by the 3.3V power supply from the ESP32, the servo motors needed to be powered by a separate power supply due their high voltage and current requirements. We used an AC-DC converter with 12V 5A output and a buck converter to step down 12V to 7.2V in order to power the servo motors because this is more flexible and simpler than using a lab DC power supply (which is only available in labs) or any kind of batteries that required additional circuitry and needed to be charged.

2. Software Logic

The software was kept very simple and due to the lack of any feedback position sensors or encoders, the control loop was run in an open loop configuration. The accuracy of the end-effector position relied on the ability of servo motors to hold their position for a given PWM duty cycle.

2.1 The main loop (on STM32)

Brief description of software flow:

- Program starts off with initialization of core MCU peripherals such as Clock, GPIO’s, DMA, Timers etc. Next the data structures related to arm motors are initialised to specific known angles and orientation is set to neutral so that arm starts from a fixed known position.
- After the setup, the program enters the infinite loop. At first, it checks for any new data in the receiving buffer from the ESP32; if there is new data related to the hand orientation, the system state structure is updated accordingly.
- Next, it enters the routine that moves the arm by a very small increment. Based on the current hand orientation system state structure, the arm is moved along one of the axes in a given direction (see section 2.5 “User Interactions and Gesture Characterization”)
- To make sure that the end effector can reach the new position, the new angles calculated by the inverse kinematics are checked against the angle limits set on each motor. If the angles are not valid, the arm is not moved and the current position of the arm is maintained in the system state, else the arm is moved to the new position, and the system state is updated.

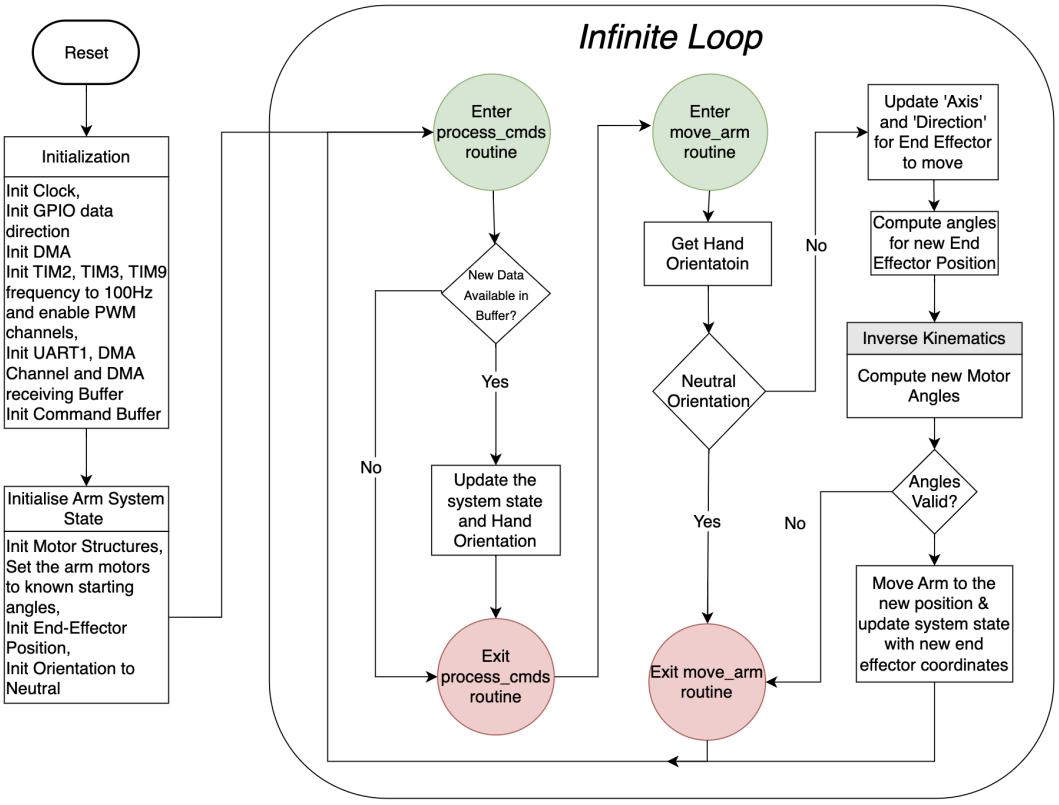

Figure 4 MCU peripheral initialisation and the infinite loop

Other than the infinite loop, the incoming serial data is handled by an Interrupt Service Routine that is initiated by the UART5 peripheral. This UART peripheral is connected to the receiving buffer using a DMA channel, so that the reception can be done with minimal CPU usage.

2.2 Code Optimisation

The most compute intensive part of the whole loop is the calculation of the motor angles using inverse kinematics. Because the function “calculate_angles()” needs to run once every 5ms to have the smoothest motion possible, it becomes paramount to optimise this function for speed.
Common Subexpression Elimination was used to pre-calculate expressions such as L2-Norms, summations, multiplications which are used multiple times in the function..

```
void calculate_angles(double vec[], double *alpha, double *beta, double *zeta, double *gamma) {
    double link0_vec[3] = {0, 0, LINK_0_LEN};
    double link3_vec[3] = {0, 0, -LINK_3_LEN};
    double r_dd_vec[3] = {vec[0] - link0_vec[0] - link3_vec[0], vec[1] - link0_vec[1] - link3_vec[1], vec[2] - link0_vec[2] - link3_vec[2]};
    double z_vec[3] = {vec[0] - link0_vec[0], vec[1] - link0_vec[1], vec[2] - link0_vec[2]};
    double z = norm(z_vec, 3);
    double zz = z*z;
    double r = norm(vec, 3);
    double r_sq = r*r;
    double link_sum = LINK_1_LEN + LINK_2_LEN;
    double link_sum_sq = link_sum*link_sum;
    double link3_len_sq = LINK_3_LEN*LINK_3_LEN;

    if (z > link_sum + LINK_3_LEN) {
        printf_("Error: Vector length is greater than the sum of the links\n");
        *alpha = 0;
        *beta = 0;
        *zeta = 0;
        *gamma = 0;
        return;
    }

    double r_dd = norm(r_dd_vec, 3);
    double r_dd_sq = r_dd*r_dd;
```

Reused Calculations

Figure 5 - Optimised compute intensive routine for speed

Reused Variable	double z	double zz	double link_sum	double link3_len_sq
# of times reused	7	5	6	3

Table - 3 Most reused variables to speed optimisation

The inverse kinematics were performed in a very crude manner for fast prototyping. In a commercial project, the inverse kinematics would have involved matrix maths for which there exists a plethora of optimised C libraries that would have made the calculations even faster.

2.3 RxBuffer Operation

As soon as the DMA channel receives the data from ESP32, an interrupt is generated, the data is stored in a temporary buffer and a 'save_cmd' routine is initiated. First it checks if the buffer has enough space for a new command, if it does, then checks for validity of the command and stores the command in the actual buffer for later processing.

```
typedef struct
{
    volatile uint8_t *buff;
    volatile uint16_t curr_pos;
    uint16_t read_pos;
    volatile uint8_t buffer_full;
    volatile uint8_t buff_init;
    volatile uint8_t catch_up;

}CommandBuffTypedef;
```

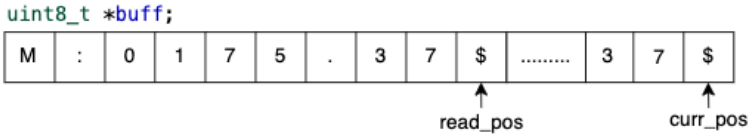

Figure 6. Data structure of the receiver buffer used for storing debug and orientation commands from hand

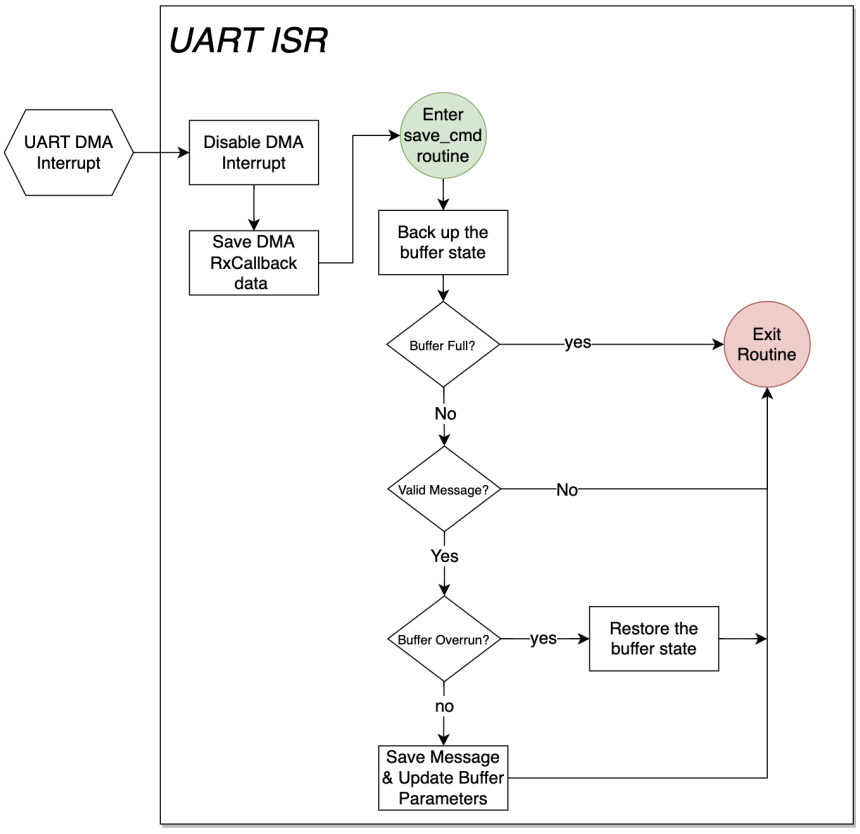

Figure 7 Interrupt Service Routine to store the incoming data from the UART_RX into the buffer

UART Data Transmission Implementation - To implement the Arduino-like ‘printf()’ function to make debugging easier, we used a lightweight library which we found on github [4]. To make it work, it required us to implement the putchar_() function that had HAL_UART_Transmit() inside it.

We defined our own protocol that is established using a set of commands which have predefined structure and length that end with ‘$’ such as in Table - 4. These commands allowed us to change the individual motor angle, send accelerometer data etc. for debugging purposes and the commands to send the orientation of the hand.

Command (Fixed Length of 10)	Mx:180.00$	O:0 $	A:x $
Description	Move motor ‘x’ to 180.00 degrees	Hand Orientation is ‘0’ or Neutral	Send measured acceleration in x-axis

Table - 4 Commands for debugging and communication with ESP32

2.4 Inverse Kinematics (IK)

Given the motor had 4 motors, inverse kinematics were required to calculate 4 motors angles for a given 3D point in space i.e.inverse kinematics had multiple solutions for a single point in 3D space. To overcome this, Link3 was fixed to point perpendicular to XY-plane when the vector $\vec{z} \leq LINK1 + LINK2$, otherwise the angle β was set to π. Following figures demonstrate these two cases. See appendix [A3] for IK equations for both of the cases.

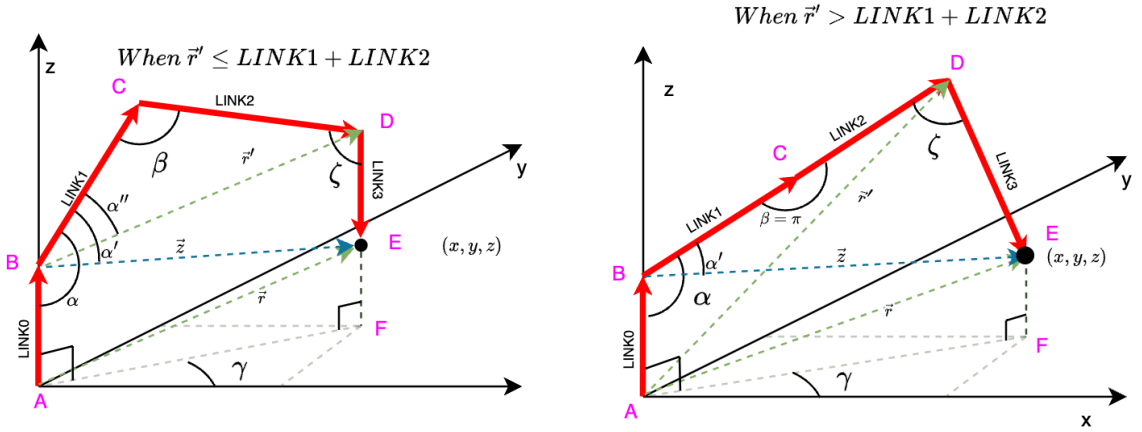

Fig 8 - Two cases of IK to make the solution singular for every position of end effector in 3D space

2.5 User Interactions and Gesture Characterization

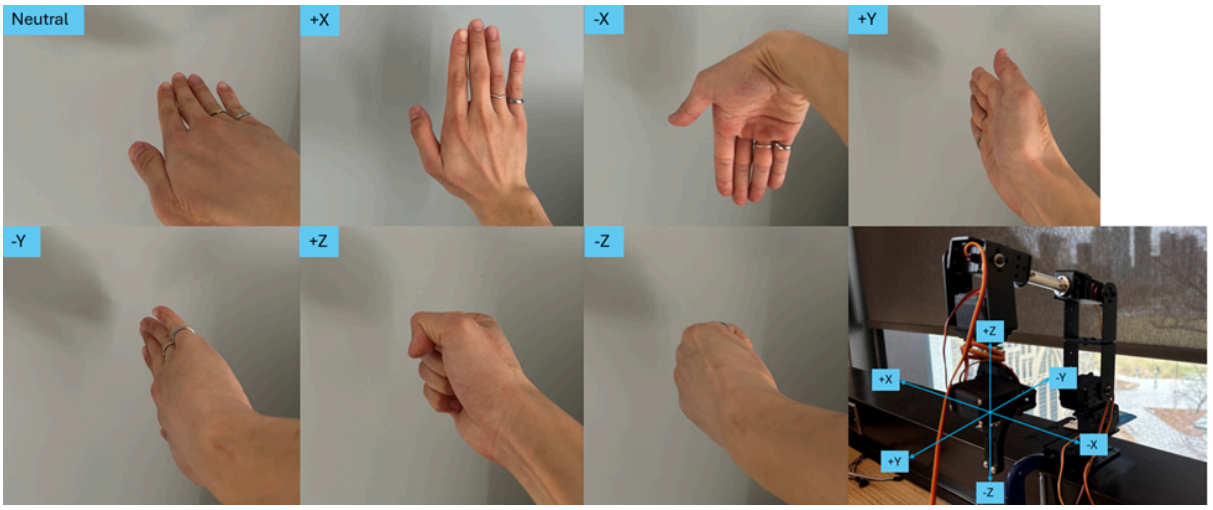

Figure 9. Gestures and arm's motion direction.

The user can interact with the robotic arm from the gestures shown in Figure 9**.** In gestures +Z and -Z, holding the fist will bend the flex sensor attached to the index finger on the glove, which changes the analog read pin A2. This will help to distinguish between the gesture +Y and -Y as the accelerometer readings are the same for +Y, +Z, and -Y, -Z respectively. Based on the characterised gesture, the robotic arm will move in a straight line on the axis indicated in Figure 9. The user will need to maintain the gesture for the robotic arm to move. If no gestures are recognized at all, the controller will consider the default gesture as neutral, where the robotic arm will maintain its current position.

3. Reflection and Evaluation

As mentioned before, The chosen MCU was much more capable that what we used it for, especially because we did not use LCD or touch controller. We could have used the SMT32F4 family of chips on a Nucleo board which would have provided us with an FPU but in a smaller 64-pin package.

Because we wanted to explore the STM32 family of devices in the project, we had to add two ESP32s to provide us with wireless functionality. Instead we could have just used two ESP32s since it has all the functionality we needed from the STM32. Some variants of ESP32 such as ESP32-S3 come with dual cores, one of the cores would have handled all the wireless communication which takes a lot of processing. The other core would have been used for Inverse Kinematics calculations without any processing delay from the wireless protocol processing. ESP32-S3 also comes with a single precision FPU that would have made calculations faster and it also has Bluetooth and Wifi functionality built in.

Developing embedded software differs significantly from conventional software development due to the dynamic nature of the hardware while code alterations are made. To mitigate potential erratic behaviour of the arm during code debugging, initial implementation of inverse kinematics and trajectory generation utilised Python for visualisation, ensuring logical integrity. Moreover, the IK code was designed hardware-agnostic, which enabled preliminary testing on a PC prior to deployment on the STM32 platform. Despite the necessity for code rewriting, this approach streamlined algorithmic testing by significantly reducing compile times. Direct testing on the arm would have incurred prolonged delays due to sluggish compile times in STM32CubeIDE, required constant hardware presence, and risked injuries and hardware damage from unpredictable movements.

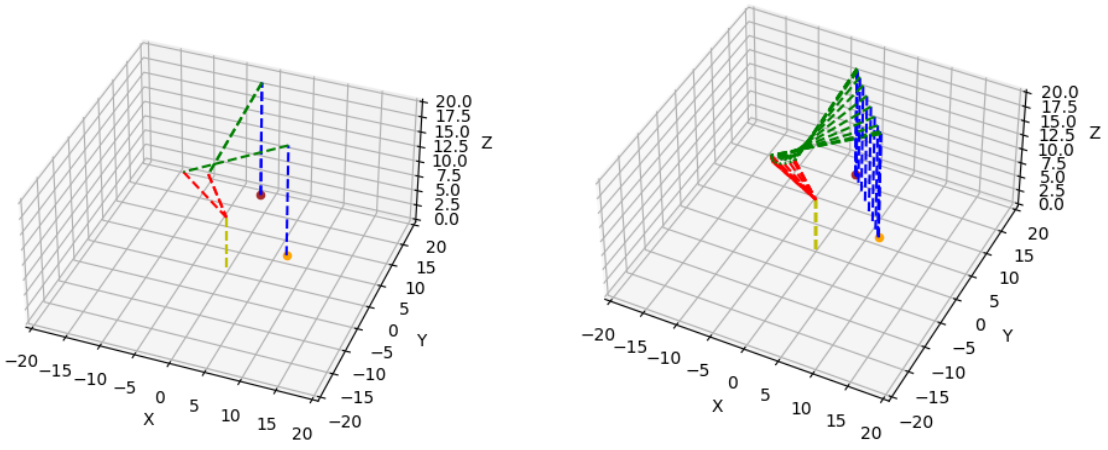

Figure.10 Initial and final position of arm (Left) and simulation of the trajectory to verify IK equations before implementing them into STM32 (Right)

As for the motors, the absence of accurate position feedback (or any kind of feedback) limits the implementation of more advanced features (like gravity compensation) on this robotic arm. Magnetic position encoders would have been very useful for feedback. But since such encoders could be expensive, we could have also used potentiometers on each joint. The potentiometers would have been read using ADC's on the MCU to provide a cheaper alternative of accurate position feedback to magnetic encoders.

Regarding the mechanical design of our arm, we didn't notice 2 of the more obvious mechanical problems until we fully assembled it, and these problems could have been identified if we included the mechanical aspect into our project. This arm design used a servo motor to directly rotate the entire arm without any kind of bearing to support, which caused the entire arm to wobble whenever this motor rotated. The gripper is made of 4-bar mechanisms, which we know to be incompatible with the servo motor from previous experiences (as the motor can't provide enough torque to open or close the gripper).

References

[1] https://www.amazon.ca/dp/B081FC4Q52?psc=1&ref=ppx_yo2ov_dt_b_product_details
[2] “ESP-NOW,” ESPRESSIF, https://www.espressif.com/en/solutions/low-power-solutions/esp-now (accessed Apr. 11, 2024).
[3] “Getting Started with ESP-NOW (ESP32 with Arduino IDE) | Random Nerd Tutorials,” Random Nerd Tutorials, Jan. 29, 2020. https://randomnerdtutorials.com/esp-now-esp32-arduino-ide/
[4] “Standalone printf/sprintf formatted printing function library”, https://github.com/eyalroz/printf

Appendix - A

[A1]

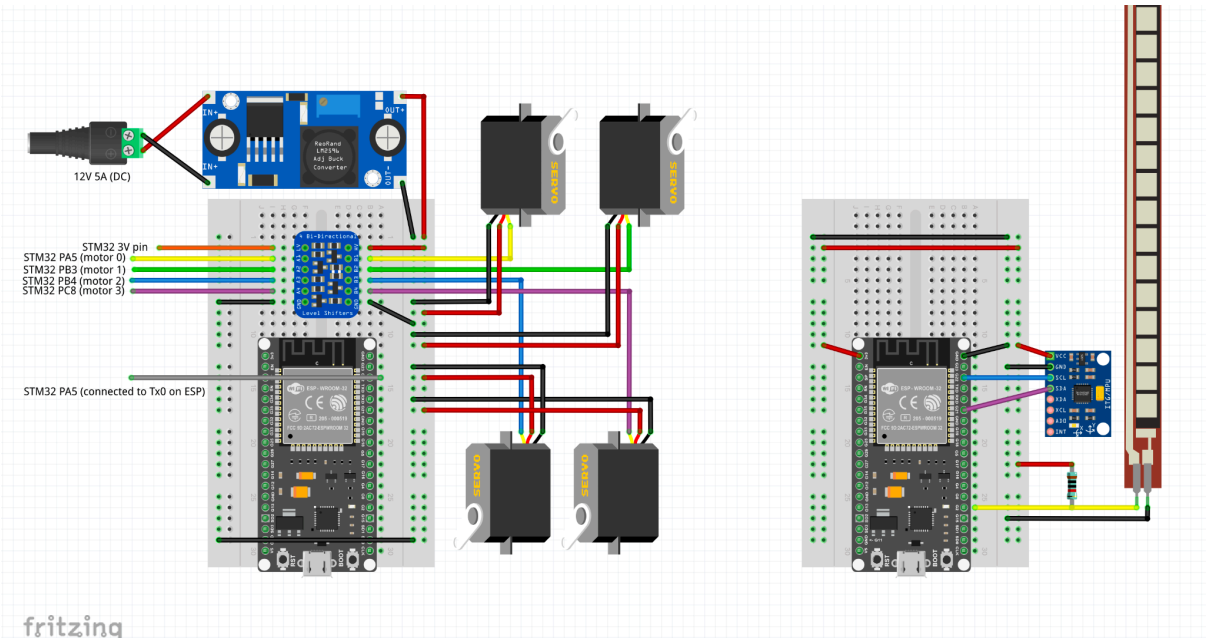

Figure.A1 Schematics of the Robotica Arm (Left) and the controller Glove (Right)

[A2] BOM

Description	Part Number	Quantity	Links
Robotic Arm Kit	NA	1	https://www.amazon.ca/dp/B081FC4Q52?psc=1&ref=ppx_yo2ov_dt_b_product_details
Dev Kit with LCD touch screen	STM32F429-DISC1	1	https://www.digikey.ca/en/products/detail/stmicroelectronics/stm32f429i-disc1/5731713
Level Shifters	CYT1076	1	https://www.amazon.ca/dp/B07V1YY8FH?psc=1&ref=ppx_yo2ov_dt_b_product_details
Accelerometer/ Gyroscope	SEN0142	1	https://www.mouser.ca/Prod

			uctDetail/DFRobot/SEN0142?qs=Zcin8yvlhnO0Rr0B1JJGiw%3D%3D
Flex Sensor	485-1070	1	https://www.mouser.ca/ProductDetail/Adafruit/1070?qs=GURawfaeGuBdaMpdIrCzxw%3D%3D
20W Adjustable DC-DC Buck Converter with Digital Display	DFR0379	1	https://www.mouser.ca/ProductDetail/DFRobot/DFR0379?qs=5aG0NVq1C4zPqXmLzkhO%252Bg%3D%3D
FIREBEETLE ESP32 IOT BOARD	DFR0478	1	https://www.digikey.ca/en/products/detail/dfrobot/DFR0478/7398878
120VAC - 12VDC 5A Power Supply	NA	1	Myhal Fab
10kΩ Resistor	NA	1	NA

[A3] Inverse Kine Equations

$\vec{z} \leq LINK1 + LINK2$	$\vec{z} > LINK1 + LINK2$
$\alpha'' = \cos^{-1}\left(\frac{\lvert\vec{r}\,'\rvert^2 + (Link1)^2 - (Link2)^2}{2 \cdot \lvert\vec{r}\,'\rvert \cdot Link1}\right)$ $\alpha' = \alpha'' + \cos^{-1}\left(\frac{r^2 + z^2 - (Link3)^2}{2 \cdot r \cdot z}\right)$ $\alpha = \alpha' + \cos^{-1}\left(\frac{(Link0)^2 + z^2 - r^2}{2 \cdot (Link0) \cdot z}\right)$ $\beta = \cos^{-1}\left(\frac{(Link1)^2 + (Link2)^2 - (r')^2}{2 \cdot (Link1) \cdot (Link2)}\right)$ $\zeta = 2\pi - \angle BAC + \angle EFA - \alpha - \beta$ $\gamma = \tan^{-1}\left(\frac{y}{x}\right)$	$\beta = \pi$ $\alpha' = \cos^{-1}\left(\frac{(Link1 + Link2)^2 + z^2 - (Link3)^2}{2 \cdot (Link1 + Link2) \cdot z}\right)$ $\alpha = \alpha' + \cos^{-1}\left(\frac{(Link0)^2 + z^2 - r^2}{2 \cdot (Link0) \cdot z}\right)$ $\zeta = \cos^{-1}\left(\frac{(Link1 + Link2)^2 + (Link3)^2 - z^2}{2 \cdot (Link1 + Link2) \cdot (Link3)}\right)$ $\gamma = \tan^{-1}\left(\frac{y}{x}\right)$