"PotRobotic 6 DoF"

California State University, Northridge

Department of Manufacturing Systems Engineering and Management

MSE 611 – Robotic & Automation

Submission Date:

May 19, 2023

Professor:

Adrian Sita

By:

Daniel Diaz

Contents

Abstract: 3
Introduction: 3
Objectives: 3
Building the Arm: 3
Set-up for Board: 5
Set-up for Programming and wiring: 5
Reflecting Troubleshooting: 9
Conclusion: 11
Result Video: 11
Works Cited 11
Appendix A: 12
Appendix B: 16
Appendix C: 18
Appendix D: 19
Appendix E: 19
Appendix F: 20
Appendix G: 21

Abstract:

This document encapsulates a series of discussions and revisions focused on a 6 Degrees of Freedom (DoF) robot project. Various challenges associated with servo motor control, coding adjustments, power supply issues, and communication breakdowns were addressed. It also touches on the failed attempt to integrate an HC-SR04 sensor due to either a power draw or electrical issue. In spite of difficulties, progress was made in terms of managing potentiometer readings, debugging I2C communication, and the ultimate creation of a functional robot controlled by a slider potentiometer.

Introduction:

This research paper chronicles the process of designing and building a six-degrees-of-freedom (6 DoF) robot. The project draws upon knowledge acquired throughout a robotics course and presents a practical implementation of the same. The key challenge was to transition the control of the robot from a potentiometer to an inductive automation system, while also incorporating essential safety measures. The journey encountered various hurdles, and the learning gained from addressing these obstacles forms the crux of the ensuing discussions.

Objectives:

The primary objective was to construct a robot with 6 DoF and control its movements initially via a potentiometer. The long-term vision was to achieve control through inductive automation, a considerably challenging endeavor. A critical aspect of the project was the safety system, consisting of an HC-SR04 sensor and a set of three LED lights: green, orange, and red, symbolizing safe, cautionary, and high-risk zones, respectively. The robot's design was such that it would stop on encountering a red signal, further emphasizing the priority placed on safety.

Building the Arm:

The first step in our project involves determining the types of arms or links that will be actuated by the motor, a process that requires careful consideration of their intended positions.

As such, it is advisable to plan the robot's orientation in advance, which will aid in establishing the optimal setup for the servos.

I used an unassembled kit featuring the aluminum brackets and MG966R servos. Navigating through the assembly process provided a valuable learning experience, highlighting the importance of identifying the servos' zero positions prior to installation. A comprehensive guide for determining whether the servo is sweeping correctly can be found in Appendix D.

To initialize the setup, start with securing the base to a fixed ground and zeroing the servo's position. This zeroing process can be achieved through a purchased servo tester or by writing your own code (see Appendix for details). Once the zero position is set, the servo effectively becomes a 'joint'.

Subsequently, a link is attached to the joint. This procedure is repeated until all desired joints are assembled. This systematic approach ensures an efficient and effective setup for the robotic system.

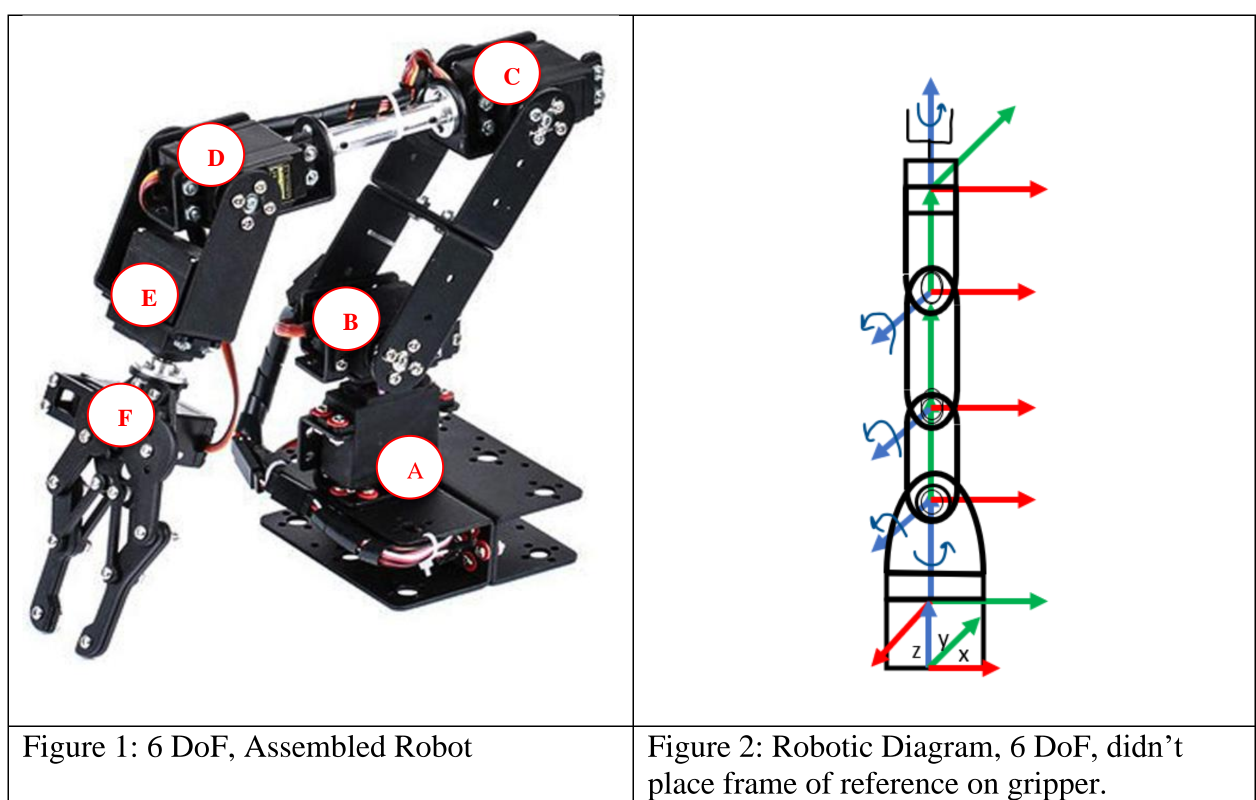

Figure 1: 6 DoF, Assembled Robot	Figure 2: Robotic Diagram, 6 DoF, didn't place frame of reference on gripper.

Set-up for Board:

Our initial choice for the project was the Arduino Uno due to its functionality and availability. However, I’ve encountered unforeseen issues that necessitated an alternative choice. I’ve considered the MKR-1000, but ultimately, its number of pins was insufficient for our project's needs.

As a result, I’ve shifted our attention to the Arduino Nano 33 IoT, a decision driven largely by its WiFi capability. This feature aligned with our goal of incorporating wireless control into the robot's operation. While I might not have achieved complete wireless control within our project's timeframe, it remains a promising avenue for future development.

It's important to bear in mind, however, that the Arduino Nano 33 IoT operates at 3.3V. This may cause compatibility issues with certain sensors that require a 5V operation. Therefore, it's critical to ensure that any sensors used in the project can function at this voltage level.

I’ve also incorporated a supplementary PCA9685 board to optimize our 6 DoF robot's performance. This board, communicating over the I2C bus, allows for the precise control of up to 16 servos with just two pins, thus preserving the Arduino Nano 33 IoT's GPIO pins for other tasks. The PCA9685 also handles PWM generation, freeing up the Arduino's processing power for more complex tasks.

One of the crucial features of the PCA9685 board is its use of capacitors for decoupling the power supply. This helps to filter out any noise that could disrupt the control signals to the servo motors, providing a more stable power supply. This in turn helps mitigate servo jittering, enhancing their overall responsiveness.

Set-up for Programming and wiring:

I’ve been inspired from a Youtuber, Scott Fitzgerald, and will be using some codes that already existed. I have, however, adapted and improved it to suit the needs of my project. One particular adjustment involved smoothing the readings from the potentiometer while the robot was idling. This is where I sought assistance from ChatGPT, which provided the appropriate code.

It's important to note that the Arduino Nano operates at 3.3V, a factor which required consideration during the code modification process (refer to fig 4 line 64). The adjustments are

illustrated in fig 3 lines 10-14 and fig 4 lines 52-64, which outline the utilization of a filter to smooth and average the readings.

```
1   #include <Wire.h>
2   #include <Adafruit_PWMServoDriver.h>
3
4   #define MIN_PULSE_WIDTH 540
5   #define MAX_PULSE_WIDTH 2350
6   #define FREQUENCY 50
7   #define numReadings 5 // Add the missing declaration for numReadings
8
9   // Create a structure to store the potentiometer's moving average data
10  struct PotentiometerData {
11    int readings[numReadings];
12    int readIndex;
13    int total;
14    int averagePotVal;
15  };
16
17  Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver();
18
19  // Define Potentiometer Inputs
20
21  int controlA = A0;
22  int controlB = A1;
23  int controlC = A2;
24  int controlD = A3;
25  int controlE = A6;
26  int controlF = A7;
```

Figure 3: First few lines, important to install the libraries first before running the code.

```
44  void moveMotor(int controlIn, int motorOut, PotentiometerData &potData)
45  {
46    int pulse_wide, pulse_width, potVal;
47
48    // Read values from potentiometer
49    potVal = analogRead(controlIn);
50
51    // Subtract the old reading and add the new reading
52    potData.total = potData.total - potData.readings[potData.readIndex] + potVal;
53
54    // Store the new reading in the array
55    potData.readings[potData.readIndex] = potVal;
56
57    // Increment the index, wrapping back to 0 if necessary
58    potData.readIndex = (potData.readIndex + 1) % numReadings;
59
60    // Calculate the average of the readings
61    potData.averagePotVal = potData.total / numReadings;
62
63    // Convert to 5V range
64    potData.averagePotVal = map(potData.averagePotVal, 0, 1023, 0, (1023 * 5) / 3.3);
65
66    // Convert to pulse width
67    pulse_wide = map(potData.averagePotVal, 0, 1023, MIN_PULSE_WIDTH, MAX_PULSE_WIDTH);
68    pulse_width = int(float(pulse_wide) / 1000000 * FREQUENCY * 4096);
69
70    // Print potentiometer value and motor position for debugging
71    Serial.print("Control ");
```

Figure 4: Line 52, 55, 58, 61, 64 modified, and edited 64 to compensate for 3.3V of nano

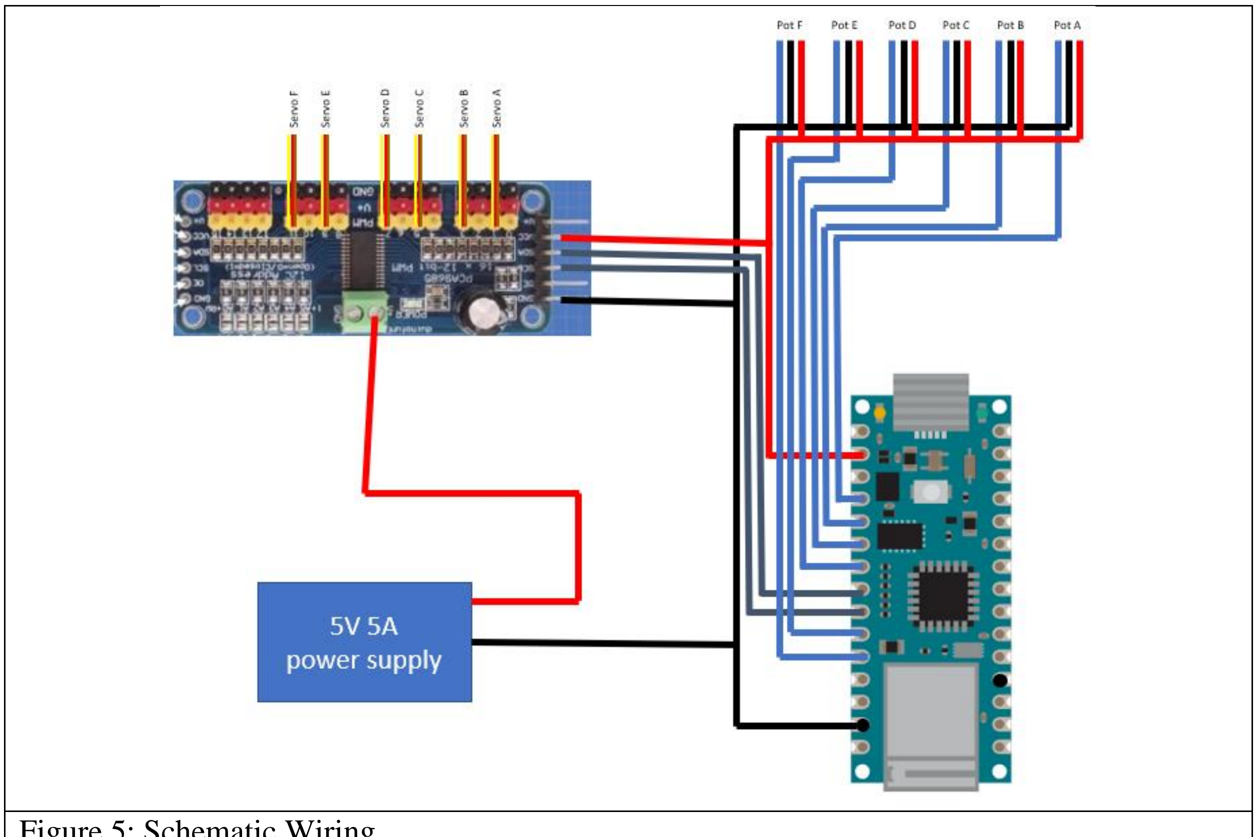

Figure 5: Schematic Wiring

Turning to the wiring phase, refer to figure 5 for a depiction of the setup I used. A critical point to remember when wiring the slider potentiometer is to correctly identify the ground, power, and signal connections as shown in figure 6. Lesson learned through this process of trial and error, which included a few painful lessons such as a hot potentiometer causing a slight burn. Be cautious and attentive to avoid such incidents.

Lastly, pay careful attention to the labeling of A, B, C, D, E, F for the Servos and Potentiometer in the diagrams. This labeling scheme is crucial for correctly setting up your robot and following along the code in appendix A. Also, see figure 7, to get an idea of how the final potentiometer would be set up. As shown, it would make sense to keep track of which is which. When in doubt, feel free to use tape tags for labeling.

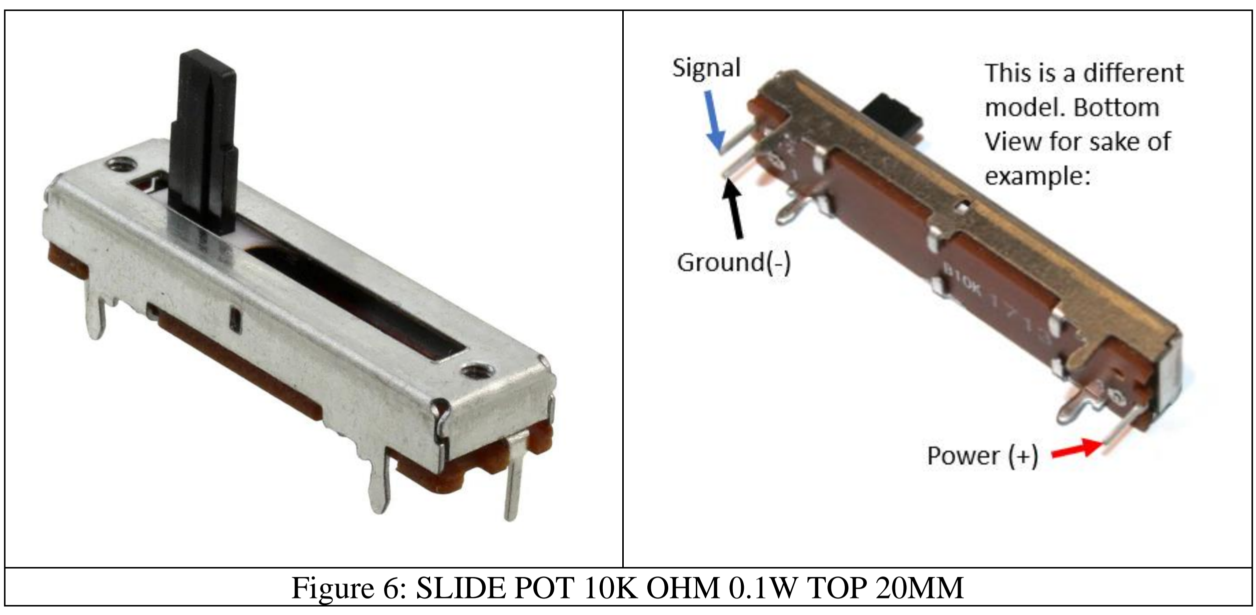

Figure 6: SLIDE POT 10K OHM 0.1W TOP 20MM

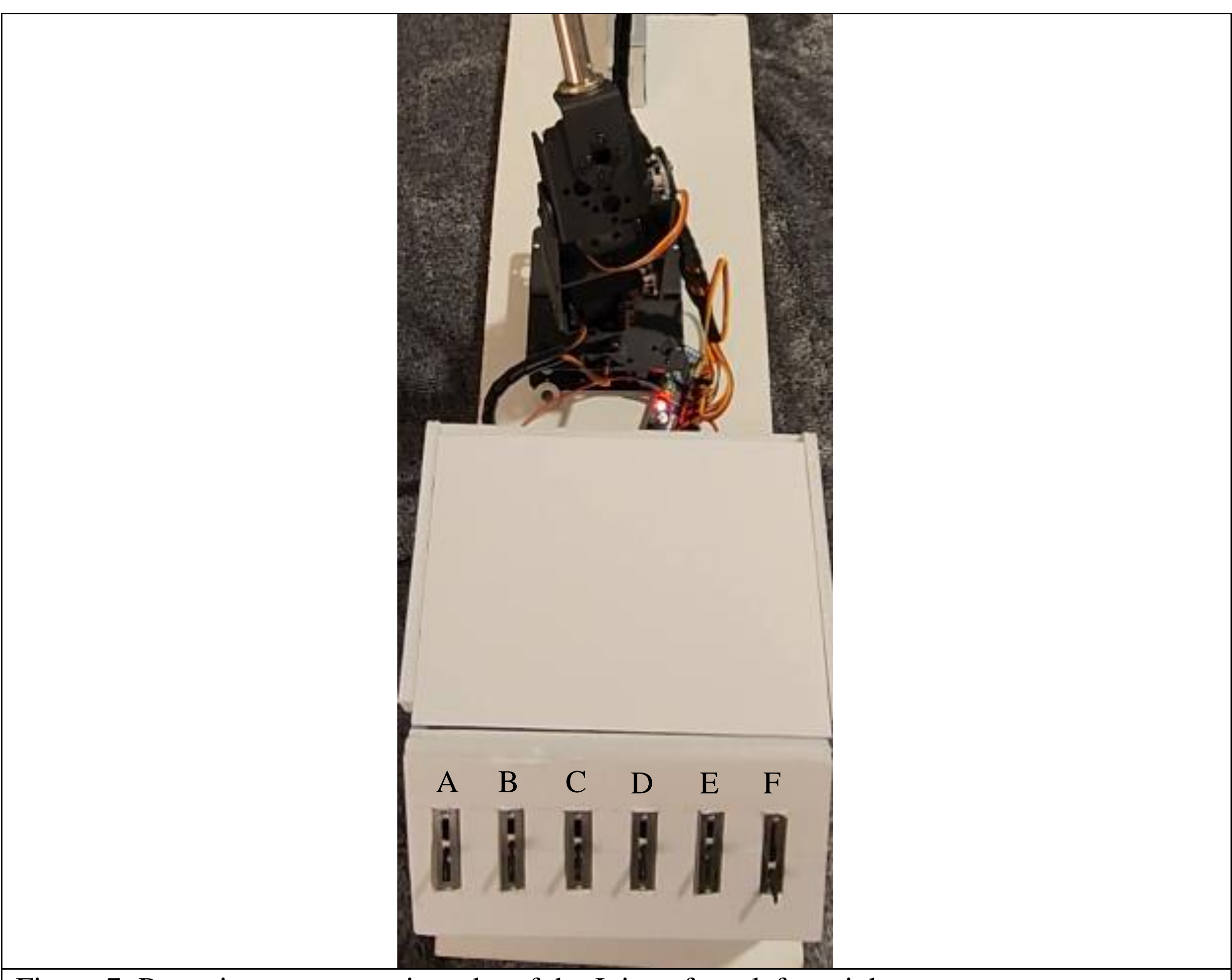

Figure 7: Potentiometer set up in order of the Joints, from left to right,

Reflecting Troubleshooting:

The construction phase of the project started with a critical step: setting the servo to zero. This step ensures that upon completion of the build, the robot moves in the right directions and avoids unnecessary complications (refer to appendices C & D for more information on coding considerations).

I encountered an issue with the servos behaving erratically due to insufficient power supply from the board. To address this, I used a set of 4x AA batteries, which resulted in abrupt robot movements. Realizing that I needed to reduce the noise, I decided to incorporate capacitors and purchased a PCA9685 board. This made the robot's movements smoother.

However, when I connected all the servos, the robot began to jitter and shut down. The solution was to use a 5V5A power supply. A new issue then arose: the robot exhibited erratic

movements while idling. I observed fluctuations in the potentiometer readings, which led me to reprogram the code to create a moving average filter. This significantly smoothed the robot's idling. Another point to consider was that the robot seemed to move every 5 degrees when adjusting the potentiometer, and moving it too quickly could cause the arm to overshoot.

The project faced another hurdle when the Arduino Uno stopped communicating with the PCA9685 board. After isolating each servo and verifying their functionality (see appendix C), I used a multimeter to ensure that voltage was being transmitted properly. I suspected a communication issue and, after reviewing Arduino's documentation, learned about the significance of I2C (SDA & SCL pins) for board communication. I worked with Chat GPT to create a debugging code for an I2C scanner (refer to appendix E for the code). As a result, I could identify the issue and decided to switch to Arduino Nano 33 IoT, which offers more pins, a requirement for another project of mine. Figures 8 & 9 provide a snapshot of what the debugging process looked like.

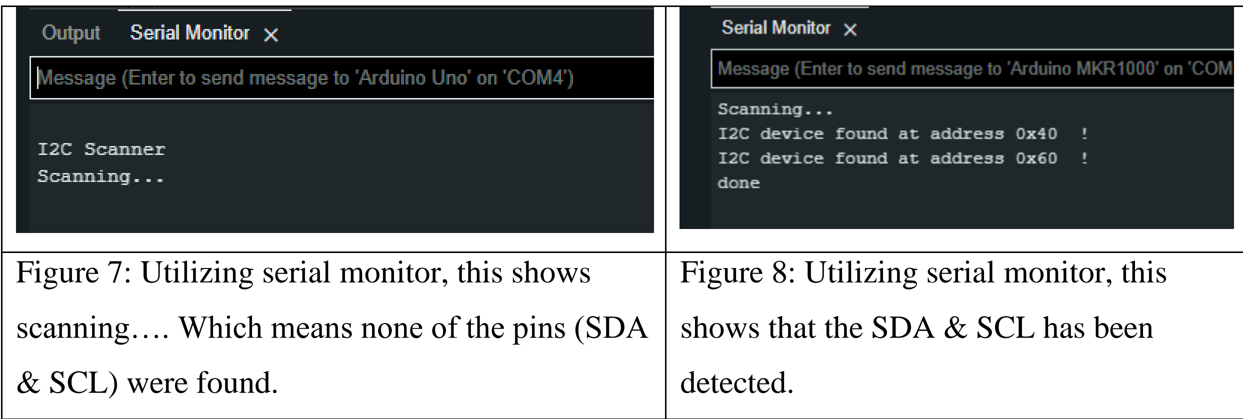

Figure 7: Utilizing serial monitor, this shows scanning…. Which means none of the pins (SDA & SCL) were found.	Figure 8: Utilizing serial monitor, this shows that the SDA & SCL has been detected.

In the initial plan, the HC-SR04 sensor was intended to enhance the safety features of the robot. While I developed a working standalone code for the HC-SR04 sensor (see Appendix F), problems arose when I attempted to integrate it into the main robotic control system (see Appendix G). The issues appeared to be either the HC-SR04 sensor drawing all the power from the Arduino Nano33IoT or an unidentified electrical issue. After browsing Raspberry Pi community forum, there were some mentions about the incompatibility, HC-SR04 sensor requires 5V otherwise it won't work. With that being said, I've taken that information and deduced that it won't work with Nano since it operates on 3.3V. Due to time constraints, it wasn't possible to diagnose and troubleshoot this issue further. Therefore, the decision was made to focus on establishing a functional robot controlled by a slider potentiometer.

Conclusion:

The journey of this project, from initial conception to eventual implementation, was riddled with trials and errors, leading to valuable learning experiences. These ranged from erratic servo behavior and issues with potentiometer readings, to a breakdown in I2C communication. Every obstacle necessitated a different approach to troubleshooting, such as code modification, employing an external power supply, and ultimately switching to a different microcontroller. An unforeseen challenge surfaced when attempting to integrate the HC-SR04 sensor into the main robotic control system due to a voltage incompatibility between the Arduino and the sensor. Despite this setback, the project concluded successfully with the creation of a robot controlled by a slider potentiometer. Future enhancements to this project could include resolving the remaining issues, refining the existing design, and reintegrating the initially planned safety features. This project, despite its challenges, served as a testament to the power of perseverance and adaptability in robotics engineering.

Result Video:

6DoF.mp4

Click on Icon or this link to see video:
For full address:
https://drive.google.com/file/d/1XcbmC8a0eqKUlDSPjq8j2AkhZzUOWI6R/view?usp=share_link

Works Cited

Community, Forum. “Hc sr04 at 3.3v.” *Raspberry Pi Forums*, 25 October 2015, https://forums.raspberrypi.com/viewtopic.php?t=124216. Accessed May 2023.

Fitzgerald, Scott. “Using Servo Motors with the Arduino.” *DroneBot Workshop*, 20 May 2018, https://dronebotworkshop.com/servo-motors-with-arduino/. Accessed May 2023.

McWhorter, Paul. “Arduino Tutorial 30: Understanding and Using Servos in Projects.” *Technology Tutorials*, 1 October 2019, https://toptechboy.com/arduino-tutorial-30-understanding-and-using-servos-in-projects/. Accessed May 2023.

McWhorter, Paul. "Arduino Tutorial 55: Measuring Distance With HC-SR04 Ultrasonic Sensor." *Technology Tutorials*, 31 March 2020, https://toptechboy.com/arduino-tutorial-55-measuring-distance-with-hc-sr04-ultrasonic-sensor/. Accessed May 2023.

"PCA9685 16-Channel 12-Bit PWM Servo Motor Driver with Arduino." *YouTube*, 27 January 2022, https://www.youtube.com/watch?v=_DgLt2Inr1E. Accessed May 2023.

Richetta, Andrea. "Nano 33 IoT." *Arduino*, https://docs.arduino.cc/hardware/nano-33-iot. Accessed May 2023.

"6DOF Manipulator Assembly Video." *YouTube*, 15 December 2021, https://www.youtube.com/watch?v=hTZ2z_C9dSU. Accessed 19 May 2023.

"Slide Pot 10K OHM 3.1W Top 20MM." *Bourns*, 31 March 2015, https://www.bourns.com/docs/Product-Datasheets/pta.pdf. Accessed 19 May 2023.

Zambetti, Nicholas. "A Guide to Arduino & the I2C Protocol (Two Wire)." *Arduino Documentation*, 16 May 2023, https://docs.arduino.cc/learn/communication/wire. Accessed May 2023.

Appendix A:

Board	Advantages	Disadvantages	Application	# of ADC pins
ESP8266 NodeMCU (V1, V2, V3)	• Low cost • Easy to use with Arduino IDE • Built-in WiFi	• Limited GPIOs • Less processing power • Limited RAM and Flash memory • No Bluetooth	• IoT devices • Home automation • Remote sensors • Web servers	1
ESP32 DevKitC (V1, V2, V3)	• More GPIOs than ESP8266 • Dual-core processor • Built-in WiFi and Bluetooth	• Slightly higher cost • Higher power consumption	• IoT devices • Home automation • Robotics • Wearable devices	18

	• More RAM and Flash memory • Easy to use with Arduino IDE		• Drones	
ESP32 WROVER	• Dual-core processor • Built-in WiFi and Bluetooth • More RAM and Flash memory • Easy to use with Arduino IDE • Camera interface, SD card support • Additional PSRAM	• Higher cost than ESP32 DevKitC	• IoT devices with high data processing • Advanced home automation • Machine learning projects • Image processing	18
ESP32-CAM	• Compact form factor • Built-in camera module • Built-in WiFi • Low cost • SD card support	• Limited GPIOs • No USB interface • Requires external programmer	• Surveillance cameras • Face recognition • Object tracking • Robotics • Drones	1 (Not exposed on the board, accessible through the pin header)
Arduino Uno	• Easy to use with Arduino IDE • Large community and libraries • Robust 5V I/O pins	• Limited GPIOs • No built-in WiFi or Bluetooth • Limited RAM and Flash memory • Less processing power	• Education • Hobbyist projects • Prototyping • Simple robotics	6
Arduino MKR1000	• Built-in WiFi	• Higher cost than Arduino Uno	• IoT devices • Home automation	7

	• Easy to use with Arduino IDE • More RAM and Flash memory than Uno • Low power consumption • Large community and libraries	• Limited GPIOs • 3.3V I/O pins	• Remote sensors • Battery-powered devices • Wireless communication	
Arduino Nano 33 IoT	• Compact form factor, similar to the original Arduino Nano. • Easy to use with Arduino IDE and extensive community support. • Built-in WiFi and Bluetooth for IoT applications. • 3-axis accelerometer and 3-axis gyroscope. • 3.3V I/O pins. • Lower power consumption, suitable for battery-powered projects.	• Compact form factor, similar to the original Arduino Nano. • Easy to use with Arduino IDE and extensive community support. • Built-in WiFi and Bluetooth for IoT applications. • 3-axis accelerometer and 3-axis gyroscope. • 3.3V I/O pins. • Lower power consumption, suitable for battery-powered projects.	• Compact form factor, similar to the original Arduino Nano. • Easy to use with Arduino IDE and extensive community support. • Built-in WiFi and Bluetooth for IoT applications. • 3-axis accelerometer and 3-axis gyroscope. • 3.3V I/O pins. • Lower power consumption, suitable for battery-powered projects.	8

References:

- ESP8266 NodeMCU:

  Official GitHub Repository: https://github.com/nodemcu/nodemcu-devkit

ESP8266 Datasheet: https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf

- ESP32 DevKitC:

  Official Espressif Documentation: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html

  ESP32 Datasheet: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

- ESP32 WROVER:

  Official Espressif Documentation: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/modules-and-boards.html#esp32-wrover-module

  ESP32 WROVER Datasheet: https://www.espressif.com/sites/default/files/documentation/esp32-wrover_datasheet_en.pdf

- ESP32-CAM:

  ESP32-CAM AI-Thinker Datasheet: https://www.ai-thinker.com/uploads/file/191112193120-0/ESP32-CAMProductSpecification.pdf

- Arduino Uno:

  Official Arduino Website: https://store.arduino.cc/usa/arduino-uno-rev3

  ATmega328P Datasheet: http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061A.pdf

- Arduino MKR1000:

  Official Arduino Website: https://store.arduino.cc/usa/arduino-mkr1000-wifi

  SAMD21 Datasheet: http://ww1.microchip.com/downloads/en/DeviceDoc/SAM_D21_DA1_Family_DataSheet_DS40001882F.pdf

  ATWINC1500 Datasheet: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42420-WINC1500-Low-Power-2.4GHz-IEEE-802.11-b-g-n-IoT-Network-Controller_Datasheet.pdf

- Arduino Nano 33 IoT

  Main Page: https://docs.arduino.cc/hardware/nano-33-iot?queryID=9d81d5fbd484e4c5018dc8ff78d60dec&_gl=1*lrjgnd*_ga*MjM1MDg3NjAyLjE2ODA2NjA1NjE.*_ga_NEXN8H46L5*MTY4NDM3NzAzNy4xNS4xLjE2ODQzNzcxNjAuMC4wLjA.

Technical Spec:
https://docs.arduino.cc/static/97dd3221a167cace69dcd032870e0d57/ABX00027-datasheet.pdf

Appendix B:

Program for the Robot

```
#include <Wire.h>
#include <Adafruit_PWMServoDriver.h>

#define MIN_PULSE_WIDTH 540
#define MAX_PULSE_WIDTH 2350
#define FREQUENCY 50
#define numReadings 5 // Add the missing declaration for numReadings

// Create a structure to store the potentiometer's moving average data
struct PotentiometerData {
  int readings[numReadings];
  int readIndex;
  int total;
  int averagePotVal;
};

Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver();

// Define Potentiometer Inputs

int controlA = A0;
int controlB = A1;
int controlC = A2;
int controlD = A3;
int controlE = A6;
int controlF = A7;

// Define Motor Outputs on PCA9685 board

int motorA = 1;
int motorB = 3;
int motorC = 5;
int motorD = 7;
int motorE = 9;
int motorF = 11;

void setup()
{
```

```
  Serial.begin(9600);
  pwm.begin();
  pwm.setPWMFreq(FREQUENCY);
}

void moveMotor(int controlIn, int motorOut, PotentiometerData &potData)
{
  int pulse_wide, pulse_width, potVal;

  // Read values from potentiometer
  potVal = analogRead(controlIn);

  // Subtract the old reading and add the new reading
  potData.total = potData.total - potData.readings[potData.readIndex] + potVal;

  // Store the new reading in the array
  potData.readings[potData.readIndex] = potVal;

  // Increment the index, wrapping back to 0 if necessary
  potData.readIndex = (potData.readIndex + 1) % numReadings;

  // Calculate the average of the readings
  potData.averagePotVal = potData.total / numReadings;

  // Convert to 5V range
  potData.averagePotVal = map(potData.averagePotVal, 0, 1023, 0, (1023 * 5) /
3.3);

  // Convert to pulse width
  pulse_wide = map(potData.averagePotVal, 0, 1023, MIN_PULSE_WIDTH,
MAX_PULSE_WIDTH);
  pulse_width = int(float(pulse_wide) / 1000000 * FREQUENCY * 4096);

  // Print potentiometer value and motor position for debugging
  Serial.print("Control ");
  Serial.print(controlIn);
  Serial.print(": ");
  Serial.println(potData.averagePotVal);

  // Control Motor
  pwm.setPWM(motorOut, 0, pulse_width);
}

// Initialize the moving average data for each potentiometer
PotentiometerData potDataA = { {0}, 0, 0, 0};
```

```
PotentiometerData potDataB = { {0}, 0, 0, 0};
PotentiometerData potDataC = { {0}, 0, 0, 0};
PotentiometerData potDataD = { {0}, 0, 0, 0};
PotentiometerData potDataE = { {0}, 0, 0, 0};
PotentiometerData potDataF = { {0}, 0, 0, 0};

void loop()
{
  // Control Motor A
  moveMotor(controlA, motorA, potDataA);
  // Control Motor B
  moveMotor(controlB, motorB, potDataB);
  // Control Motor C
  moveMotor(controlC, motorC, potDataC);
  // Control Motor D
  moveMotor(controlD, motorD, potDataD);
  // Control Motor E
  moveMotor(controlE, motorE, potDataE);
  // Control Motor F
  moveMotor(controlF, motorF, potDataF);
  delay(600); // Add a 600 ms delay between each iteration
}
```

Appendix C:

Code to sweep the servo to make sure it works

```
#include <Servo.h>

Servo myServo;  // create servo object

int pos = 0;  // variable to store the servo position

void setup() {
  myServo.attach(9);  // attaches the servo on pin 9
}

void loop() {
  for (pos = 0; pos <= 180; pos += 1) {  // sweeps from 0 to 180 degrees
    myServo.write(pos);  // tell servo to go to position
    delay(15);  // waits 15ms for the servo to reach the position
  }
  for (pos = 180; pos >= 0; pos -= 1) {  // sweeps from 180 to 0 degrees
    myServo.write(pos);  // tell servo to go to position
    delay(15);  // waits 15ms for the servo to reach the position
```

```
  }
}
```

Appendix D:

Code to “zero” the position before installing.

```
#include <Servo.h>

Servo myServo;  // create servo object

void setup() {
  myServo.attach(9);  // attaches the servo on pin 9
  myServo.write(0);   // set the servo to the 0-degree position
}

void loop() {
  // Nothing here. We just want to set the position once in the setup.
}
```

Appendix E:

I2C Scanner

```
#include <Wire.h>

void setup()
{
  Wire.begin();
  Serial.begin(9600);
  Serial.println("\nI2C Scanner");
}

void loop()
{
  byte error, address;
  int nDevices;

  Serial.println("Scanning...");

  nDevices = 0;
  for (address = 1; address < 127; address++ )
  {
    // The i2c_scanner uses the return value of
    // the Write.endTransmisstion to see if
    // a device did acknowledge to the address.
```

```
    Wire.beginTransmission(address);
    error = Wire.endTransmission();

    if (error == 0)
    {
      Serial.print("I2C device found at address 0x");
      if (address < 16)
        Serial.print("0");
      Serial.print(address, HEX);
      Serial.println("  !");
      nDevices++;
    }
    else if (error == 4)
    {
      Serial.print("Unknow error at address 0x");
      if (address < 16)
        Serial.print("0");
      Serial.println(address, HEX);
    }
  }
  if (nDevices == 0)
    Serial.println("No I2C devices found\n");
  else
    Serial.println("done\n");

  delay(5000); // wait 5 seconds for next scan
}
```

Appendix F:

Program to test HC-SR04

```
const int buttonPin = 2; // Button pin
const int ledPin = 4; // LED pin

int buttonState = HIGH; // current state of the button
int lastButtonState = HIGH; // previous state of the button
int ledState = LOW; // current state of the LED

void setup() {
  // Set up the pins as inputs and outputs
  pinMode(buttonPin, INPUT_PULLUP);
  pinMode(ledPin, OUTPUT);

  // Start serial communication
```

```
  Serial.begin(9600);
}

void loop() {
  // Read the state of the button
  buttonState = digitalRead(buttonPin);

  // If the button state has changed, toggle the LED
  if (buttonState != lastButtonState && buttonState == LOW) {
    ledState = !ledState;
    digitalWrite(ledPin, ledState);
    if (ledState == HIGH) {
      Serial.println("Button pressed - LED ON");
    } else {
      Serial.println("Button pressed - LED OFF");
    }
  }

  // Save the current button state as the last button state
  lastButtonState = buttonState;

  // Wait a bit before checking the button state again
  delay(100);
}
```

Appendix G:

Programming Robot with Safety, however, could not get the robot to operate.

```
#include <Wire.h>
#include <Adafruit_PWMServoDriver.h>

// HC-SR04 sensor pins
const int trigPin = 2;
const int echoPin = 3;

// LED pins
const int greenLedPin = 4;
const int yellowLedPin = 5;
const int redLedPin = 6;

// Button pin
const int buttonPin = 7;

// HC-SR04 variables
```

```
long duration;
float distance;

// Button state
int buttonState = 0;

// Define safety distances
const int safeDistance = 152; // 5 feet in cm
const int warningDistance = 61; // 2 feet in cm

#define MIN_PULSE_WIDTH 540
#define MAX_PULSE_WIDTH 2350
#define FREQUENCY 50
#define numReadings 5 // Add the missing declaration for numReadings

// Create a structure to store the potentiometer's moving average data
struct PotentiometerData {
  int readings[numReadings];
  int readIndex;
  int total;
  int averagePotVal;
};

Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver();

// Define Potentiometer Inputs

int controlA = A0;
int controlB = A1;
int controlC = A2;
int controlD = A3;
int controlE = A6;
int controlF = A7;

// Define Motor Outputs on PCA9685 board

int motorA = 1;
int motorB = 3;
int motorC = 5;
int motorD = 7;
int motorE = 9;
int motorF = 11;

void setup()
{
```

```
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
  pinMode(greenLedPin, OUTPUT);
  pinMode(yellowLedPin, OUTPUT);
  pinMode(redLedPin, OUTPUT);
  pinMode(buttonPin, INPUT_PULLUP);

  Serial.begin(9600);
  pwm.begin();
  pwm.setPWMFreq(FREQUENCY);

}
void moveMotor(int controlIn, int motorOut, PotentiometerData &potData)
{
  int pulse_wide, pulse_width, potVal;

  // Read values from potentiometer
  potVal = analogRead(controlIn);

  // Subtract the old reading and add the new reading
  potData.total = potData.total - potData.readings[potData.readIndex] + potVal;

  // Store the new reading in the array
  potData.readings[potData.readIndex] = potVal;

  // Increment the index, wrapping back to 0 if necessary
  potData.readIndex = (potData.readIndex + 1) % numReadings;

  // Calculate the average of the readings
  potData.averagePotVal = potData.total / numReadings;

  // Convert to 5V range
  potData.averagePotVal = map(potData.averagePotVal, 0, 1023, 0, (1023 * 5) /
3.3);

  // Convert to pulse width
  pulse_wide = map(potData.averagePotVal, 0, 1023, MIN_PULSE_WIDTH,
MAX_PULSE_WIDTH);
  pulse_width = int(float(pulse_wide) / 1000000 * FREQUENCY * 4096);

  // Print potentiometer value and motor position for debugging
  Serial.print("Control ");
  Serial.print(controlIn);
  Serial.print(": ");
  Serial.println(potData.averagePotVal);
```

```
  // Control Motor
  pwm.setPWM(motorOut, 0, pulse_width);
}

// Initialize the moving average data for each potentiometer
PotentiometerData potDataA = { {0}, 0, 0, 0};
PotentiometerData potDataB = { {0}, 0, 0, 0};
PotentiometerData potDataC = { {0}, 0, 0, 0};
PotentiometerData potDataD = { {0}, 0, 0, 0};
PotentiometerData potDataE = { {0}, 0, 0, 0};
PotentiometerData potDataF = { {0}, 0, 0, 0};

void loop()
{
  buttonState = digitalRead(buttonPin);

  if (buttonState == LOW) // When button is pressed, read sensor and control LEDs
  {
    // Clear the trigger pin
    digitalWrite(trigPin, LOW);
    delayMicroseconds(2);

    // Trigger the sensor by setting the trigPin high for 10 microseconds
    digitalWrite(trigPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(trigPin, LOW);

    // Read the echo pin and calculate the distance
    duration = pulseIn(echoPin, HIGH);
    distance = duration * 0.0344 / 2;

    // Control LED based on distance
    if (distance > safeDistance)
    {
      digitalWrite(greenLedPin, HIGH);
      digitalWrite(yellowLedPin, LOW);
      digitalWrite(redLedPin, LOW);
    }
    else if (distance <= safeDistance && distance > warningDistance)
    {
      digitalWrite(greenLedPin, LOW);
      digitalWrite(yellowLedPin, HIGH);
      digitalWrite(redLedPin, LOW);
    }
```

```
    else
    {
      digitalWrite(greenLedPin, LOW);
      digitalWrite(yellowLedPin, LOW);
      digitalWrite(redLedPin, HIGH);
    }
  }
  else // When button is not pressed, control motors if not in danger zone
  {
    if (distance > warningDistance)
    {
      // Control Motor A
      moveMotor(controlA, motorA, potDataA);
      // Control Motor B
      moveMotor(controlB, motorB, potDataB);
      // Control Motor C
      moveMotor(controlC, motorC, potDataC);
      // Control Motor D
      moveMotor(controlD, motorD, potDataD);
      // Control Motor E
      moveMotor(controlE, motorE, potDataE);
      // Control Motor F
      moveMotor(controlF, motorF, potDataF);
    }
    delay(600); // Add a 600 ms delay between each iteration
  }
}
```